UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

OSCAR NÚÑEZ MORI

TCP HolyWood

Dissertation submitted as partial fulfillment of the requirements for obtaining the degree of Master in Computer Science

Prof. Dr. Juergen Rochol
Advisor

Porto Alegre, February 2005.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Núñez Mori, Oscar
TCP HolyWood / Oscar Núñez Mori. - Porto Alegre: PPGC da UFRGS, 2005.
214p.: il.
Dissertation (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2005. Advisor: Juergen Rochol.
1. TCP. 2. Wired Networks 3. Congestion. 4. Throughput. 5. Jitter I. Juergen Rochol. II. Title.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-reitor: Prof. Pedro Cezar Dutra da Fonseca
Pró-Reitora Adjunta de Pós-Graduação: Profa. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Flávio Rech Wagner
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

DEDICATORY

To my two mothers, Blessed Virgin Mary and Maria Teolita Mori Hidalgo, for their endless love, tender and support.

To the Republic of PERU, my beloved Country, heart of the great Inca Empire, called the Empire of the Sun, where my story begun.

In memory of my father, Oscar Núñez Basurco, and my beloved son, Oscar Jesus Núñez Obregón I love you.

To Oscar Brian Núñez Obregón and Oscar Marcelo Núñez Garcia, my beloved children, for enduring with courage the weight of my absence, you are my life.

To my beautiful sister Teresa Núñez Mori, that is fighting as a brave Peruvian does, when is outside of her beloved country, God be with you.

To my school Colegio Nacional San José de Chiclayo, PERU, where I learned to love the science and the art, this humble work is your fruit.

To Brazilian People, for their hospitality and genuine friendship, receive my everlasting gratitude.

To you, dear reader, of this dissertation, God bless you and our Blessed Lady, Queen of Peace, brightens your life as the sun in the dawn.

ACKNOWLEDGEMENTS

As wrote Sir Henri Ridder Haggard in his magnificent book King Solomon's Mines; *‘‘Our future was so completely unknown, and I think the unknown and the awful always bring a man near to his maker "*.

With all the force of my soul and mind, Oh! My God, creator of the entire universe, Oh! Our Holy Father, thank you very much; thank you very much, our Lord Jesus Christ, our savior; thank you very much beloved mom Blessed Virgin Mary, *Queen of Peace*, that through your messages in Medjugorje, Bosnia-Herzegovina made my loneliness and suffering disappear filling my heart with heavenly love and humbleness (BLESSED VIRGIN MARY, 2005).

To my beloved and tender mother Dr. Maria Teolita Mori Hidalgo, I do not have words ... thanks again ten thousand times and more, mom, you trust in me when other let me alone and made possible this adventure.

To my auntie Rosa Jagosz that since my early years taught me to love the language of William Shakespeare.

To Dr. Eraldo Luiz Batista, Dr. Jose Heitor Machado Fernandes, Dr. Carlos Eduardo Nevado Díaz and the people of the Brazilian Health System thank you very much my dear friends, for keeping me alive in Porto Alegre.

To my dearest friends Eliene Sulzbacher and Captain Juergen Rochol, their support and style advices for writing this dissertation were crucial, friendship forever.

To Prof. S.Y. Wang and his team for the fastest and strong support, they gave us, when we were in our initial stage of finding an appropriate simulator.

To my dearest friends: Volkan Rivera Arenas, Francisco Valentin Reategui Upiachihuay, Ricardo Hernandes Fernandes, Tórgan Flores de Siqueira, Vicente Ströher Bürger, Paulo Ricardo Silveira Trainini, Sidinei Antonio Gobbi, Luis Henrique Girardi, Guilherme Pumi, Edson Antônio Werle, Jaçanã Ribeiro, Wilson Gavião Neto, Jacob Scharcanski, and Luciano Silva da Silva that putting aside their own works came to my aid as the hands of God to solve some of my problems, not only with their wit but with their precious sympathy.

To my dear friends Eliene Ricardo Iranço, Jorge Sanhudo dos Santos, Astrogildo, R. dos Santos, Luis Otavio Soares, Elisiane da Silveira Ribeiro, Luciano Motta, Julio Sanhudo da Rocha, Gilberto Aurelio Cunha de Farias, Henri Costa, Ida Rossi, Beatriz Haro, Carlos Alberto da Silveira Jr., Paulo Ricardo Pizzani de Jesus and Ângela Regina Rosa da Silva, Silvania Vidal de Azevedo, and Sulamar Figueira Marcelino, for their invaluable logistic support, vital for the conclusion of this work.

To my dear friends outside of Brazil: Jarowlaw Malek (Poland), Vicent Bricard-Vieu (France), Jorge Nuevo (Mexico), Xavier Marchador (Spain), Soo-Hyun Choi, that make this dissertation possible with their invalueble tips.

To my friends of GloMoSim and NS-2 electronic lists, you made this work possible.

To the authors of TCP Reno, the Standard; TCP Westwood, UCLA; TCP Vegas, University of Arizona; and TCP Veno, University of Hong Kong; thank you very much for inspiring this humble work.

To my dear Brazilian family of the Student's Houses of Porto Alegre in special CEFAV-UFRGS, CEU-UFRGS, CEUACA, that gave me a place where to sleep, dream and do this dissertation.

To the reviewers of this dissertation, Prof. Dr. Liane M. R.Tarouco, Prof. Dr. João Cesar Netto and Prof. Dr. Valter Roesler, for their contribution with clever insights and enrichment of this text, that simply the author could not note, thank you very much.

To all my dear friends in the UFRGS, Porto Alegre, Brazil, Peru, and the whole World, if I forgot someone I beg your pardon, I love you all, thank you very much for your comprehension and friendship.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS **10**

LIST OF FIGURES **11**

LIST OF TABLES **13**

ABSTRACT **14**

RESUMO **15**

1 INTRODUCTION **16**

1.1 Motivation **17**

2 BACKGROUND CONCEPTS **18**

2.1 Transmission Control Protocol **18**
2.2 TCP congestion control algorithms **19**
2.2.1 RTT Variance estimation 20
2.2.2 Exponential RTO Backoff 20
2.2.3 Karn's algorithm 20
2.2.4 Slow Start 21
2.2.5 Congestion avoidance 22
2.2.6 Fast Retransmit 23
2.2.7 Fast Recovery 24
2.3 Additional Algorithms **25**
2.3.1 Delayed Acknowledgments 25
2.3.2 Explicit Congestion notification 25
2.3.3 Large Initial Window 25
2.3.4 Nagle's Algorithm 26
2.3.5 Window Scaling 26
2.4 Detection of packet loss 26
2.5 Congestion 27

3 THE STANDARD AND RELATED WORKS **28**

3.1 TCP Reno, The Standard **28**
3.2 TCP New Reno **30**
3.3 TCP SACK **30**
3.4 TCP Santa Cruz **30**
3.5 TCP Vegas **31**

3.6 TCP Westwood……………………………………………………....... 31
3.7 TCP Veno………………………………………………………………… 32

4 TCP HolyWood …………………………………………………………. 33

4.1 The Model……………………………………………………………….. 34
4.2 The TCP HolyWood Code……………………………………………… 36
4.3 Window Behavior……………………………………………………….. 37
4.4 Performanced Metrics……..…………………………………………… 39
4.5 Factors..………………………………………………………………... 40

5 SIMULATION …………………………………………………………… 41

5.1 Verification……………………………………………………………. 41
5.2 Simulation Methodology………………………………………………… 44
5.3 Simulated Network Topologies…………………………………………. 45
5.4 Discrete-Event Simulators………………………………………………. 45
5.4.1 GloMoSim………………………………………………………………. 46
5.4.2 NCTUns 2.0…………………………………………………………….. 46
5.4.3 Network Simulator 2……………………………………………………. 46
5.5 Post Processing Stage……………………………………………………. 47
5.6 Limitations and assumptions…………………………………………….. 47

6 RESULTS AND ANALYSIS………………………………………………... 49

6.1 Methodology of the Analysis…………………………………………… 49
6.2 Terminology……………………………………………………………... 50
6.3 TCP HolyWood Versus TCP Reno……………………………………….. 51
6.3.1 Impact of Error Rate on Throughput…………………………………… 52
6.3.2 Impact of Propagation Time on Throughput…………………………… 55
6.3.3 Impact of Bottleneck Bandwidth on Throughput……………………… 60
6.3.4 Impact of Error Rate on Jitter………………………………………….. 64
6.3.5 Impact of Propagation Time on Jitter………………………………….. 69
6.3.6 Impact of Bottleneck Bandwidth on Jitter…………………………….... 74
6.3.7 Percentage of Lost Packets……………………………………………….. 78
6.3.8 Latency…………………………………………………………………. 80
6.3.9 Fairness………………………………………………………………… 80
6.3.10 Friendliness……………………………………………………………….. 83
6.4 TCP HolyWood Versus Other TCP Protocols………………………….. 84
6.4.1 Impact of Error Rate on Throughput………………………………….. 84
6.4.2 Impact of Propagation Time on Throughput………………………….. 88
6.4.3 Impact of Bottleneck Bandwidth on Throughput……………………….. 92
6.4.4 Impact of Error Rate on Jitter………………………………………… 96
6.4.5 Impact of Propagation Time on Jitter………………………………… 101
6.4.6 Impact of Bottleneck Bandwidth on Jitter……………………………. 106
6.4.7 Percentage of Lost Packets…….……………………………………….. 110
6.4.8 Latency………………………………………………………………... 112
6.4.9 Fairness And Friendliness…………………………………………...... 112

7 CONCLUSIONS AND FUTURE WORKS ……………………………... 113

REFERENCES………………………………………………………………... 115

APPENDIX A STATISTICS OF THE RESULTS…………………………… 120

A.1 Statistics of TCP HolyWood Versus TCP Reno ……………………... 121

A.1.1 Statistics of Average throughput versus Error Rate of TCP HolyWood and TCP Reno…………………………………………….. 121
A.1.2 Statistics of Average throughput versus Propagation Time of TCP HolyWood and TCP Reno……………………………………… 122
A.1.3 Statistics of Average throughput versus Bottleneck Bandwidth of TCP HolyWood and TCP Reno……………………………….. 123
A.1.4 Statistics of the Average Jitter Versus Error Rate of TCP HolyWood with TCP Reno………………………………………………… 124
A.1.5 Statistics of Average Jitter Versus Propagation Time of TCP HolyWood with TCP Reno………………………………………………… 126
A.1.6 Statistics of the Average Jitter Versus Bottleneck Bandwidth of TCP HolyWood with TCP Reno……………………………………. 128
A.1.7 Statistics of Average Throughput Versus Increasing Number of Flows – Fairness……………………………………………………... 129
A.1.8 Statistics Friendliness between TCP HolyWood and TCP Reno………… 130
A.2 Statistics of TCP HolyWood Versus Other Protocols………………….. 131
A.2.1 Statistics of Average Throughput Versus Error Rate of TCP HolyWood with other Protocols……………………………………… 131
A.2.2 Statistics of Average Throughput versus Propagation Time of TCP HolyWood with other Protocols………………………………… 133
A.2.3 Statistics of Throughput Versus Bottleneck Bandwidth of TCP HolyWood with other Protocols……………………………………… 134
A.2.4 Statistics of Average Jitter Versus Error Rate of TCP HolyWood with other Protocols……………………………………… 135
A.2.5 Statistics of Average Jitter Versus Propagation Time of TCP HolyWood with other Protocols……………………………………… 138
A.2.6 Statistics of Average Jitter Versus Bandwidth of TCP HolyWood with other Protocols…………………………………………………... 140
A.3.1 Percentage of Lost Packets of TCP HolyWood versus TCP Reno……... 141
A.3.2 Percentage of Lost Packets of TCP HolyWood versus other Protocols…. 143

APPENDIX B COMPUTER CODES………………………………………. 145

B.1 Main Code of TCP HolyWood for ns-2.1b8a…………………………... 145
B.2 Addendum to Main Code of TCP HolyWood for ns-2.1b8a………….. 153
B.3 Base Script : test-1-simple.tcl…………………………………………….. 154
B.4 Script of Network topology 1: test-1-simple-HOLYWOOD-150s.tcl…. 157
B.5 script of network topology 2: test-1-simple-10-flows-150s-Err-0.001-Fairness-x.tcl…………………... 160
B.6 AWK Script 1, to calculate Average, Standard Deviation, COV and Confidence Intervals: avg-s-cov-TESTED-END.awk……… 166
B.7 AWK Script 2, To calculate COV and Confidence Intervals with known Average and standard deviation………………………………... 168
B.8 AWK Script 3, to calculate the Lost packet Percent: lost-packet-stat-6O_PERCENT_TESTED_END.awk………………. 169
B.9 AWK script 4, to Calculate the Average of an Interval………………. 173
B.10 Gnuplot script 1: for one figure……………………………………... 174
B.11 Gnuplot script 2: for four figures…………………………………… 175

APPENDIX C COMPLETE SET OF RESULTS FIGURES……………….. 178

C.1 TCP HolyWood versus TCP Reno……………………………………….. 178
C.1.1 Impact of Error rate on Throughput……………………………………… 178
C.1.2 Impact of Propagation Time Throughput………………………………… 181

C.1.3 Impact of Bottleneck bandwidth on Throughput………………………… 183
C.1.4 Impact of Error rate on Jitter…………………………………………... 185
C.1.5 Impact of Propagation Time on jitter…………………………………... 188
C.1.6 Impact of Bottleneck bandwidth on Jitter……………………………… 190
C.2 TCP HolyWood Versus Other Protocols………………………………… 192
C.2.1 Impact of Error Rate on throughput of TCP HolyWood, TCP Westwood and TCP Vegas………………………………………………………… 192
C.2.2 Impact of Propagation time on throughput of TCP HolyWood, TCP Westwood and TCP Vegas……………………………………………….. 195
C.2.3 Impact of Bottleneck Bandwidth on throughput of TCP HolyWood, TCP Westwood and TCP Vegas……………………………………………….. 197
C.2.4 Impact of Error rate on Jitter of TCP HolyWood, TCP Westwood and TCP Vegas…………………………………………………………………... 199
C.2.5 Impact of Propagation Time on Jitter of TCP HolyWood, TCP Westwood and TCP Vegas………………………………………………... 202
C.2.6 Impact of Bottleneck Bandwith on Jitter of TCP HolyWood, TCP Westwood and TCP Vegas……………………………………………….. 204

APPENDIX D SOFTWARE INSTALATIONS……………………………. 206

D.1 Installation of NS-2.1b8a with TCP HolyWood in FreeBSD 4.10……... 206
D.2 Installation of Trace Graph 2.02 ....…………………………………….. 207

APPENDIX E TCP HolyWood 210

LIST OF ABBREVIATIONS

ACK	Acknowledgement
BER	Bit Error Rate
CWND	Congestion Window
E2E	End to End
ISO	International Organization for Standardization
RFC	Request for Comments
RTT	Round-trip time
SSTHRESH	Slow Start Threshold Size
TCP	Transmission Control Protocol
DLL	Data link layer
FIFO	First in first out
LAN	Local area network
LLC	Logical link control
MAC	Medium access control
PDU	Protocol data unit
PHY	Physical Layer

LIST OF FIGURES

Figure 2.1: TCP Slow Start in action …..…………………………………….. 22
Figure 2.2: TCP Slow Start and Congestion Avoidance behavior in Action ………………………..……………………………………. 23
Figure 2.3: TCP Fast Retransmit Algorithm…………………………………. 24
Figure 2.4: TCP Fast Recovery Algorithm…………………………………... 25
Figure 4.1: TCP HolyWood…………………………………………………… 36
Figure 4.2: Congestion window behavior……………………………………… 38
Figure 4.3: Congestion Window Behavior with different Error Rates…………… 38
Figure 5.1: Variation of *cwnd* and *sstresh* for TCP Westwood using the script *test-1-simple.tcl (*UCLA, 2004) …………………………………... 42
Figure 5.2: Variation of *cwnd* and *sstresh* for TCP Westood using the script *test-1-simple.tcl*-Verification…………………………………………... 42
Figure 5.3: TCP Westwood Bandwidth Estimation using script *test-1-simple.tcl* (UCLA, 2004)…………………………………………………... 43
Figure 5.4: TCP Westwood Bandwidth Estimation using script *test-1-simple.tcl*-Verification…………………………………………………... 43
Figure 5.5: Simulated Network Topology 1…………………………………… 45
Figure 5.6: Simulated Network Topology 2 ……………………..…………… 45
Figure 6.1: Throughput versus Time with Different Error Rates………………… 53
Figure 6.2: Average Throughput versus Error Rate……………………………. 54
Figure 6.3: Throughput Ratio versus Error Rate……………………………….. 55
Figure 6.4: Throughput versus Time with Different Propagation Time Values……………………………………………………………… 57
Figure 6.5: Average Throughput versus Propagation Time……………….……… 58
Figure 6.6: Throughput Ratio versus Propagation Time……………………….. 59
Figure 6.7: Throughput versus Time with different Bottleneck Bandwidth values. 61
Figure 6.8: Average Throughput versus Bottleneck Bandwidth………………….. 62
Figure 6.9: Throughput Ratio versus Bandwidth………………………………. 63
Figure 6.10: Jitter Versus Sequence Number for different Error Rates…………... 65
Figure 6.11: Average Jitter versus Error Rate……………………………………. 67
Figure 6.12: COV of the Average Jitter versus Error Rate……………………….. 68
Figure 6.13: Jitter Versus Sequence Number with different Propagation Time Values……………………………………………………………... 70
Figure 6.14: Average Jitter versus Propagation Time…………………………….. 72
Figure 6.15: COV of Average Jitter versus Propagation Time…………………… 73
Figure 6.16: Jitter versus Sequence Number with Different Bottleneck Bandwidth values…………………………………………………. 75
Figure 6.17: Average Jitter versus Bottleneck Bandwidth……………………….. 76
Figure 6.18: COV of the Average Jitter versus Bandwidth……………………… 77
Figure 6.19: Percentage of Lost Packets of TCP HolyWood versusTCP Reno….. 79

Figure 6.20: Fairness of TCP HolyWood and TCP Reno………………………… 81
Figure 6.21: Average Throughput versus Increasing Number of Flows………….. 82
Figure 6.22: Friendliness between TCP HolyWood and TCP Reno……………… 83
Figure 6.23: Throughput versus Time with different Error Rates over TCP HolyWood with other Protocols……………………………………. 85
Figure 6.24: Average Throughput versus Error Rate of TCP HolyWood with other Protocols……………………………………. 86
Figure 6.25: Throughput Ratio versus Error Rate of TCP HolyWood with other Protocols………………………………………………………. 87
Figure 6.26: Throughput versus Time with Propagation Time of 50ms of TCP HolyWood with other Protocols……………………………. 89
Figure 6.27: Average Throughput versus Propagation Time of TCP HolyWood with other Protocols………………..……………….………… 90
Figure 6.28: Throughput Ratio versus Propagation Time of TCP HolyWood with Other Protocols……………………………………….…………. 91
Figure 6.29: Throughput versus Time with different Bottleneck Bandwidth values of TCP HolyWood with other Protocols…………………….. 93
Figure 6.30: Average Throughput Versus Bottleneck Bandwidth of TCP HolyWood with other Protocols……………………………………. 94
Figure 6.31: Throughput Ratio Versus Bandwidth of TCP HolyWood with other Protocols……………………………………. 95
Figure 6.32: Jitter Versus Sequence Number with different Error Rates TCP HolyWood with other Protocols……………………………… 97
Figure 6.33: Average Jitter Versus Error Rate of TCP HolyWood with other Protocols……………………………………………………….. 99
Figure 6.34: COV of the Average Jitter Versus Error Rate of TCP HolyWood with Other Protocols………………………………………………… 100
Figure 6.35: Jitter Versus Sequence Number with Different Propagation Time Values of TCP HolyWood with other Protocols…………………... 100
Figure 6.36: Average Jitter Versus Propagation Time of TCP HolyWood with other Protocols…………………………………………………… 104
Figure 6.37: COV of Average Jitter Versus Propagation Time of TCP HolyWood with other Protocols……………………………………. 105
Figure 6.38: Jitter Versus Sequence Number with Bandwidth of 20Mbit/s of TCP HolyWood with other Protocols…………………………......... 107
Figure 6.39: Average Jitter Versus Bandwidth of TCP HolyWood with other Protocols…………………………………………………… 108
Figure 6.40: COV of the Average Jitter Versus Bandwidth of TCP HolyWood with other Protocols………………………………………………… 109
Figure 6.41: Percentage of Lost Packets of TCP HolyWood versus TCP Westwood and TCP Vegas…………………………………………. 111

LIST OF TABLES

Table 3.1: Implementation of TCP Congestion Control Measures……………….. 29
Table 4.1: Rapid start………………………………………………………….. 34
Table 6.1: Statistics of Latency of TCP HolyWood and TCP Reno……………… 80
Table 6.2: Statistics of Latency of TCP Vegas, TCP Westwood, and TCP HolyWood…………………………………………………………... 112

ABSTRACT

We introduce a new end-to-end, sender side Transport Control Protocol called TCP HolyWood or in short TCP-HW. In a simulated wired environment, TCP HolyWood outperforms in average throughput, three of the more important TCP protocols ever made, we are talking about TCP Reno, TCP Westwood, and TCP Vegas; and in average jitter to TCP Reno and TCP Vegas too. In addition, according to Jain's index, our proposal is as fair as TCP Reno, the Standard.

Keywords: TCP, Wired Networks, Congestion, throughput, jitter.

TCP HolyWood

RESUMO

Apresentamos um novo Protocolo de Controle de Transporte fim-a-fim, implementado somente do lado do transmissor, chamado TCP HolyWood ou, abreviadamente, TCP-HW. Em um ambiente de rede cabeada simulada, TCP HolyWood supera em vazão media três dos mais importantes protocolos TCPs já elaborados. Estamos falando de TCP Reno, TCP Westwood, e TCP Vegas; e em variação de retardo media ao TCP Reno bem como ao TCP Vegas. Alem disso, de acordo com o índice de Jain, nossa proposta e tão imparcial quanto o padrão, TCP Reno.

Palavras-Chave: TCP, Rede cabeada, congestão, Vazão, variação de retardo.

1 INTRODUCTION

"Sim, naturalmente vãos foram todos os homens que ignoraram a DEUS e que, partindo dos bens visíveis, não foram capazes de conhecer AQUELE que é, nem, considerando as obras, de conhecer o ARTÍFICE".

A Biblia de Jerusalem. (Sabedoria: 13,1)

The transmission control protocol (TCP) is a stream-oriented transport layer protocol that has several targets as for examples: adjust the transmission rate of packets to the available bandwidth, create a reliable connection by retransmitting lost packets, and keep off congestion at the network.

First TCP versions only owned a simple sliding window and flow control mechanism, without any congestion control. Nevertheless after a series of collapses in 1980's Van Jacobson introduced TCP Tahoe in 1988 (JACOBSON, 1988) and two year later introduced TCP Reno, considered as standard until now. What make majestic these two TCPs was their dynamic congestion controls and in the case of the latter served us as inspiration for this dissertation.

After the work of Jacabson since 90s until our days several TCPs proposal were arising. TCP became a continuous focus of research by the computer science community as shown by the several proposals presented, for example TCP Veno, TCP Westwood, TCP Santa Cruz, etc. all them trying to improve the throughput performance in different networks environments.

The Transport Control Protocol is located in the fourth layer of the OSI (Open System Interconnect) Reference Model (ISO 7489) surrounded with the upper fifth layer, Session, and the lower third layer, called Network.

In the dissertation we will demonstrate, that in a simulated wired environment, TCP HolyWood outperforms in Average throughput three of the more important TCP protocols ever made, we are talking about TCP Reno, the standard; TCP Westwood and TCP Vegas; and in average jitter to TCP Reno and TCP Vegas. In addition, our proposal is as fair as TCP Reno.

The structure of this dissertation is as follow: in chapter 2, we make a general review of background concepts. In chapter 3, review the current related works.

In chapter 4 is the reason of this dissertation, our proposal called "TCP HolyWood". In chapter 5, is the test-bed we used and the simulations we made in NS-2. In chapter 6, are the results, comparison and analysis of this work. Finally, in chapter 7, are our conclusion and future works.

1.1 Motivation

Oscar called Violeta, Hi darling, I wrote you a mail two hours ago, but you did not answer it. I beg your pardon Oscar, said Violeta; it was because I was making a commercial transaction by Internet and took me lots of time, 'cause the net really was heavy. Well darling if you read the email you will notice that I downloaded an excellent film ''The Reborn of the Empire of the Sun", that told the true story of the remaining descendants of the great Inca Empire in this century. I bought a peruvian wine as well, for this occasion; maybe you will give me the pleasure to watch the film with me tonight. I will be pleased said Violeta, so we will meet tonight kisses Oscar, kisses Violeta, see you later.

In this small dialogue, Violeta and Oscar were using transmission Control Protocol (TCP) without being aware of it, it was when Oscar sent an email, and when he downloaded the film and also when Violeta was making her commercial transaction. She also expressed her discomfort with the network, even with the highest speed link and updated technology congestions in the network may occur (JAIN, 1990). It showed us also that research to improve the TCP performance has to go on.

We may appreciate nowadays that TCP protocol is deployed world wide, and that simple dialogue is a very frequent situation. According to (ALTMAN; JIMENEZ, 2003), the TCP is the transport protocol that is responsible for the transmission of around 90% of the internet traffic, and understanding TCP is thus crucial for dimensioning the Internet

Now with the wireless technology in its different forms, wireless LAN and variations, Cellular Networks, Satellite Networks, wired-wireless cam networks, in general heterogeneous networks are being deployed in increasing rate, challenging situation may appear where TCP certainly will have to deal with.

Let us remember the word of Jain (1990) "*a scheme that works for one network may not work equally well for other networks"*. So we will still have different network environments, metrics, factor, schemes, arising communication technologies, that appropriated mixed will let us keep on playing with TCP for a long time in the search of the ideal TCP.

Finally, in the desire of conquer the space and with the developments of space technologies that are enabling the realization of space scientific missions, Interplanetary Internet is expected to be the next step in the design and development of deep space networks where TCP will play a primordial and decisive roll (AKYILDIZ, 2003) .

2 BACKGROUND CONCEPTS

"... Precisamos, entretanto, dar um sentido humano a nossas construções, e quando o amor ao dinheiro, ao sucesso nos estiver deixando cegos, saibamos fazer pausa, para olhar os lírios do campo e as aves do céu".

Erico Verissimo, *Olhai os Lírios do Campo.*

The concepts we are about to introduce in this chapter will let us to understand the technical structure of TCP HolyWood, our proposal, definitions as TCP and algorithms that form TCP are explained, as well as additional algorithms like Window Scaling, Delayed Acknowledgment and Large Initial Windows, not less important.

2.1 Transmission Control Protocol

The Transmission Control Protocol or well known as "TCP" is based on concepts first described by Cerf and Kahn (1974), since then until now, it is in a constant evolving process, and new challenging proposals are appearing for different environments as wired networks, wired-cam-wireless networks, wireless ad hoc networks, etc.

As stated by RFC 793 (DARPA, 1981), the primary purpose of TCP was to provide reliable logical circuit or connection service between pairs of processes. It does not assume reliability from the lower-level protocols (such as IP), so TCP must guarantee this itself. TCP can be characterized by the following facilities:

- **Basic Data Transfer:** From the application's viewpoint, TCP transfers a contiguous stream of bytes through the network. The application does not have to bother with chopping the data into basic blocks. TCP does this by grouping the bytes in TCP segments, which are passed to IP for transmission to the destination. In addition, TCP itself choose how to segment the data and TCP can forward that data at its own convenience. Sometimes, an application needs to be sure that all the data passed to TCP has actually been transmitted to the destination. For that matter, a push function is defined. It will push all remaining TCP segments still in storage to the destination host. The normal close connection function also pushes the data to the destination (DARPA, 1981) (RODRIGUEZ, 2003).
- **Reliability:** TCP assigns a sequence number to each byte transmitted and expects

a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. Since the data is transmitted in blocks (TCP segments), only the sequence number of the first data byte in the segment is sent to the destination host. The receiving TCP uses the sequence numbers to rearrange the segments when they arrive out of order, and to eliminate duplicate segments (DARPA, 1981) (RODRIGUEZ, 2003).

- **Flow Control:** The receiving TCP, when transmiting an ACK back to the sender, also shows to the sender the number of bytes it can receive beyond the last received TCP segment, without producing overrun and overflow in its internal buffers. This is transmitted in the ACK in the form of the highest sequence number it can receive without any trouble (DARPA, 1981) (RODRIGUEZ, 2003).
- **Multiplexing:** To allow many processes within a single host to deploy TCP communication facilities at the same time, The TCP gives a set of addresses or ports within each host (DARPA, 1981).
- **Logical Connection:** The reliability and flow control mechanisms described above require that TCP initializes and maintains certain status information for each data stream. The combination of this status, including sockets, sequence numbers and window sizes, is called a logical connection. The pair of sockets used by the sending and receiving processes uniquely identifies each connection (DARPA, 1981).
- **Full Duplex:** TCP provides for concurrent data streams in both directions (RODRIGUEZ, 2003).
- **Security:** the users of TCP may indicate the security and precedence of their communication (DARPA, 1981).

2.2 TCP Congestion Control Algorithms

Although a wide variety of TCP versions are used in the Internet, the current de facto standard and the most popular implementation is the TCP Reno protocol (STEVENS, 1994) (BALAKRISHNAN, et al., 1997) (PAXTON, 1997). We are going to review the control measures against congestion of this protocol.

The TCP congestion algorithms prevent a sender from overrunning the capacity of the network (for example, slower WAN links). TCP can adapt the sender's rate to network capacity and attempt to avoid potential congestion situations.

Because the value of the Retransmission timer (*RTO*, also called retransmission time out) has a significant effect on the reaction of TCP to Congestion (STALLING, 2000), TCP Reno considers three techniques that deal with the calculation of the retransmission time: *RTT* Variance estimation, Exponential *RTO* backoff and Karn algorithm. A very important matter also is the design of a proper value of the retransmission timer (*RTO*). If the value is too small, there will be many unnecessary retransmissions, wasting network capacity. If the value is too large, the TCP protocol will be slow, in answering to a lost segment.

In addition, TCP Reno also considers four techniques that deal with window management, because the size of TCP's send window can have a critical effect on whether TCP can be used efficiently without causing congestion and they are the Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery algorithms.

2.2.1 Round-trip time variance estimation

The round-trip time (*RTT*) variance estimation is also called Jacobson's algorithm, and is fundamental to TCP's timeout and retransmission. First TCP protocol have to quantify the *RTT* between transmitting a byte with particular sequence number and receiving an *ACK* that covers that sequence number, normally there is not a one-to-one correspondence between data segments and *ACKs* (STEVENS, 1994). According to RFC 793, we have:

$$Smoothed_RTT = \alpha .\ Smoothed_RTT + (1-\alpha).\ Measured\ RTT \qquad Eq.\ 2.1$$

Where α is a smoothed factor with a recommended value of 0.9.

The smoothed *RTT* is updated every time a new measure is made. 90% of each new estimate is from the previous estimate and 10% is from the new measurement (STEVENS, 1994). In order to give greater weight to more recent instances because they are more likely to reflect future behavior, and based on this, the *RTO* (Retransmission Time-Out) is:

$$RTO = \beta .\ Smoothed_RTT \quad \text{(DARPA, 1981)} \qquad Eq.\ 2.2$$

Where: β is a delay variance factor with a recommended value of two.

Calculating the *RTO* based on both the mean and variance provide much better response to wide fluctuations in the round-trip times, than just calculating the *RTO* as a constant multiple of the mean (STEVENS, 1994).

2.2.2 Exponential RTO Backoff

The exponential RTO backoff algorithm is used to determine what RTO value should be used on a retransmitted segment. However, when a TCP sender times out a segment, it has to retransmit that segment again and RFC 793 takes as true that the same *RTO* value will be deployed for this retransmitted segment, nevertheless for the reason that the time out is likely due to network congestion, keeping the same *RTO* value is a bad suggestion. In addition, in a scenario with several TCP sources transmitting into the Internet may cause a sustained or even an increasing congestion (STALLING, 2000).

In a backoff process, a TCP source increases its *RTO* each time the same segment is retransmitted. In a scenario of several TCP senders transmitting into the Internet and loosing their packet due to congestion, after a first retransmission of a segment on each affected connection, the TCP sources will all expect a longer time before trying a second retransmission. This may let the Internet time to clear the current congestion. If a second retransmission is needed, each TCP source will expect an even longer time before timing out for a third retransmission, giving the Internet an even longer period to get over (STALLING, 2000).

A simple technique for implementing *RTO* backoff is to multiply the *RTO* for a segment by a constant value for each retransmission:

$$RTO = q \times RTO \qquad Eq.\ 2.4$$

The equation before mentioned makes *RTO* to develop exponentially with each retransmission. Where the most commonly value of q is two.

2.2.3 Karn's Algorithm

The Karn's algorithm is used to determine which round-trip samples should be deployed as input to Jacobson's algorithm (STALLING, 2000). Karn and Partidge (1987) specified that, when a time out and retransmission happens, we are not able to update the *RTT* estimators when the ACK for the retransmitted data finally arrives. This

is because we do not know to which transmission the *ACK* corresponds. Perhaps the first transmission was delayed and not thrown away, or maybe the *ACK* of the first transmission was delayed. This is also called *retransmission ambiguity problem* (STEVENS, 1994).

Karn's algorithm solves this problem with the following rules (STALLING, 2000):

1. Do not use the Measured *RTT* for a retransmitted segment to update Smoothed*RTT* and smoothed mean deviation (A).
2. Compute the backoff *RTO* using equation 2.4 when a retransmission occurs.
3. Deploy the backoff *RTO* value for succeeding segments until an ACK arrives for a segment that has not been retransmitted.

2.2.4 Slow Start

We may say that the TCP sender uses ACKs as a "clock" to strobe new packets into the network. Since the TCP receiver can generate ACKs no faster than data packets can get through the network, the protocol is "self clocking". It means that the TCP automatically adjust the bandwidth and delay variations and have a wide dynamic range, but at the same time what makes a self-clocked system stable when it is running makes it hard to start – to get data flowing there must be ACKs to clock out packets but ACKs there must be data flowing (JACOBSON, 1988). To start the "clock", Jacobson (JACOBSON, 1988), developed a slow start algorithm.

In the Slow Start phase, the sender starts by transmitting one segment and waiting for its *ACK*. When that *ACK* is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four and so forth (see Fig. 2.1). This provides an exponential growth, although it is not exactly exponential, because the receiver may delay its ACKs, typically sending one *ACK* for every two segments that it receives. (RODRIGUEZ, 2003).

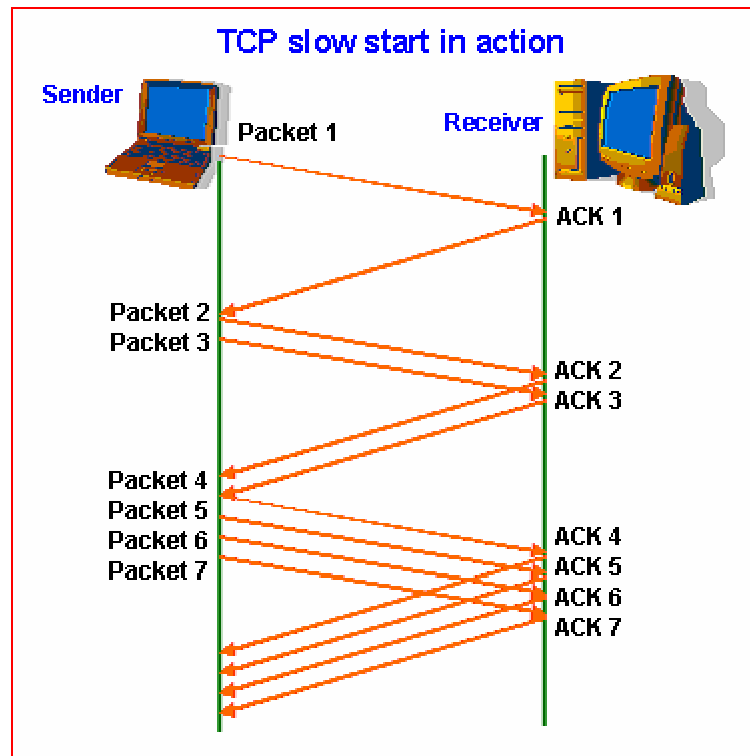

Figure 2.1: TCP slow start in action

2.2.5 Congestion Avoidance

The assumption of the algorithm is that packet loss caused by damage is very small (lesser than 1%). Therefore, the loss of a packet signals congestion somewhere in the network between the source and destination. There are two indications of packet loss, when a timeout occurs and when duplicate ACKs are received (RODRIGUEZ, 2003).

Congestion avoidance and slow start are independent algorithms with different objectives. Nevertheless, when congestion occurs TCP must slow down its transmission rate of packets into the network, and invoke slow start to get things going again. In practice, they are implemented together. Congestion avoidance and slow start require that two variables be maintained for each connection the Congestion Window or abbreviated *CWND* and the Slow Start Threshold Size or abbreviated *SSTHRESH* (RODRIGUEZ, 2003).The combined algorithm operates as follows (RODRIGUEZ, 2003):

1. Initialization for a given connection sets *CWND* to one segment and *SSTHRESH* to 65535 bytes.
2. The TCP output routine never sends more than the lower value of CWND or the receiver's advertised window.
3. When congestion occurs (timeout or duplicate *ACK*), one-half of the current window size is saved in *SSTHRESH*. Additionally, if the congestion is indicated by a timeout, *CWND* is set to one segment.
4. When new data is acknowledged by the other end, increase *CWND*, but the way it increases depends on whether TCP is performing slow start or congestion avoidance. If *CWND* is less than or equal to *SSTHRESH*, TCP is in slow start; otherwise, TCP is performing congestion avoidance.

Slow start continues until TCP is halfway to where it was when congestion occurred (since it recorded half of the window size that caused the problem in step 2), and then congestion avoidance takes over. Slow start has *CWND* begin at one segment, and incremented by one segment every time an ACK is received. As mentioned earlier, this opens the window exponentially: send one segment, then two, then four, and so on.

Congestion avoidance dictates that *CWND* be incremented by (*SEGSIZE*SEGSIZE) / CWND* each time an *ACK* is received, where *SEGSIZE* is the segment size and CWND is maintained in bytes. This is a linear growth of *CWND*, compared to slow start's exponential growth. The increase in CWND should be at most one segment each round-trip time. Regardless of how many *ACKs* are received in that *RTT* (RODRIGUEZ, 2003).

In the Figure 2.2, the value of Slow Start Threshold (*SSTHRESH*) is 8. Until this threshold is reached, TCP uses the exponential slow start procedure to expand the congestion Window. Afterward, *CWND* is increased linearly. Note also that it takes 11 round-trip times to recover to the *CWND* level that initially took 4 round-trip times to achieve.

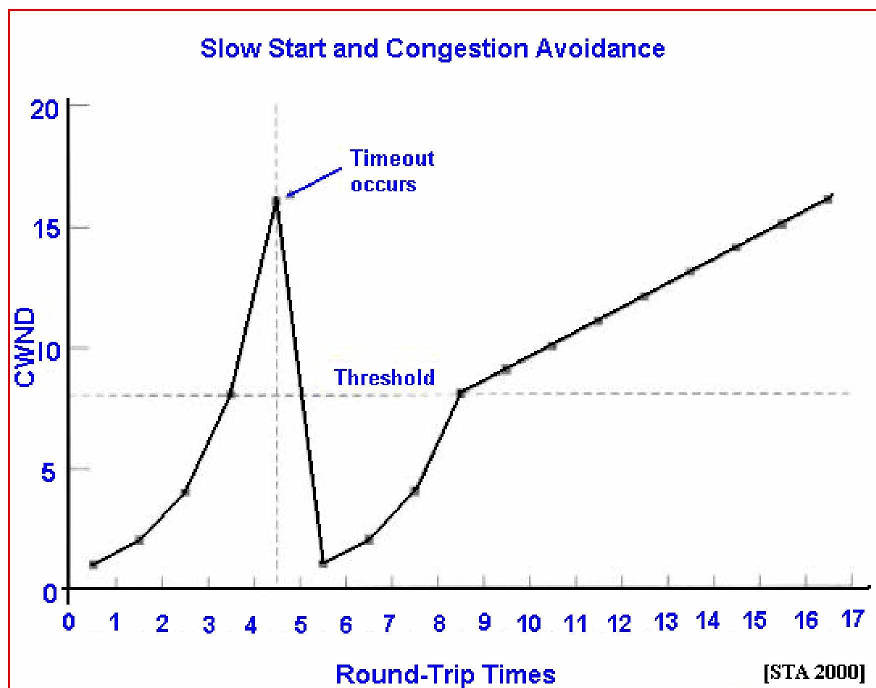

Figure 2.2: TCP slow start and congestion avoidance behavior in action

2.2.6 Fast Retransmit

Fast retransmit avoids TCP waiting for a timeout to resend lost segments. Note that TCP may generate an immediate acknowledgment (a duplicate *ACK)* when an out-of-order segment is received. The purpose of this duplicate *ACK* is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected. Since TCP does not know whether a lost segment or just a reordering of segments causes a duplicate *ACK*, it waits for a small number of duplicate *ACKs* to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate *ACKs* before the reordered segment is processed, which will then generate a new *ACK*. If three or more duplicate *ACKs* are received in a row, it is a strong indication that a segment has been lost as shown in Figure 2.3 TCP then

performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire (RODRIGUEZ, 2003).

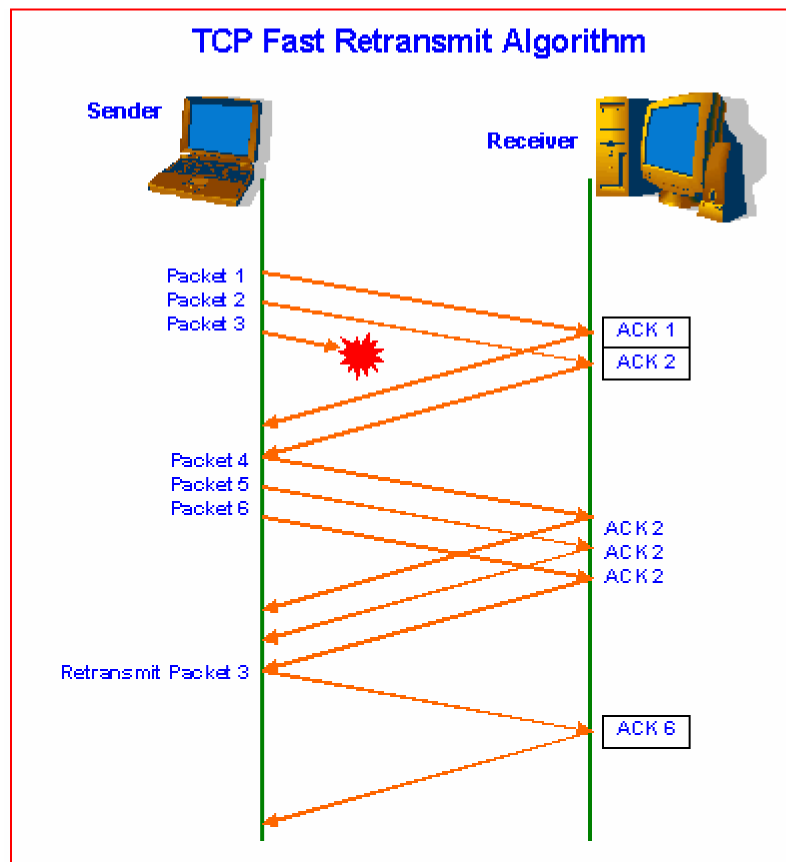

Figure 2.3: TCP Fast Retransmit Algorithm

2.2.7 Fast Recovery

After fast retransmit sends, what appears to be the missing segment, congestion avoidance, but not slow start, is performed. This is the fast recovery algorithm. It is an improvement, which allows high throughput under moderate congestion, especially for large windows. The reason for not performing slow start in this case is that the receipt of the duplicate *ACKs* tells TCP more than just a packet has been lost. Since the receiver can only generate the duplicate *ACK* when another segment is received, that segment has left the network and is in the receiver's buffer. That is, there is still data flowing between the two ends, and TCP does not want to reduce the flow abruptly by going into slow start (RODRIGUEZ, 2003). The fast retransmit and fast recovery algorithms are usually implemented together as follows (RODRIGUEZ, 2003):

1. When the third duplicate *ACK* in a row is received, set *SSTHRESH* to one-half the current congestion window, *CWND*, but no less than two segments. Retransmit the missing segment. Set *CWND* to *SSTHRESH* plus three times the segment size. This inflates the congestion window by the number of segments that have left the network and the other end has cached.

2. Each time another duplicate *ACK* arrives, increment *CWND* by the segment size. This inflates the congestion window for the additional segment that has left the network. Transmit a packet, if allowed by the new value of *CWND*.

3. When the next *ACK* arrives that acknowledges new data, set *CWND* to *SSTHRESH* (the value set in step 1). This *ACK* should be the acknowledgement of

the retransmission from step 1, one round-trip time after the retransmission. Additionally, this ACK should acknowledge all the intermediate segments sent between the lost packet and the receipt of the first duplicate *ACK*. This step is congestion avoidance, since TCP is down to one-half the rate it was at when the packet was lost as seen in Figure 2.4.

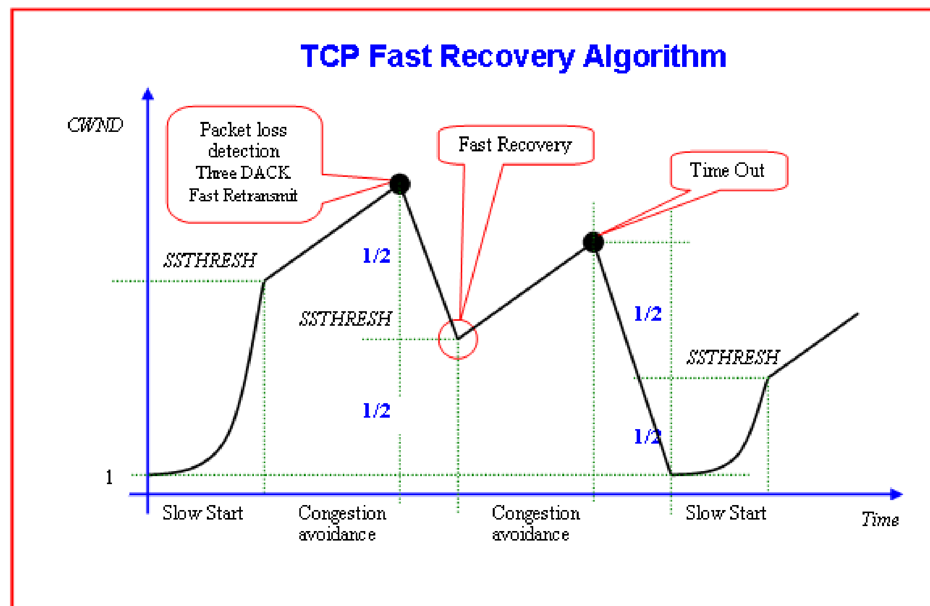

Figure 2.4: TCP Fast Recovery Algorithm

2.3 Additional Algorithms

Some algorithms that currently are being deployed with the TCP Reno, the Standard, in the several papers, we studied, and that we used in our proposal, TCP HolyWood, are as follows:

2.3.1 Delayed Acknowledgments:

It is defined in RFC 1122 (BRADEN, 1989), and permits that a TCP can abstain from transmitting an ACK for each incoming data segment, nonetheless rather should send an ACK for every second full-sized data segment received. If a second data segment is not received within a given timeout (not to exceed 500ms.) an ACK is transmitted (ALLMAN; FALK, 1999).

2.3.2 Explicit Congestion Notification

In RFC 2481 (RAMAKRISHNAN; FLOYD, 1999) is defined a technique in which a router can transmit an explicit message stating that the network is becoming congested, rather than dropping a segment, In (FLOYD, 1994), this Explicit Congestion Notification is further detailed. This algorithm is still a non-standardized extension of TCP (ALLMAN; FALK, 1999).

2.3.3 Large Initial Windows:

In RFC 2581(ALLMAN; PAXSON; STEVENS, 1999) allows TCP to use an initial congestion window of 1 or 2 segments. Nonetheless in RFC 2414 (ALLMAN; FLOYD; PARTRIDGE, 1998) describes an experimental TCP algorithm that increase the initial

congestion window to 3-4 segments, being founded on the segment size. This algorithm is still a non-standardized extension of TCP (ALLMAN; FALK, 1999).

2.3.4 Nagle's Algorithm

The Nagle algorithm is defined in RFC 896 (NAGLE, 1984), and is deployed to combine many small bits of data produced by applications into larger TCP segments. The Nagle Algorithm has been shown to reduce the number of segments transmitted into the network, but also interfere with HTTP and NNTP (Network News Transport Protocol) protocols as well as the delayed acknowledgement strategy, thus reducing performance (ALLMAN; FALK, 1999). The beauty of this algorithm is that is self-clocking, the faster the ACKs come back, the faster the data is sent (STEVENS, 1994).

2.3.5 Window Scaling

The standard TCP header in RFC 793 (DARPA, 1981), limits the advertised window size to 64 KB. Which is not adequate in many situations, therefore a network path that exhibits a long delay and/or a large bandwidth may require windows size of more than 64 KB. To solve the problem is defined in the RFC 1323 (BRADEN; BORMAN, 1992) allowing the use of window size of more than 64 KB. Windows scaling extension of TCP can lead to more rapid use of the TCP sequence space (ALLMAN; FALK, 1999).

2.4 Detection of packet loss

Packet loss can be detected in one of two ways, either by the reception at the TCP sender of "triple-duplicate" acknowledgments, i.e. four ACK's with the same sequence number (it depends on the implementation, for example in Linux the loss indications occur after two duplicate ACK's); or time-outs (PADHYE, 2000). In the case where the TCP sender times out, this happens when packets or acknowledgements are lost, and less than three duplicate acknowledgements are received. The sender waits for a period denoted by T_0 and then retransmit non-acknowledged packets. Following a time-out, the congestion window is reduced to one, and one packet is thus resent in the first round after time-out. In the case that another time-out occurs before successfully retransmitting the packet lost during the first time-out, the period of time-out doubles to $2T_0$ this doubling is repeated for each unsuccessful retransmission, we mean $4T_0$, $8T_0$, $16T_0$, $32T_0$, until $64T_0$ (PADHYE, 2000).

It is important to mention in this chapter the key points of Stallings (2004, P. 239) about TCP traffic Control:

- TCP uses a sliding-window flow control mechanism that allows multiple segments to be in transit at a time.
- The throughput on a TCP connetion depends on the window size, propagation delay, and data rate.
- Although the TCP credit-based mechanism was designed for end-to-end flow control, it is also used to assist in internetwork congestion control. When a TCP entity detects the presence of congestion in the Internet, it reduces the flow of data onto the Internet until it detects an easing in congestion.
- A Key element in TCP congestion control is the value of the retransmission timer. A variety of algorithms and strategies have been introduced to make the most effective use of the timer.
- Another important factor in TCP Congestion Control is the management of the window for segment transmission. Again, a variety of strategies are used to optimize performance.

- The most recent TCP congestion control technique, explicit congestion notification, involves explicit signals to the TCP endpoints from routers in the Internet indicating the onset of congestion Stallings (2004, P. 239).

2.5 Congestion

We cannot finish this chapter without understanding, the Congestion in Computer network. As stated by Forouzan (2004), congestion in a network may occur if the load on the network --The number of packets sent to the network – is greater than the capacity of the network – the number of packet a network can handle. Therefore, Congestion Control refers to the mechanisms and techniques to control the congestion and keep the load below the capacity. Congestion happens in any system that involves waiting; as well as occurs because routers and switches have queues – buffers that hold the packets before and after processing. An example of Congestion Control in TCP is also provided by Forouzan (2004):

> As we have said, an internet is a combination of networks and connecting devices (e.g., routers). A packet from a sender may pass through several routers before reaching its final destination. A router has a buffer that stores the incoming packets, processes them, and forward them. If a router receives packets faster than it can process, congestion might occur and some packets could be dropped. When a packet does not reach the destination, no acknowledgement is sent for it. The sender has no choice but to restransmit the lost packet. This may create more congestion and more dropping of packets, which means more retransmission and more congestion. A Point may then be reached in which the whole system callapses and no more data can be sent. TCP therefore need to find some way to avoide this situation. (FOROUZAN, 2004, P639-640).

Moreover, here is where entering into action the algorithms before mention in this chapter to try to avoid the Congestion.

3 THE STANDARD AND RELATED WORKS

"I can shake off everything if I write, my sorrows disappear, my courage is reborn".

Anne Frank, *her Diary*

There is still nowadays, a great effort of scientific community to improve the performance of the Internet because of its tremendous growth in the world; in that respect enhancements of TCP Reno and new proposals for different network environments are arising, we choose some of them, all with the same characteristic of maintaining End-End TCP Semantics.

3.1 TCP Reno - The Standard

We cannot start to talk about TCP Reno if we do not talk about TCP Tahoe. Historically, TCP Tahoe was the first modification to TCP. The TCP Tahoe protocol includes Slow Start, Congestion Avoidance, and Fast Retransmit. It also implements a Round Trip Time (*RTT*) based estimation of retransmission time out. In the Fast Retransmit mechanism, a number of successive (usually set at three), duplicate acknowledgments *(DACKS*) carrying the same sequence number triggers a retransmission, without waiting for the associated timeout event to occur.

The window adjustment strategy for this “early timeout” is the same as for a regular timeout; Slow Start is applied. The problem, however, is that Slow Start is not always efficient, especially if the error was purely transient or random in nature, and not persistent. In such a case, the shrinkage of the congestion window is, in fact, unnecessary, and renders the protocol unable to utilize the available bandwidth of the communication channel during the subsequent phase of window re-expansion (TSAOUSSIDIS; BADR, 2000).

In addition to TCP Tahoe, TCP Reno introduces Fast Recovery in conjunction with Fast Retransmit. The idea behind Fast Recovery is that duplicate acknowledgments (*DACK*) are an indication of available channel bandwidth since a segment has been successfully delivered. This, in turn, implies that the congestion window (*CWND*) should actually be incremented. Receiving the threshold number of duplicate acknowledgments Fast Recovery is triggered; the sender retransmits the missing segment then, instead of entering Slow Start as in TCP Tahoe, increases *CWND* by the *DACK* threshold number.

Thereafter, and for as long as the sender remains in Fast Recovery, *CWND* increased

by one for each additional *DACK* received. This procedure is called “inflating” *CWND*. The Fast Recovery stage is completed when an acknowledgment (*ACK*) for new data is received. The sender then halves *CWND* (“deflating” the window), sets the congestion threshold to *CWND*, and resets the *DACK* counter.

In Fast Recovery, *CWND* is thus effectively set to half its previous value in the presence of *DACKS*, rather than performing Slow Start as for a general retransmission timeout. TCP Reno, however, is not optimized for multiple segment drops from a single window (TSAOUSSIDIS; BADR, 2000). The algorithm of TCP Reno has two parameters: current window size *CWND* and slow start threshold *SSTHRESH*, which are updated as follows (STEVENS, 1997) (FENG; MIN; CHUANSHAN, 2001)

1. When new data is acknowledged and if $CWND \leq SSTHRESH$, set $CWND = CWND+1$; *(Slow Start Phase).* Else set, $CWND = CWND + 1/CWND$.
2. A Packet loss can be detected either by the reception of duplicate acknowledgements, i.e., four ACK with the same sequence number, or via timer expiry (time-out), as was mentioned in the section 2.4. Upon timer expiry, the algorithm goes into slow start phase as before: $SSTHRESH = CWND/2$, *and* $CWND = 1$. Otherwise, fast retransmission and fast recovery are taken up.
3. When triple duplicate ACKs are received in a row, TCP Reno performs a retransmission of what appears to be the missing packet, without waiting for a retransmission timer to expire, and set; $SSTHRESH = CWND/2$, and $CWND = SSTHRESH + 3$.
4. For every other duplicate *ACK* received, set $CWND = CWND + 1$. Transmit a packet if allowed by *CWND* (Fast Retransmit Phase)
5. When the next *ACK* arrives, that acknowledges new data, set $CWND = SSTHRESH$, and a new cycle starts - Fast Recovery Phase.

The following table 3.1 shows exactly what algorithms are currently involved with Tcp Reno; none of these algorithms extends or violates the original TCP standard, published in RFC 793 (DARPA, 1981).

Table 3.1: Implementation of TCP Congestion Control Measures

MEASURE	RFC 1122	TCP TAHOE	TCP RENO
RTT Variation Estimation	X	X	X
Exponential RTO Backoff	X	X	X
Karn’s Algorithm	X	X	X
Slow Start	X	X	X
Dynamic Window Sizing on Congestion [Congestion Avoidance]	X	X	X
Fast Retransmit		X	X
Fast Recovery			X

Source: STALLINGS, 2004, p. 249.

3.2 TCP New Reno

This Protocol addresses the problem of multiple segment drops. In effect, it can avoid many of multiple retransmit timeouts of TCP Reno. The New Reno modification introduces a partial acknowledgment strategy in Fast Recovery. A *partial acknowledgment* is defined as an ACK for new data, which does not acknowledge all segments that were in flight at point when Fast Recovery was initiated.

It is thus an indication that not all data sent before entering Fast Recovery has been received. In TCP Reno, a partial *ACK* causes exit from Fast Recovery. In the TCP New Reno it is an indication that (at least) one segment is missing and needs to be retransmitted.

This retransmission is effectuated and Fast Recovery continues. In this way, when multiple segments are lost from a window of data, TCP New Reno can recover without waiting for a retransmission timeout.

3.3 TCP SACK

TCP SACK was defined in RFC 2018 by Mathis et al. in 1996, and later extended in RFC2883 by Floyd et al. in 2000. TCP SACK further improves TCP performance by allowing the sender to retransmit packets based on the selective ACKs provided by the receiver. The implementation constitutes a SACK field that contains a number of SACK blocks. The first block reports the most recently received packets. The additional blocks repeat the most recently reported SACK blocks (ELAARAG, 2002).

The TCP SACK uses the basic congestion control algorithms and uses retransmit timeouts as a last option for recovery. The main difference is the way it handles the loss of multiple packets from the same window, in fast recovery. Like Reno, SACK enters fast recovery upon receiving duplicate ACKs. It then retransmits a packet and cuts its congestion window in half. In addition to that, SACK has a new variable called the *pip*e, and a data structure called the *scoreboar*d. The *pipe* is incremented when the sender sends a new or a retransmitted packet (ELAARAG, 2002), and is decremented when the receiver receives a new packet. This is indicated when the sender receives a duplicate *ACK* with a *SACK* option. The *scoreboard* stores *ACKs* from previous SACK options, allowing the sender to retransmit packets that are implied to be missing at the receiver. Like New-Reno, the sender exits fast recovery (ELAARAG, 2002).

3.4 TCP Santa Cruz

This is a new implementation of TCP that detects not only the initial stages of congestion in the network but also identifies the direction of congestion i.e., it determines whether congestion is developing in the forward or reverse path of the connection. TCP Santa Cruz is able to isolate the forward throughput from events such as congestion that may occur in the reverse path. Congestion is determined by calculating the relative delay that a packet experiences with respect to another as it traverses the network (PARSA; GARCIA-LUNA-ACEVES, 2000).

This relative delay is the foundation of their congestion control algorithm. The relative delay is used to estimate the number of packets residing in the bottleneck queue; the congestion control algorithm keeps the number of packets in the bottleneck queue at a minimum level by adjusting the TCP source's congestion window. The congestion window is reduced if the bottleneck queue length increases (in response to increasing congestion in the network) beyond a desired number of packets. The window

is increased when the source detects additional bandwidth available in the network (i.e., after a decrease in the bottleneck queue length) (PARSA; GARCIA-LUNA-ACEVES, 2000).

TCP Santa Cruz can be implemented as a TCP option by utilizing the extra 40 bytes available in the options field of the TCP header (PARSA; GARCIA-LUNA-ACEVES, 2000).

3.5 TCP Vegas

TCP Vegas was introduced in 1994 as an alternative to TCP Reno. It improves upon each of the three mechanisms of TCP Reno. The first enhancement is a more prudent way to grow the window size during the beginning of slow-start and leads to fewer losses. The second enhancement is an improved retransmission mechanism where time-out is checked on receiving the first duplicate acknowledgment, rather than waiting for the third duplicate acknowledgment (as Reno would), and leads to a more timely detection of loss (LOW; PETERSON; WANG, 2002).

The third enhancement is a new congestion avoidance mechanism that corrects the oscillatory behavior of Reno. In contrast to the Reno algorithm, which induces congestion to learn the available network capacity, a Vegas source anticipates the onset of congestion by monitoring the difference between the rate it is expecting to see and the rate it is actually realizing. Vegas' strategy is to adjust the source's sending rate (window size) in an attempt to keep a small number of packets buffered in the routers along the path (LOW; PETERSON; WANG, 2002].

Although experimental results presented by Brakmo and Peterson in 1995, and duplicated in Ahn et al. in 1995, showed that TCP Vegas achieves better throughput and fewer losses than TCP Reno under many scenarios. At least two concerns remained whether TCP Vegas is stable, and if so, whether it stabilizes to a fair distribution of resources; and whether Vegas results in persistent congestion. In short, TCP Vegas has lacked a theoretical explanation of why it works (LOW; PETERSON; WANG, 2002).

An advantage of TCP Vegas is that achieve higher throughput with less retransmission, nevertheless fairness problems has been reported: (1) when a TCP Vegas connection shares a link with TCP Reno connection, The TCP Reno connection uses most of the buffer space and the TCP Vegas backs off, interpreting this as a signal of network congestion. (2) There is Unfairness between TCP Vegas Connections. The connections that start up later could observe a larger RTT, causing the congestion window to be lower (SANADIDI, 2002).

3.6 TCP Westwood

TCP Westwood is a sender-side-only modification to TCP Reno that is intended to better handle bandwidth-delay product paths, with potential packet loss due to transmission or others errors, and with dynamic load. TCP Westwood relies on mining the ACK stream for information to help it better set the congestion control parameters - Congestion Window and Slow Start Threshold (UCLA, 2004).

TCP Westwood Modules for ns-2 come in two flavors: TCP Westwood based on TCP Reno; and TCP WestwoodNR that is a TCP Westwood based in TCP New Reno. We use in this the dissertation the first one with its default filter type, set to 3 (filter type_ 3) for more details see (UCLA, 2004).

As explained by their authors (GERLA et al., 2001), in TCP Westwood the sender

continuously computes the Connection Bandwidth Estimate (BWE) that is defined as the share of bottleneck bandwidth used by the connection. Thus, BWE is equal to the rate at which data is delivered to the TCP receiver. The estimate is based on the rate at which ACKs are received and on their payload. After a packet loss indication, (i.e., reception of 3 duplicate ACKs, or timeout expiration, the sender resets the congestion window and the slow start threshold based on BWE. More precisely, *cwin = BWE x RTTmin*.

Finally, we may say that the experimental results of TCP Westwood shows that it can handle losses caused by link errors more efficiently than TCP Reno and converges to "Fair Share" at steady state under uniform path conditions. For a deeper understanding of the fuctionality of TCP Westwood, you may look at (GERLA et al., 2001).

3.7 TCP Veno

TCP Veno is a sender-side protocol, without requiring any changes of the receiver stack or intervention of the intermediate network nodes. It can be deployed in server applications over the current Internet, coexisting with TCP Reno, the standard (FU; LIEW, 2003).

TCP Veno makes use of mechanism similar to that of TCP Vegas to estimate the state of the connection; however, such a scheme is used to deduce what kind of packet loss -- Congestion loss or random loss – is most likely to have occurred, rather than to pursue preventing packets loss as in TCP Vegas. If a packet loss is detected while the connection is in the congestive state, TCP Veno assumes the lost is due to congestion; otherwise, it assumes the loss is random (FU; LIEW, 2003).

The results of the experiments (FU; LIEW, 2003), show that TCP Veno can obtain significant throughput improvements without adversely affecting other concurrent TCP connections, as well as with TCP Reno Connections. Also was observed that in wireless networks with 1% of random packet loss rate, throughput improves up to 80% in comparison with TCP Reno.

4 TCP HolyWood

"Dear children! With great joy, also today, I carry my Son Jesus in my arms to you; He blesses you and calls you to peace. Pray, little children, and be courageous witnesses of Good News in every situation. Only in this way, will God bless you and give you everything you ask of Him in faith. I am with you as long as the Almighty permits me. I intercede for each of you with great love. Thank you for having responded to my call." *December 25, 2004.*
Bosnia-Herzegovina, Medjugorje.

Blessed Virgin Mary

As stated by Stallings (2004, P239), to get a good performance for end-to-end systems in a network environment, the design and implementation of the Transport protocol is a vital ingredient. With this idea in mind, we started to make different trials on TCP Reno with the unique purpose of finding a better throughput gain and a less variable jitter. Other important caracteristic of TCP HolyWood is that it inherited all the dynamic characteristics of TCP Reno. It is a sender-side protocol. So there is no need to reconfigure intermediate router or the receiver in a communication to start to work, even though it is only a simulated model the results as we will see in chapter 6 are promising. In addition, we may consider our proposal as a fine-tuning of TCP Reno.

TCP HolyWood will be desirable for the following standpoints: *usability:* due to that the modification we made in our proposal are small, simple and easy to set, an only one side requires change for the two ends of communication terminal. *Interaction, a*s we observed in the result section, our proposal is as fair as TCP Reno and may work perfectly and harmoniously together with the standard. *Competence,* our proposal showed a good performance in term of throughput and jitter against, TCP Reno, Westwood and Vegas. Certainly this will compell us to implement it in a real operative system and make future tests to sclutinize TCP HolyWood in other network environments as wireless, and heterogeneous networks.

4.1 The Model

Our model is derived from TCP Reno, we made some calibrations in order to achieve higher throughput performance but without the desire to change the essence of the standard TCP, our modifications were as follows:

1. We modified the slow start algorithm of TCP Reno from *CWND = CWND + 1* to *CWND= CWND + 9/5*, to have a rapid start. We used 9/5 or 1.8; nevertheless, it is not a magical number but fruit of several unsuccessful trials until we arrived to Table 4.1. In the Table 4.1, for example, assuming a slow start threshold (*SSTHRESH*) of 65536 bytes, our proposal was faster than slow start. TCP Reno required 16 steps instead TCP HolyWood that needed only 11 steps to get to an approximately value meaning that our proposal is more or less 1/3 faster than the slow start of TCP Reno.

   Table 4.1: Slow start in TCP Reno and TCP HolyWood

Steps	**TCP Reno** *CWND = CWND + 1* Bytes	**TCP HolyWood** *CWND=CWND+9/5* Bytes	**TCP HolyWood** $\lceil CWND=CWND+9/5 \rceil$ Bytes
1	2	2.0	2
2	4	5.6	6
3	8	16.8	17
4	16	47.6	48
5	32	134.4	134
6	64	375.2	375
7	128	1050.0	1050
8	256	2940.0	2940
9	512	8232.0	8232
10	1024	23049.6	23050
11	2048	64540.0	64540
12	4096	-	-
13	8192	-	-
14	16384	-	-
15	32768	-	-
16	65536	-	-

   We consider this a moderate values but we will really know only after the experimental simulations. We may also notice that the Table 4.1 in step 1, we started it with a Congestion Window of 2 for both TCPs, it is due to RFC 2581(ALLMAN; PAXSON; STEVENS, 1999) and the network simulator we deployed here (ns-1b8a) also consider this value as default parameter.

2. Because we make a change that made faster the slow start of TCP Reno we compensate it in Congestion Avoidance phase diminishing it from:

   CWND = CWND + 1/CWND to,

   CWND = CWND + 1/ (4.CWND)

   The idea behind this was to have approximately a constant function with a vertical slope and a plane top. In other words that our throughput curve resembles a constant line with value of the bottleneck bandwidth.

3. After three duplicate ACK, TCP Reno reduces the *SSTHRESH* to a half (50%) and the Congestion window to the half plus three (+3). TCP HolyWood instead reduce the Slow start threshold to five sixth (83%) and the congestion window to

[1] $\lceil x \rceil$ indicates maximum integer of x

five sixth plus three (+3), decreasing the window by a factor of 1/6. We took the idea from TCP Veno (FU; LIEW, 2003) that reduce them four fifth (80%), we used a slightly higher value with the intuition to win in the performance competition arena for one head, nevertheless based in background provided by TCP Veno.

4. When timeout occurs due to Congestion or higher Error Rate TCP Reno reduce the Slow Start Threshold to the half (50%) and set the Congestion Window to 1; TCP HolyWood instead Reduce the Slow start Threshold to Thirteen Twentieth (65%) and set the Congestion Window to 3. We decided to use an increase of 15% more than the TCP Reno because after a timeout occurs we are unaware if there is still congestion or not, we may intuit that there is equal probability. So why do not give a bias to the possibility, that there is no more congestion. After all, if the congestion persists, other timeout will occur and the time of this timeout will double as explained in section 2.4. We decided to set the Congestion Window to 3 instead of 1 after the timeout finishes and when a new Slow Start begins based on the RFC 2414 (ALLMAN; FLOYD; PARTRIDGE, 1998) briefly explained in section 2.3.3.

We may summarize the previous ideas of TCP HolyWood as follows:

a) When new data is acknowledged, and if $CWND \leq SSTHRESH$, set CWND = CWND + 9/5, *(Rapid Start).* Else set $CWND = CWND + 1 / (4.CWND)$ *(New Congestion Avoidance Phase)*

b) A Packet loss can be detected either by the reception of duplicate acknowledgments, i.e., four ACK with the same sequence number, or via timer expiry (timeout), as was explained in the section 2.4. Upon timer expiry, the algorithm goes into slow start phase as before:

   $SSTHRESH = (13 / 20).CWND$, and $CWND = 3$, it means timeout

   Otherwise, fast retransmission and fast recovery are taken up.

c) When triple duplicate ACKs are received in a row TCP Reno performs a retransmission of what appears to be the missing packet, without waiting for a retransmission timer to expire, and set.

   $SSTHRESH = (5/6).CWND$, $CWND = SSTHRESH + 3$

d) For every other duplicate ACK received, set

   $CWND = CWND + 5/6$. Transmit a packet if allowed by

   CWND (Fast Retransmit Phase)

e) When the next *ACK* arrives that acknowledges new data, set $CWND = SSTHRESH$, and a new cycle starts - Fast Recovery Phase. To clarify this ideas observe Figure 4.1.

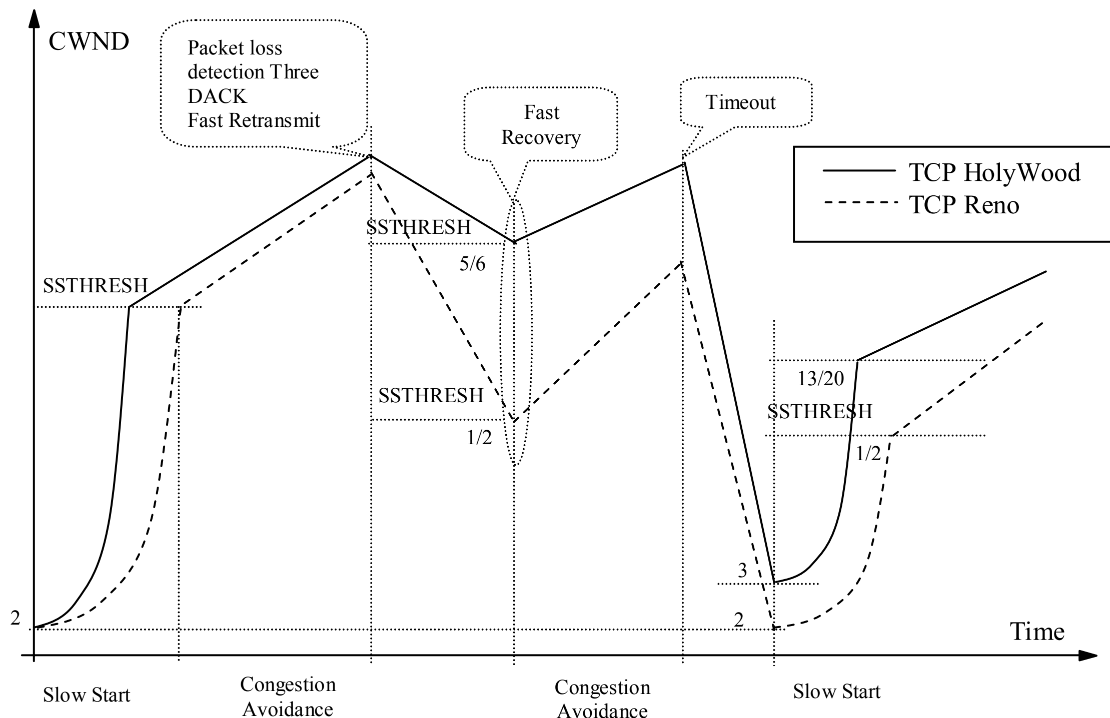

Figure 4.1: TCP HolyWood

4.2 The TCP HolyWood Code in NS-2

The code of TCP-holywood.cc is based on the code of TCP-reno.cc of the network simulator ns-2.1b8a. We created a HolyWood TCP agent in the network simulator as well as an independent c++ header function called TCP-holywood.h. The main functions that TCP HolyWood code is made up are:

a) timeout(int tno);
b) dupack_action();
c) opencwnd();
d) slowdown(int how);
e) process_qoption_after_send ();
f) rtt_counting();

We will give a brief description of the TCP Holywood code with the main modifications we made.

In function *dupack_action(),* we changed the method from: *slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_HALF),* that produced 50% in the values of CWND and SSTHRESH to: *slowdown(CLOSE_SSTHRESH_FIVE_SIXTH|CLOSE_CWND_FIVE_SIXTH)* that produced 83.33% in the Values of CWND and SSTHRESH

In the function *timeout (int tno),* we changed the CWND=1 to CWND=3, using the function slowdown from: *slowdown(CLOSE_CWND_ONE)* to *slowdown(CLOSE_CWND_THREE)*.

the function *timeout(int tno)* calls other function named "slowdown" that let us change the SSTHRESH from 50 % to 65% in this way, from:

slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_RESTART) to:
slowdown(CLOSE_SSTHRESH_THIRTEEN_TWENTIETHCLOSE_CWND_THREE)

In the function *opencwnd(),* we made the changes of the slow start (cwnd_ += 1) to rapid start (cwnd_ += 9/5) and the congestion avoidance

from: cwnd_ += increase_num_ / cwnd_

to: cwnd_ += increase_num_ /(4 * cwnd_)

The necessary changes for obtaining the SSTHRESH and CWND values as required for proper work of TCP HolyWood are in the function *slowdown(int how)*

The function *process_qoption_after_send()* checks if the sender has been idle or application-limited for more than an RTO, and if so, reduce the congestion window.

Finally, the function *rtt_counting()* checks if the sender has been idle or application-limited for more than an RTO, and if so, reduce the congestion window, for a TCP sender that "counts RTTs" by estimating the number of RTTs that fit into a single clock tick.

We intentionally clean the properties of the functions: *process_qoption_after_send()* and *rtt_counting(),* because when the sender transmits again or after the idleness of the sender finishes, we do not want a reduced congestion window, so the performance will not be diminished.

For a better visualization of the pictures of this dissertation, in colors, as well as the TCP HolyWood ns-2.1b8a code, you may find them available at the following sites:

http://www.inf.ufrgs.br/~oscar/TCP_HolyWood/

http://www.geocities.com/oscar_n_mori/TCP_HolyWood/

http://oscarnmori.freesuperhost.com/TCP_HolyWood/

4.3 Window Behavior

In the Figure 4.2 when the error rate is 0%, we may see that the congestion window (cwnd) and the slow start threshold (ssthresh) of TCP HolyWood are bigger than the TCP Reno. We may also observe that after 4 seconds the curves start to stabilize and the congestion window of TCP HolyWood present fewer variations for the same environment of the last Figure. Nevertheless, changing the error rate from 0% to 0.01% and 1% respectively and discounting the first 4 seconds due to transient state of the simulation we see that in Figure 4.3 the values of TCP HolyWood keeps higher than those of the TCP Reno for both congestion window and slow start threshold. Depending on the increase of the error rate the variation in the Congestion windows and slow start threshold increase it is because both TCPs are in full action preventing congestion.

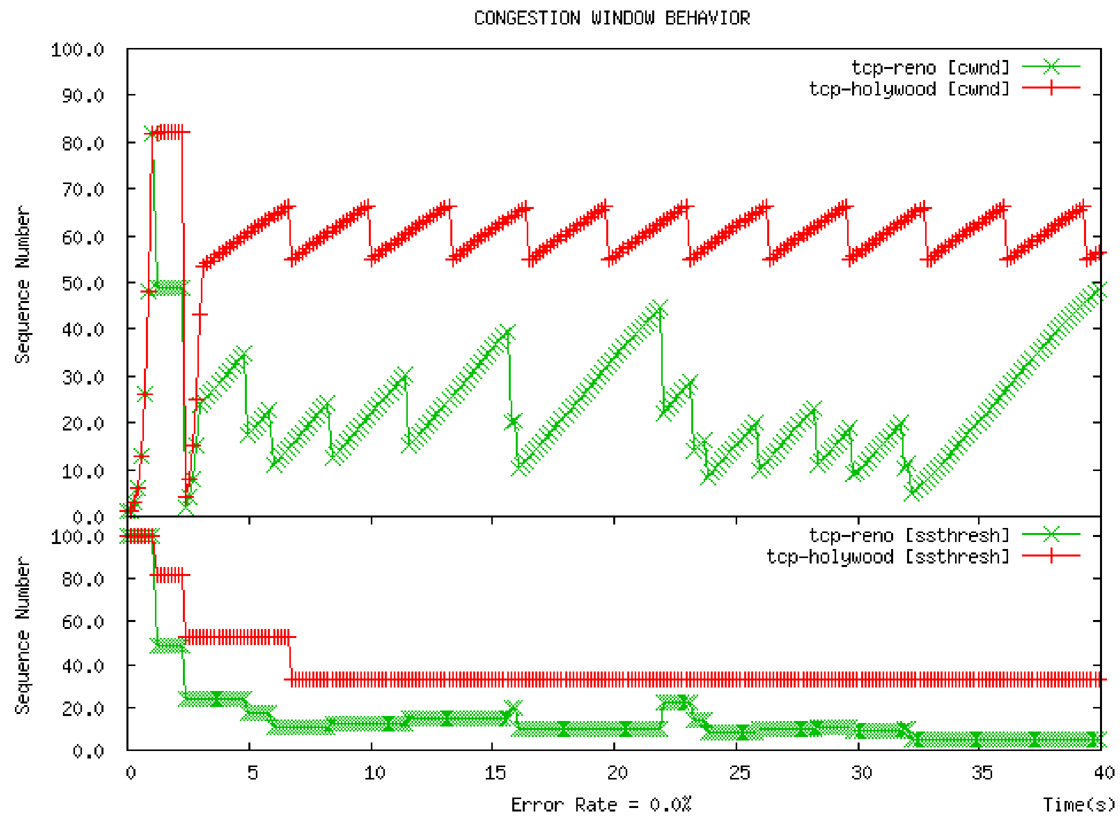

Figure 4.2: Congestion window behavior

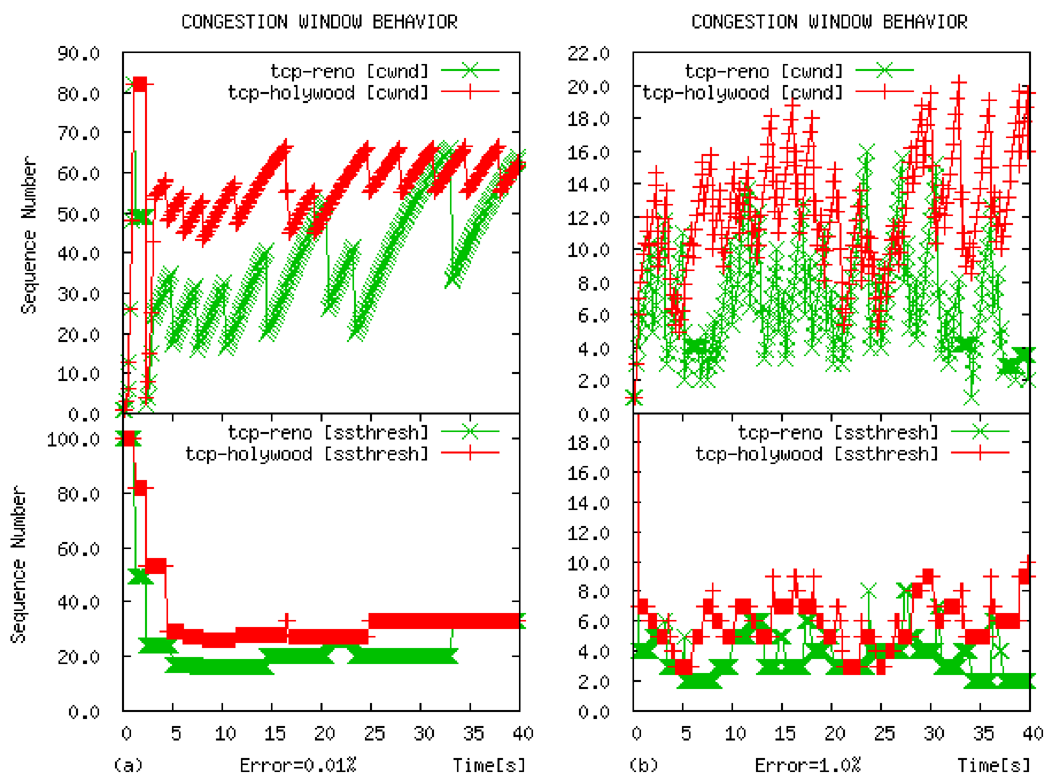

Figure 4.3: Window Congestion Behavior with different Error Rates

4.4 Performance Metrics

In order to measure the performance of our proposal with the Standard TCP and other protocols in a Network Environment, we need to define Performance Metrics. It is here that came to our help Raj Jain through his famous book "The Art of Computer Systems Analysis" (JAIN 1991), as stated by him, Considering the problem of comparing two different congestion controls for packets sent in order, it is required the following metrics:

1) Response Time: the delay inside the network for individual packets (Packet Delay)
2) Throughput
3) Processor time per packet on the source end system
4) Processor time per packet on the destination end systems
5) Processor time per packet on the intermediate systems
6) The variance of the response time or Jitter
7) The probability of out-of-order arrivals
8) The Probability of duplicated packets
9) The Probability of lost packets
10) The probability of disconnections
11) The Fairness

We used some of these detailed metrics for the following reasons as stated by Jain (1991, p.35-37):

- The packet delay (1) and the throughput (2) are redundant metrics; Jain also suggested that to use the power, that is a ratio of Throughput divided by Packet Delay, nevertheless in scientific papers is common use throughput (2), and it was our choice.
- The probability of duplicated packets (8) and the probability of the disconnection (10) is redundant with the jitter (6).
- Even though the processor time per packet on the source end system (3), On the destination end system (4), and on the intermediate system (5), are important metrics, in the case we have slower processors certainly the performance of network will diminish. Nevertheless, if they are faster ones as nowadays especially in high-speed networks, the processor time per packet could be rejected, moreover this metrics are vital to measure Computer systems.
- Regrettably, we could not use "The probability of out-of-order arrivals metric due to that the post processing package, Trace graph, deployed in this dissertation, in his last version, does not implement it**.**

For the reasons explained above and remembering the words of Jain (1991) that stated that "*The right metric to measure the performance of an analyst is not the number of analyzes performed but the number of analyzes that helped the decision makers*", in that sense, we used the following metrics: throughput, jitter, fairness and friendliness.

Throughput we consider as the most significant metric. It is defined as the rate at which a system can process a given computation (LILJA, 2000). For Networks, the throughput is measured in packets per second (pps) or bits per second (bit/s)

The Jitter, or variance of the response time, is the variability of the packet delay; other way of say the same thing is that, the jitter is the successive subtraction of packet delays.

Fairness, as stated by Jain (1991), is defined as a function of variability of throughputs per TCP connections. For any given set of TCP connections, with throughputs *x1, x2,..., x_n,* the following function can be used to assign a fairness index to the set.

$$f(x_1,x_2,...,x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n\sum_{i=1}^{n} x_i^2}$$

In other words Fairness is how equitable is resource sharing among a set of connections that use the same TCP Version (SANADIDI, 2002).

Friendliness is a variation of fairness and it tests the effects that the introduction of a new protocol has on the other protocols. The idea is that the proposed protocol should not hurt the standard one, and is of crucial importance to choose the varied parameter that makes the most impact in our network environment.

4.5 Factors

The factors are parameters that are varied in the script of our chosen ns-2.1b8a simulator and its values are called *levels*, we alter the factors to monitor the perturbation in the behavior of the TCP protocols under test. Factors now to be mentioned are:

a. Error rate
b. Propagation time
c. Bottleneck Bandwidth

Error rate can be a metric of accuracy (JAIN, 1991), but we use it here as a factor as well. It is a rate of random error affecting the network link due to external conditions as for example electromagnetic interferences that affect copper cables, as well as wireless links. It is used a normal distribution function to get random values.

Propagation Time is the time that a packets last to go from a source end system to a destination end system. It is proportional to the distance between the source and destination

Bandwidth is the maximum amount of information per unit of time (bit/s) that can be transmitted along a link. If there are several nodes connected in a chain topology, and each link with different bandwidth values, the link with smaller bandwidth will be the Bottleneck Bandwidth.

5 SIMULATION

"Since I left Birtwick I had never been so happy as with my dear master, Jerry; but three years of cab work, even under the best conditions, will tell one's strength, and I felt I was not the horse that I had been."

Anna Sewell, *Black Beauty*

Due to our limited research time and resources, we choose a medium level of accuracy, among the evaluation performance techniques that are analytical model, measuring a real system and simulation, the latter was selected, using in this dissertation the Network Simulator 2, (ns-2), version ns-2.1b8a. We deployed in this work a laptop computer Toshiba; model Satellite A20-S207. It has an INTEL Pentium IV processor with 2.66 GHz, with a hard disk of approx. 40GB.

We formated the hard disk and created two partitions just of the same size, and a swap memory 944MB. After that, we made a full installation of the Operative FreeBSD 4.10 (University of California in Berkeley, 2005). We installed also the netwok simulator, ns-2.1b8a (ns-2, 2004) and a post-processing tool called Trace Graph (MALEK, 2003). Then, we installed a word processor called OpenOffice 1.1.3 (OPENOFFICE, 2004), we also installed a Plotting tool called Gnuplot 4 (WILLIAMS; KELLEY, 2004) and finally after a proper configuration we install xcdroast (NIEDERREITER, 2004). Xcdroast is the software that let us burn Compact disks (CDs) as multi sessions in order to record our advances. With all these ready, we started our journey through the world of the discrete event simulation.

5.1 Verification

It is worthless if we do not trust in our simulations, and in order to obtain credibility, the following four figures show the verification of our simulator. Figure 5.1 (Variation of *cwnd* and *sstresh* for TCP Westwood using the script *test-1-simple.tcl) and* figure 5.3 (TCP Westwood Bandwidth Estimation using script *test-1-simple.tcl) are* exhibited in (UCLA, 2004). After download the TCP Westwood code from (UCLA, 2004) and installed it together with the ns-2 simulator, we required to verify the proper functioning of it. We proved if the verification was correct, generating two figures (5.2 and 5.3) in our equipment and comparing them with their equivalents in (UCLA, 2004). Then we may discern that figure 5.1 and figure 5.2 are practically similar, the same thing we affirm for figures 5.3 and figure 5.4. For this evidence, we claim that the simulator

version ns-2.b18a and TCP Westwood deployed in our equipment are verified. With the intention to verify also the ns-2 TCP HolyWood code, we used the same script (*test-1-simple.tcl*) that uses TCP Westwood in its web page. It helped us also to verify, the modifications we made with other script but always based in *test-1-simple.tcl* script.

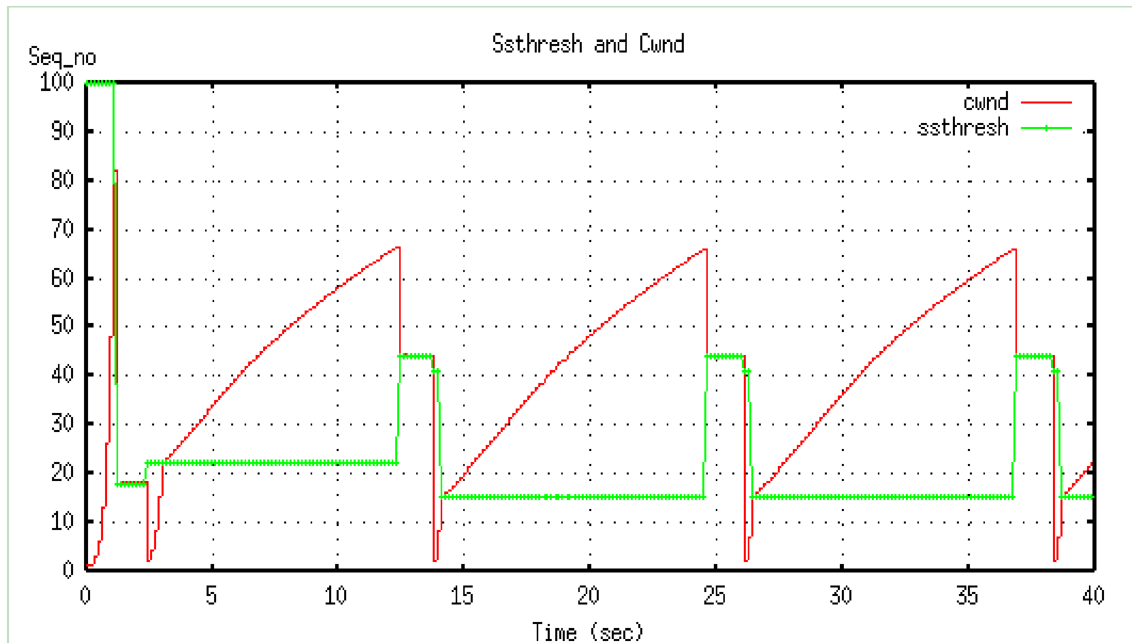

Figure 5.1: Variation of *cwnd* and *sstresh* for TCP Westwood using the script *test-1-simple.tcl (*UCLA, 2004)

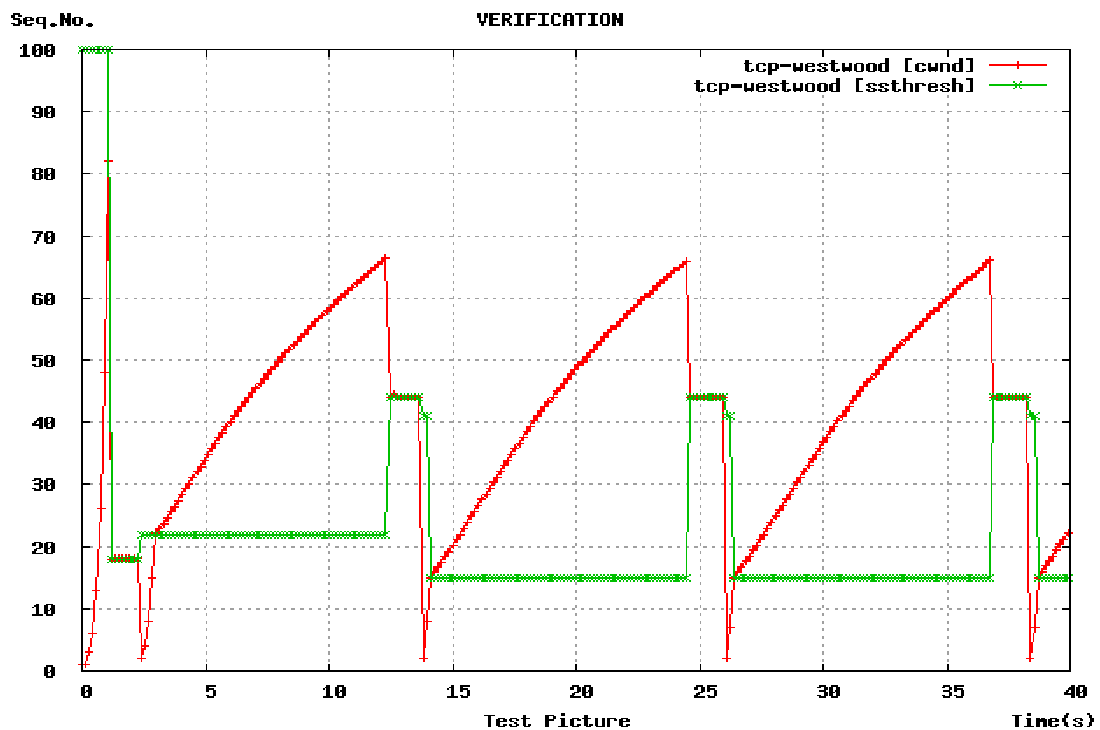

Figure 5.2: Variation of *cwnd* and *sstresh* for TCP Westood using the script *test-1-simple.tcl*-Verification

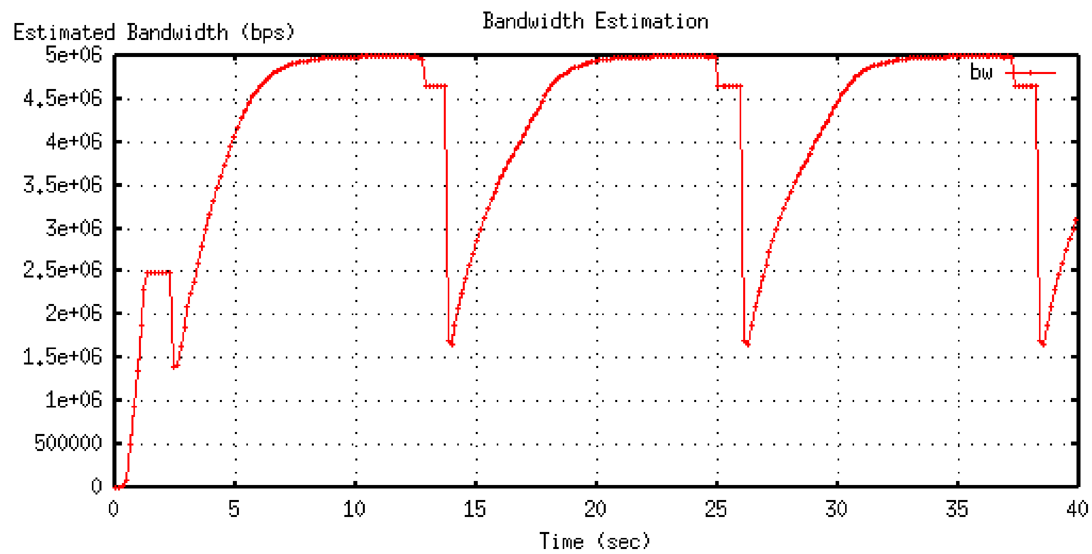

Figure 5.3: TCP Westwood Bandwidth Estimation using script *test-1-simple.tcl* (UCLA,, 2004)

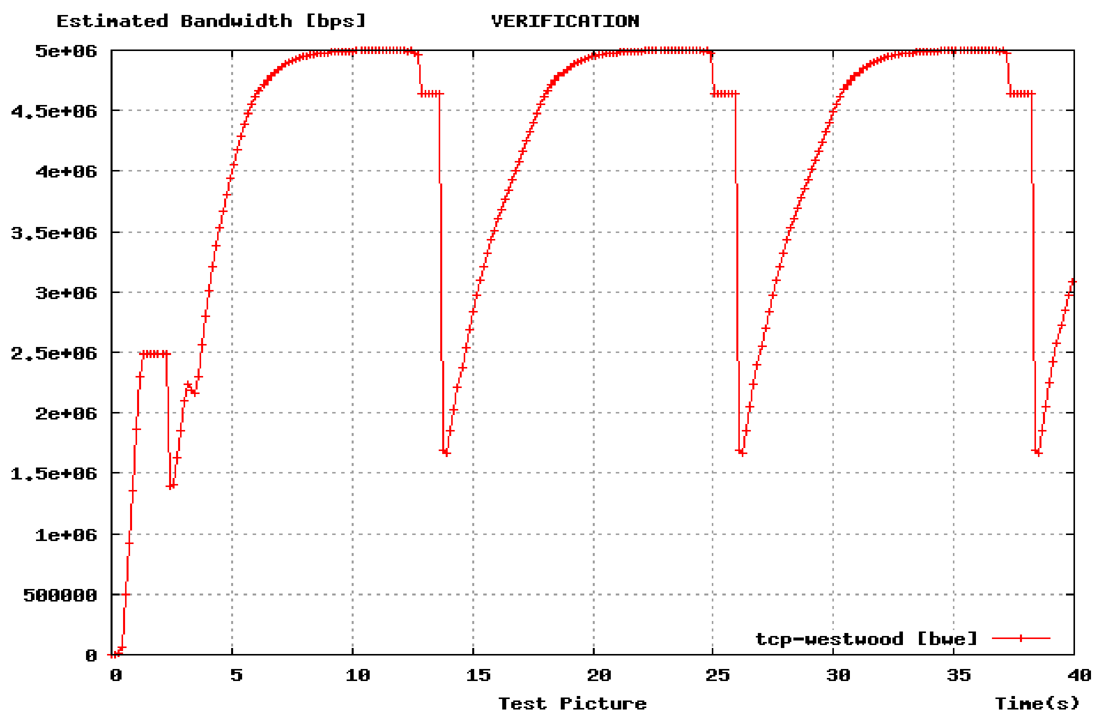

Figure 5.4: TCP Westwood Bandwidth Estimation using script *test-1-simple.tcl-* Verification

5.2 Simulation Methodology

About the installation, a complete systematic guide of how to install ns-2 Version ns-2.1b8a in our operative system FreeBSD 4.10 is in the appendix D.1. We used, in all the experiments the default values of ns-21b8a found in the file "ns-default.tcl"; the path to get there is *"/usr/local/ns-allinone-2.1b8a/ns-2.1b8a/tcl/lib"* in the supposition we install it in the directory *"/usr/local"* from FreeBSD Operative System.

All the simulations we made where of 150 seconds, but the question arise. Why that value?

The answer is simple, we wanted to diminish the warm up time or transient time before the system is stabilized. Nevertheless, another question arise, is enough 150 seconds?

Well this time obtain the answer was not so simple we tested several time values until we get approximately to 700 Sec. as a highest time value due to the post processing tool, Trace Graph (MALEK, 2003), we used for our specific simulation (that is the same of test-1-simple.tcl but with 700 Sec.) did not worked properly for more time. We compared the result of 150 Sec. with 700 Sec. with four TCP protocols and we get the Table 5.1.

Table 5.1: Comparison between simulations of 150s, and 700s.

TCP Protocol	Time [sec]	Average troughput [bit/s]	COV [%]	C1 [95%] [bit/s]	C1 [95%] [bit/s]	Ratio 150s/700s [%]	Relative change
HolyWood	150	4896404.8	12.95	4794968.3	4997841.2	98.367	-1.633
	700	4977689.5	5.96	4956722.0	4999657.0		
Westwood	150	3895673.6	40.24	3644827.2	4146520.1	98.272	-1.728
	700	3964193.3	38.07	3852398.3	4075988.4		
Reno	150	4865326.6	13.64	4759143.1	4971510.1	98.132	-1.868
	700	4957941.7	6.31	4934760.6	4981122.8		
Vegas	150	4896106.0	12.33	4799527.6	4992684.3	98.362	-1.638
	700	4977625.1	5.68	4956685.7	4998564.5		

After analyzing the Table 5.1 we conclude that adding to our 150 seconds simulation 550 seconds to get to 700 seconds we have a relative change of less than 1.9%. Therefore, we will work with this small error because 1.9% is not representative and we may say that the results of our 150s experiments are approximately equal to 700 seconds experiment with and Error in the Average of less than 1.9% with relation to 700-second-experiments

About the script *test-1-simple.tcl*, which we used as reference, it deploys the default FTP protocol of the Network Simulator 2, version ns-2.1b8a, with a normal error rate distribution, a fixed packet size of 1400, a bottleneck bandwidth of 5 Mbit/s, a propagation time of 35 ms, a simulation time of 40 Sec. The script before mentioned, was used to test TCP HolyWood in the Topology 1, in section 5.3.2, and the script also plot automatically the figures, Congestion window versus Time, Slow Start threshold versus Time and the Estimation Bandwidth versus Time used in the Verification, section 5.1. A copy of this script may be found in the Appendix B.3

5.3 Simulated Network Topologies

We have used in our simulations two network scenarios, called *network topology 1* and *network topology 2*. The latter is an evolution of the former. Figure 5.5 presents the first topology, where n_0 and n_3 are the sender and receiver respectively and n_1 and n_2 are node routers. Between these routers is the bottleneck link.

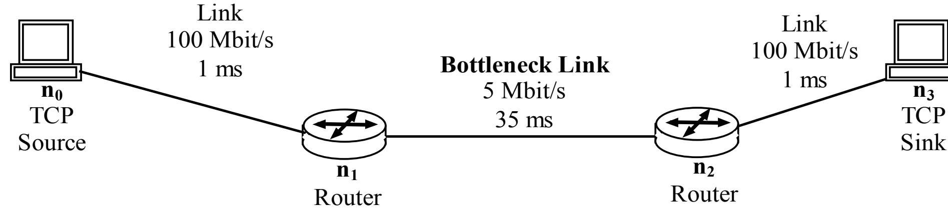

Figure 5.5: Simulated Network Topology 1

In Network Topology 2, Figure 5.6, we add 10 node transmitters ($n_{101}, n_{102}, ..., n_{110}$) and 10 node receivers ($n_{201}, n_{202}, ..., n_{210}$) each one attached with its respective TCP agent in the sender and a TCP-sink in the receiver. The nodes n_o, n_1, n_2, n_3 became node routers and the bottleneck link is the same as in the topology 1.

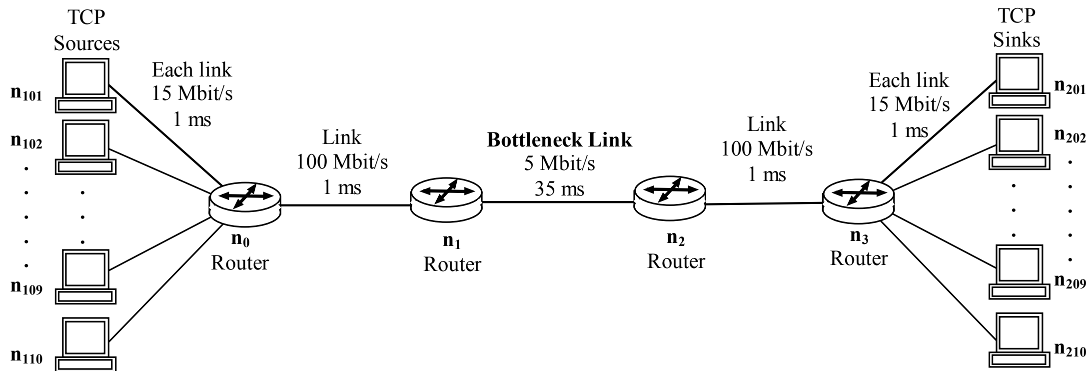

Figure 5.6: Simulated Network topology 2

5.4 Discrete-Event Simulators

A simulator using a discrete-state model of the system is called a discrete event simulator. Observe that the term "discrete" does not apply to the time values used in the simulation. A discrete-event simulator may use a discrete or continues time values. All the discrete-event simulators have a common structure. No matter of the system under modeling the simulator will have some of the following components: Event Scheduler, Simulation clock and a Time advancing Mechanism, System State variables, Event Routings, Input routings, Report Generator, Initialization Routings, Trace Routings, Dynamic Memory Management, an the Main Program, that brings all the routing together (JAIN, 1991, p.406-408) more information may be found in (LUL 2000, p185).

In this regard, we start the chase of a proper simulator for our proposal. First, we think to use the well-known simulator ns-2, but being considering ns-2 by the Computer

science Department of the University of Boston (2004), as probably the most user-unfriendly software in this world, as well as our own little experience we considered the same, so we tried to find other respecTable simulators to help to model our proposal

5.4.1 GloMoSim

Glomosim (UCLA, 2003) is a scalable simulation library that works with a discrete event simulator parsec (PARallel Simulation Environment for Complex Systems). It works with wired and wireless environments, being created especially for the latter; it has just a single TCP (TCP Tahoe) implementation. Moreover, to work with it meant for us to make an extra work creating TCP Reno, and also it do not have other flavors of TCP to compare.

5.4.2 NCTUns 2.0

A Discrete Event simulator as NCTUns 1.0 as stated by Wang (2003) was a good choice; nevertheless, currently during the development of this dissertation appeared NCTUns 2.0; because, it has an attractive graphical interface and post-processing data trace built-in. We were ready to work with it if not for a small detail, it uses a tunnel network interface, it means that it deploys existing real-world FreeBSD Protocol stack to provide high-fidelity TCP/IP network simulation results. Therefore, if we would like to use it, for our proposal, we had to recompile the real-world UNIX kernel code, and create a real implementation of TCP Holy Wood, goal that was out of our time and effort. Well after a good voyage through the world of other discrete event driven simulators we came back to the same point where we started, use the Network Simulator 2.

5.4.3 Network Simulator 2

Also called by its initials ns-2, is the standard Network Simulator deployed world wide, researchers of institutions as INRIA and Several Universities as University of Berkeley, even in this alma mater is used for modeling networks for Papers, dissertation and theses. We will not be the exception, nevertheless we deployed a past version of NS-2, the version ns-2.1b8a, because, that version was used by the creators of TCP Westwood, and aid us to verify our proposal.

The ns-2 uses two languages; C++ and Otcl (MIT's Object Tool Command Language). C++ is fast to run and is used for protocol implementations, as ours, Otcl runs much slower but can be changed quickly, it is ideal for simulation configuration, that is why we use scripts as test-1-simple.tcl where we test a myriad of possibilities. NS-2 provides glue (tclcl) to make object and variables appear on both languages.

As explaining by a tutorial we advise to use (ALTMAN; JIMENEZ, 2003), the ns-2 is a discrete event driven simulator, where the advance of time depends on the timing of events, which, are maintaining by a scheduler. An event is an object composed of: an unique identification, a scheduled time, and a pointer to an object that handle the event; and the scheduler maintains an ordered data structure with the events to be executed and discharge them one by one, invoking the handler of the event. Other source of consult besides the ns Manual (FALL; VARADHAN, 2004), that we found interesting is a Brazilian tutorial called: " Network Simulator: Guia Básico para Iniciantes" (COUTINHO, 2003), and (ROCHOL, et al. 2003) where it is found a comparison of the three simulators before mentioned, also in Portuguese.

5.5 Post-Processing Stage

We use two softwares for the post-processing stage: The Trace graph (MALEK,2003) package and the Awk program (GOEBEL, 2004) . The Trace Graph 202 package for linux is a powerful tool that works with a library of Matlab 6.1 (MATHWORKS, 2004). It converts your "tr" ns-2 trace into a "mat" format ready to use with Trace Graph. You may also get a graph in 2 and 3 dimensions; in addition you may get ready made Figures of throughput Versus Time, Jitter Vs time and others as in our case, in a relatively fast way, at least in the laptop we used.

Because Trace Graph was intended for Linux and not for FreeBSD, we use a Linux emulator that comes with FreeBSD to unpack the Matlab 6.1 libraries. Trace Graph is a didactic tool, nevertheless it lacks of enough documentation and the version we are using Trace Graph 2.02 for Linux is the last. We honestly hope that the author changes his mind and keeps on maintaining it, due to that ns-2 misses of a post-processing tool integrated to it. For details about Trace Graph may be found in (MALEK, 2003), and a systematic guide to install it on FreeBSD 4.10 (UC Berkeley, 2005) is the appendix D.2. Finally, we had some troubles for example the graphical interface of KDE 3.2 (2004) did not start after using Trace Graph. It should be that some libraries of Trace Graph with the libraries of Matlab 6.1 (MATHWORKS, 2004) are conflicting with KDE 3.2 (2004). That is why we start to use the light graphical interface WINDOW MAKER (KOJIMA, 2004). Other post-processing program we used is AWK. AWK is a text-processing language that comes with the default FreeBSD 4.10, distribution. The basic function of AWK is to search patterns in each line of a file we choose. When a line fit one of the patterns, AWK executes specified actions on that line, and it goes on until the last line of a chosen file (ROBBINS, 2003).

Common programs as BASIC, C or PASCAL are *procedural* ones, it means that you have to describe in detail every step the program is to take and it is clearly harder to depict the data your program will process. AWK instead is *data-driven*, that is, you describe the data you want to work with and then what to do when you find it. For example in a file with several columns you may choose any of them, compare with the others and make calculations that is what we did to get the Coefficient of Variation (COV) and the Confidence intervals. Examples of AWK programs you may find in the appendix B.6 B.7, B.8, and B9. If it is of interest of the reader, we may recommend using an on line tutorial called “A Guided Tour of Awk” by Goebel (2004) and of course, GAWK: Effective AWK Programming by Robbins (2003).

5.6 Limitations and Assumptions

In our experiments, we did not use a complete repetitions of our simulations with the exactly equal experimental configuration as that in a previous run of simulation, Using different Seeds simply, because the script used (*test-1-simple.tcl*, original script from TCP Westwood of University of California. Los Angeles,) to verify our simulation does not deploy it. Even though the replication of our simulations by one hand would let us to have a closer approach to real experimental environment in the other hand it would consume more time of our manpower resources and the traceability of our simulations for interested scientists would be injured.

Other limitation of our experiments as deployed by the script test-1-simple.tcl is that we have a fixed size of packets. Certainly using a large, medium and small size of packets, we would be closer to reality.

In our simple experimental design, each factor was changed one by one in order to monitor the perturbations between TCP protocols under test. In spite of that, it could

lead us to wrong conclusions if our chosen factors would interact such that the effect of one factor depends upon the value of other factors (JAIN, 1991) (LILJA, 2000), in other words, whether one factor is a function of the others.

In our case, guided by our intuition, and because we did not make further tests, we assume that factors used in our experiments as propagation time, bottleneck bandwidth and error rate are independent among them. However, it should be advisable for future experimental designs, if the case, that the factors would be dependent among them to use the full factorial experiment designs and fractional factorial designs (JAIN 1991) (LILJA, 2000). For better it would be our TCP model, it is a simulation and not reality, so all our results are approximations to get enough insight for a future implementation.

Because we could not suppress the transient time (or warm up time) due to a bug in the post processing Trace Graph package when working with intervals, to diminish the effects of the transient time we prolonged the simulation time from 40 seconds to 150 seconds. But the question arose if with that time the error introduced by the transient time would be sufficient diminished to discard it, that is why we decide to use the highest simulation time that our post processing package could tolerate. It was 700 seconds, for our experiments.

We assumed that 700 seconds were enough to suppress the unstable time before mention, and as explained in section 5.2 we used 150 seconds in all the tests.

Finally, in our experiments, we deployed the default values of the Network Simulator 2, version ns-2.1b8a.

6 RESULTS AND ANALYSIS

"How all this will terminate, I know not, but I had rather die than return shamefully, my purpose unfulfilled. Yet I fear such will be my fate; the men, unsupported by ideas of glory and honor, can never willingly continue to endure their present hardships."

Mary Wollstonecraft Shelley, *Frankenstein*

In this chapter, we present the methodology to get our results, a common terminology, and mainly in section 6.3 a performance comparison of throughput and jitter as well as Fairness of TCP HolyWood with TCP Reno, the standard. Finally, in section 6.4 we showed also a performance comparison of Throughput and Jitter, nevertheless of TCP HolyWood with other protocols --TCP Westwood and TCP Vegas. In addition, we used Error Rate, Propagation Time and Bottleneck Bandwidth as factors.

6.1 Methodology of the Analysis

The sections Impact of Error Rate, Propagation time, and bottleneck bandwidth on throughput and jitter we simple used the Trace Graph (MALEK, 2003) outputs ".trg" deploying Gnuplot (WILLIAMS; KELLEY, 2004) for a more appropriated presentations, we used several small figures compounding a bigger picture to the reader should grasp the idea in a single view. In the other graphs, we used the statistics generated from the Trace Graph with ".trg" extension too. When we used Trace graph we configure it as follows:

- In the Trace Graph 2.02 main window (MALEK, 2003), we set in ''Options'' the option "Count Packets ID only once'' to ON, for topology 1 and 2; and we changed "Current node" option to 3 (destination node) for topology 1 and for topology 2 was varied according how many receivers we had one at a time. The rest was the default Trace graph configuration.
- In the Trace Graph "Graphs" window we set always options "Save graphs", "Save graphs statistics" to ON, in order to generate the ".trg" files needed for Gnuplot.
- At last, in the Trace Graph "Network information" window, we set in "Options" the option "Save Information" ON and in "Network Informations we set the option "Simulation information" ON. There, we found the information of the

Sent packet and lost packets of each protocol.

The statistics we made were the Average or arithmetic mean, the standard deviation, Coefficient of Variation, and the Confidence intervals; all this concepts are well explained in the next section 6.2, Terminology. All the statistics we made from STAT files of Trace graph and were processed after, with the AWK programs (appendix B.6 B.7, B.8 and B.9) are in the appendix A.

We used Coefficient of Variation (COV) to compare the relative size of the variations among the TCPs protocols used here and Confidence intervals of the average values to quantify the precision of our simulated measurements. To calculate Percentage of COV is divide the Standard Deviation by the Average and multiplies the result for 100. We worked with confidence intervals of 95%; for further information read (LILJA, 2000, p.48-52), and t distribution values at page (LILJA 2000, p.250) or also read (JAIN, 1991, p.204-208) and t distribution values at (JAIN, 1991, p.630-631).

The Standard Deviation formula that we used is from (LILJA, 2000, p.38-39), and the idea to make the Awk program considering the standard deviation is from (ALTMAN; JIMENEZ, 2003, p.34).

Besides, in the simulations if we did not specify Error Rate, Propagation Time, etc; it meant, we used an Error Rate of 0.1%, Propagation time of 35ms, a simulation time of 150 seconds, a Bottleneck Bandwidth of 5Mbit/s and a buffer size calculated by the automatic pipe size setting of the script *test-1-simple.tcl*, usually 32. In all the Figures used here, we processed the output ".trg" from Trace Graph with Gnuplot version 4, an excellent tutorial is found in (KAWANO, 2004). In the appendix B.10 and B.11, the reader may find examples of the Gnuplot programs we deployed in this dissertation. Finally, for the analysis of the ratios of this dissertation, in the case of lower is better as it is with jitter we did as Jain (1991, p.171) to take our protocol TCP HolyWood as base. Furthermore, in the case of higher is better as it is with throughput we use TCP Reno, or TCP Westwood or TCP Vegas, according the case, as a base.

Note that all the test made in this chapter, were made in the reception side as reference to measure the metrics, specifically in topology 1, we used node n3 (TCP Sink) and in topology 2 we used, nodes n201, n202 … n210 (TCP Sinks)

6.2 Terminology

The terminology used in our analysis of performance can sometimes be confusing. Here we define some of the most important terms used.

Accuracy: The absolute difference between a measured value and corresponding reference value (LILJA, 2000).

Coefficient of Variation (COV): The ratio of the sample's standard deviation to the corresponding mean of the sample. (Standard Deviation / Mean). It provides a dimensionless value that compares the relative size of the variation in a set of measurements with the mean value of those measurements (LILJA, 2000).

Confidence Interval: It is not possible to get a perfect estimate of the population mean from any finite number of finite size samples. The best we can do is to obtain probabilistics bounds; these bonds are called Confidence interval (JAIN, 1991, p.204).

Confidence level: The probability that a confidence interval actually contains the real means (LILJA, 2000)

Factors: The input variables of an experiment that can be controlled or changed by the experimenter (LILJA, 2000).

Interaction: When the effect of one factor depends on the level of another factor (LILJA, 2000).

Levels: The specific values to which the factors of an experiment may be set (LILJA, 2000).

Outlier: a measured value that is significantly different than the other values in a set of measurements (LILJA, 2000).

Perturbation: The Changes in a system's behavior caused by measuring some aspects of its performance (LILJA, 2000).

Precision: The amount of scatter in a set of measurements. Corresponds to the repeatability of the measurements (LILJA, 2000).

Random Errors: Errors in measurements that are completely unpredictable, nondeterministic, and perhaps not controllable. They are unbiased in that a random error has an equal probability of either increasing or decreasing a measurement (LILJA, 2000).

Range: The difference between the largest and smallest values in a set of measurement (LILJA, 2000).

Replication: a complete repetition of an experiment performed with exactly the same experimental configuration as that in a previous run of experiment (LILJA, 2000).

Resolution: The smallest incremental change that can be detected and displayed by a measuring tool (LILJA, 2000).

Significance level: The probability tipically denoted by α, that a confidence interval does not contain the actual mean (LILJA, 2000).

Systematic Error: Errors in measurements that are the result of some experimental "mistake" such as a change in the experimental environment or an incorrect procedure that introduces a constant or slowly changing bias into the measurements

Trace: A time-ordered sequence of events (LILJA, 2000).

Validation: Determine how close the results of a simulation are to what would be produced by an actual system (LILJA, 2000).

Verification: To determine whether a simulation model is implemented correctly (LILJA, 2000).

6.3 TCP Holy Wood versus TCP Reno

We present here a set of results of two metrics throughput and jitter using three factors, error rate, propagation time, and bottleneck bandwidth to appreciate the perturbations and interactions of TCP HolyWood in network environment in comparison with the standard, TCP Reno. We also present two additional metrics fairness and friendliness.

We did not use replication in our experiments because even if we repeat several times the same simulation and considering a normal random error rate, and with the same configuration values, apparently there were no changes in the output of the

simulation using *test-1-simple.tcl* script.

The accuracy of our result is limited to what ns-2.1b8a may offer to us. Only after validating it in a real network and with a real implementation of our proposal, we will know how accurate our results will be. We zoomed figures of our results where they were appropriated, using logarithmic and linear scales to obtain through Gnuplot the best possible visual resolution. Finally, the round number was of 10 in Trace Graph by default in all its outputs but in order to fit the results into tables as presented in appendix A. we use integer arithmetic with round numbers of different decimal places, from 1, 2, 3, 4 until 5.

We try to suppress all the systematic errors while we were handling the results. However, if you realize any involuntary mistake we made, please we appreciate you report it to us. Moreover, it brings to our memory the words of the English writer, Oscar Wilde, in his novel the *The Picture of Dorian Gray, " ... that is one of the secrets of life. Nowadays most people die of a sort of creeping common sense, and discover when is too late that the only things one never regrets are one's mistakes."*. Well now with no more preambles let us start the analysis of the results.

6.3.1 Impact of Error Rate on Throughput

In this subsection, we analyze how the throughput was affected by a varying and increasing error rate. When we analyze Throughput, we hope to find a higher Value and as stable as possible. In the Figure 6.1, we presented the Throughput Versus Time with different Error Rates; a different TCP was used in Topology 1, one at a time. Besides, in all the simulated experiments of this subsection, we used Propagation Time of 35 ms and a Bottleneck bandwidth of 5 Mbit/s. We discounted the first 10% of the simulation time in our observations; it means 15 Sec. from 150 Sec., as a rule of thumb due to warm up time (COUTINHO, 2003) (MACDOUGALL, 1987).

In Figure 6.1.a, with an error rate of 0.1% we may notice that TCP HolyWood shows a more stable appearance than TCP Reno with three thin significant outliers that reduce the performance from the maximum available Bottleneck Bandwidth of 5 Mbit/s to approx. 3.5 Mbit/s, 3 Mbit/s and 3.7 Mbit/s respectively. Instead, TCP Reno presents several (eight) thick negative outliers that reduce its average performance, being the range of them between approx. from 4 Mbit/s to 1 Mbit/s. When there were not errors in figure 6.1.b both TCPs outperformed very stable at the maximum bottleneck bandwidth; nevertheless accordingly the error rate was increasing also the instability of the throughput increased; but in all the cases (Figure 6.1.c, 6.1.d, 6.1.e) TCP HolyWood exhibited a smoother throughput curve that TCP Reno. You may find in appendix C.1.1 the complete set of simulations we made about this subsection.

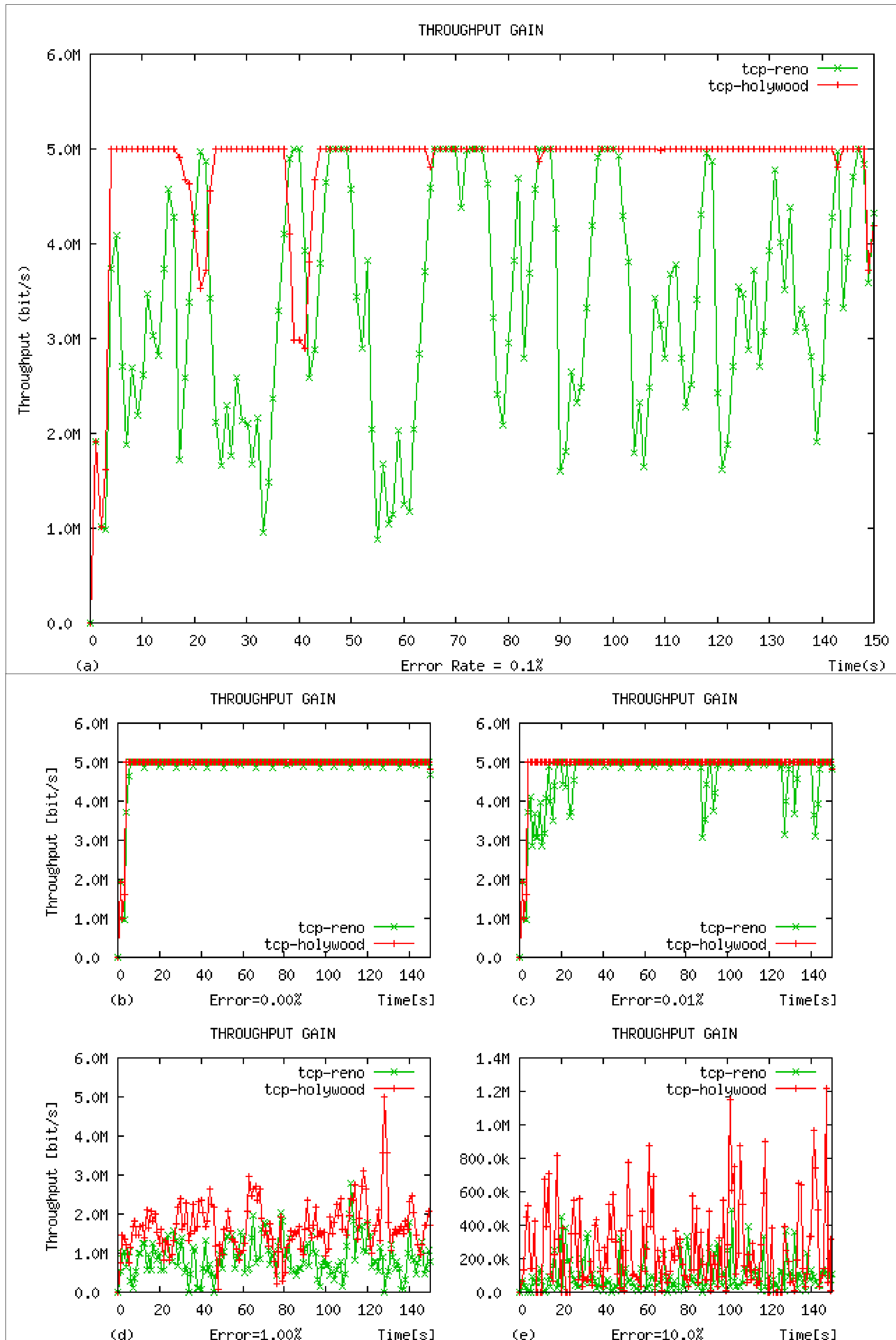

Figure 6.1: Throughput versus Time with Different Error Rates

Now we will observe if TCP HolyWood or TCP Reno presented better average throughput performance; at a glance, figure 6.2, shows that TCP HolyWood outperforms TCP Reno, with the exception of one point with an error rate of 8.0%. In addition, the Coefficient of variation (COV) of TCP HolyWood is smaller than the TCP Reno from Error Rate of 0% to 6%, and from 8.0 % to 60.0% the Coefficient of Variation of TCP HolyWood is bigger. The exact percentage of COV and Confidence intervals of both TCPs are found in appendix A, Table A.1.1.

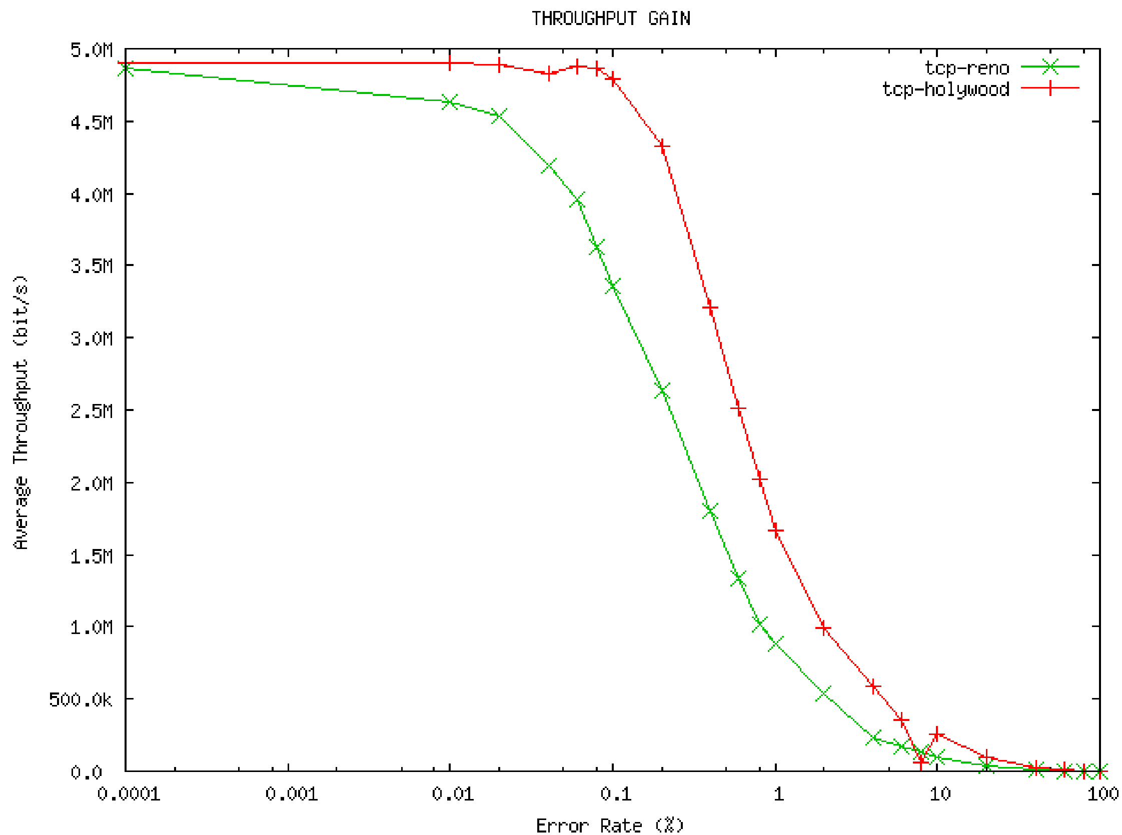

Figure 6.2 - Average Throughput versus Error Rate

Nevertheless, a question arise how much is the gain of throughput of our proposal with the TCP Standard. This answer we get in the next Figure 6.3 where we present a percentage of throughput ratios between TCP Holywood and TCP Reno. From an error rate of 0% to 0.01%, TCP HolyWood is slightly better than TCP Reno with a value of 3.22% with TCP Reno as a base. From an error rate of 0.01% to 1.0 %, TCP HolyWood throughput is 49.7% more than the standard. From an error rate of 1% to 6.0%, TCP HolyWood throughput is 111.26% more than the standard. In all the cases, we used TCP Reno as a base. In an error rate of 8.0%, TCP Reno wins with a throughput of 57.47%; and finally in the Error rate interval between 8% until 60% our proposal wins with approximately 155.81% with TCP Reno as a base.

Making a total average from Error rates of 0% to 60%, we get a gain in throughput of TCP HolyWood over TCP Reno of 73.46% with TCP Reno as a base. We calculate this general average using the Awk program in the appendix B.9.

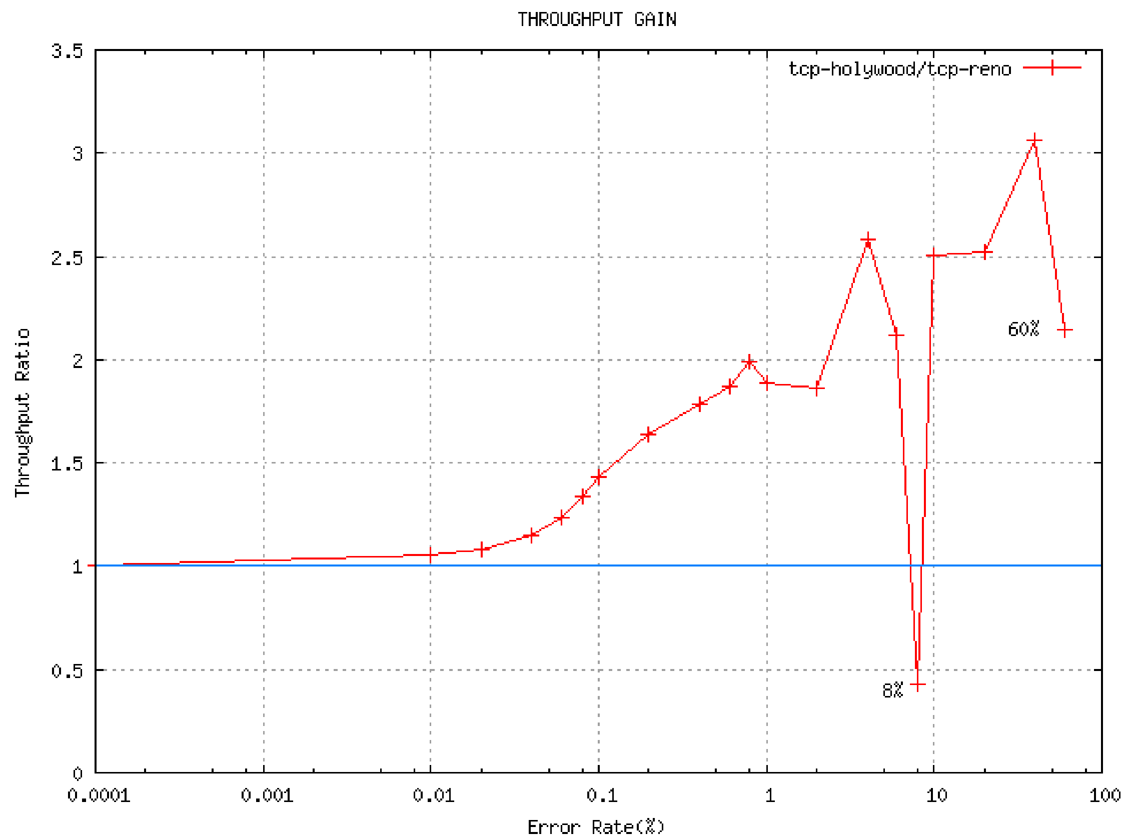

Figure 6.3 - Throughput Ratio versus Error Rate

6.3.2 Impact of Propagation Time on Throughput

We used an error rate of 0.1% and a bottleneck bandwidth of 5 Mbit/s in all the simulated experiments of this subsection. In Figure 6.4.a we worked with a propagation time of 50 ms. We also discount the first 15 seconds of the analysis of Figure 6.4.

In Figure 6.4, we do not consider as in the previous subsection, the first 15 seconds of simulation due to warm up time or better explained, 15 seconds onwards, a steady-state cyclic regime of TCP is attained and TCP is always in Congestion avoidance phase. However, before 15 seconds approximately, we see a transient behavior in which TCP is in the slow start phase, but in second 3.5 of the Figure 6.4.a we observe a timeout that affects both TCPs.

We observe in Figure 6.4.a that our proposal works better than TCP Reno, presenting just two thick negative outliers or falls in second 22 and 45 approximately of approximately of the half or 2.5 Mbit/s of the maximum bottleneck bandwidth of 5 Mbit/s. In general, we observe an oscillation of the TCP HolyWood curve in the range of 5 Mbit/s to 4 Mbit/s. TCP Reno instead presented different throughput behavior 6 thick negative outliers plus 2 thin negative outliers or falls and its throughput range oscillate between 5 Mbit/s to 0.5 Mbit/s.

When TCP HolyWood presented its two falls in the seconds approximately 22 and 45, TCP Reno also did but thicker and deeper. We may intuit that this situation was due to lost packets and the algorithms fast retransmit and fast recovery was at full work. In both TCPs the congestion avoidance phase is occurring. Nevertheless, because TCP HolyWood thresholds are bigger than TCP Reno the recovery of our proposal was faster.

In the same environment but working with propagation, time of 1 ms there was no difference of the performance of both TCPs. In 10 ms, for example, our TCP is slightly better, whereas with bigger propagation times of 100 ms and 1000 ms, the TCP HolyWood presented better throughout performance, meanwhile, working both TCP in Congestion avoidance phase. Just with a unique exception, that in Figure 6.4.e there was a timeout in second 27.15 that brought down the performance of both TCPs. You may find in appendix C.1.2 the complete set of simulations we made about this subsection.

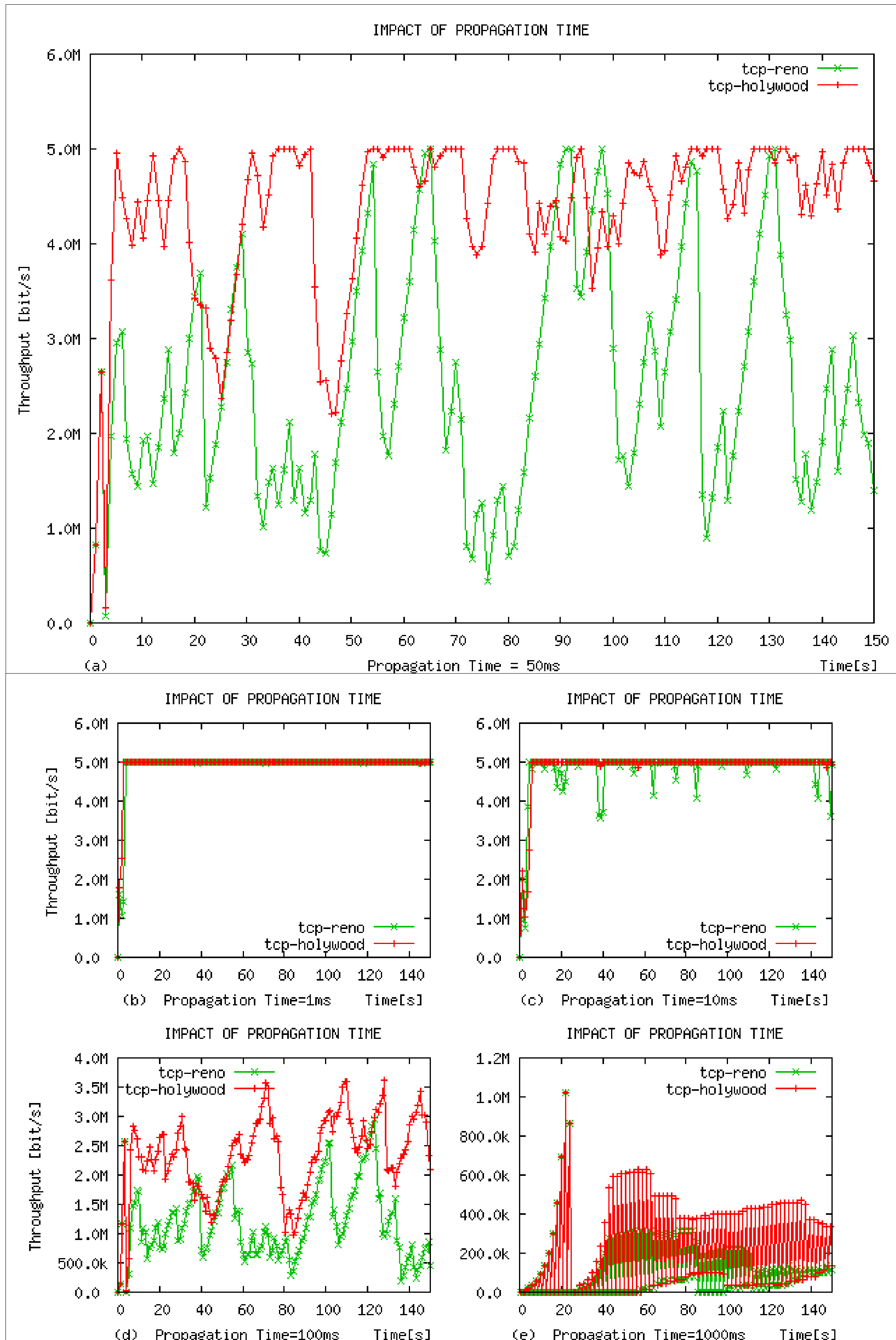

Figure 6.4: Throughput versus Time with Different Propagation Time Values

Now we will observe how the impact of Propagation Time affects the throughput. In Figure 6.5, Average Throughput Versus Propagation Time, in all the points we measured the TCP HolyWood has a higher throughput in comparison with TCP Reno, with exception of one point with propagation time of 5 ms in which the latter is higher with a slighter difference of 3.49 Kbit/s.

The coefficient of variation that TCP HolyWood presents in all the point of Figure 6.5 is lower than TCP Reno, but as before our proposal lost against the standard slightly in the point of 5 ms. A detailed statistical table may be found in the appendix A, Table A.1.2 In the next Figure 6.6, we will see how much the gain in throughput performance of TCP HolyWood versus TCP Reno is.

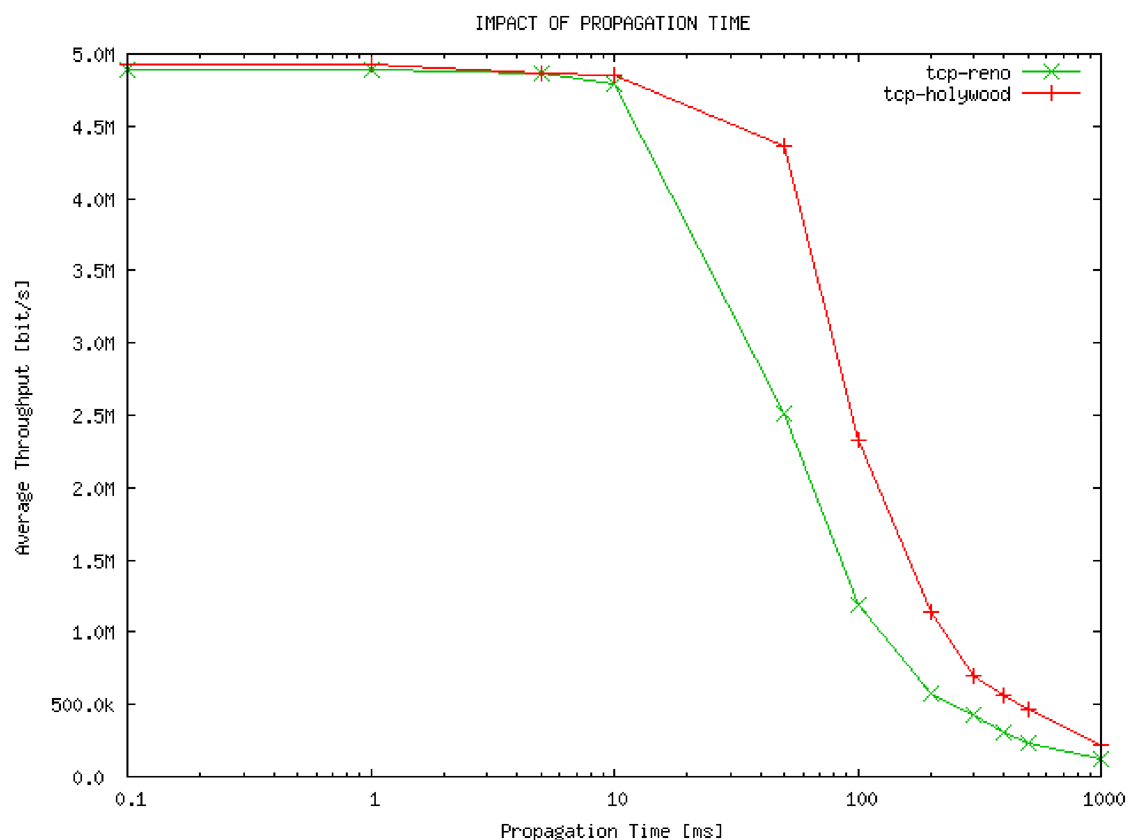

Figure 6.5 - Average Throughput versus Propagation Time

In this Figure 6.6, the ratio of TCP HolyWood to TCP Reno versus Propagation time, we observed that in the interval of propagation time from 0.1 ms to 10 ms there was not much difference, TCP HolyWood is higher for 0.67%. From Propagation Time of 10 ms to 100 ms the throughput of TCP HolyWood is still higher with an average throughput of 57.17%. In the propagation time interval of 100 ms to 1000 ms the throughput of TCP HolyWood is the highest of all the intervals we measured, with an average throughput of 83.82% in all the cases as before as shown we used TCP Reno as a base. This methodology of analysis was taken from Jain (1991, p.165-167). In all the values of propagation time, we measured; the average throughput of TCP HolyWood outperforms TCP Reno in 53.59% with the latter as a base. We used an Awk program to do the measurement; it is in the appendix B.9

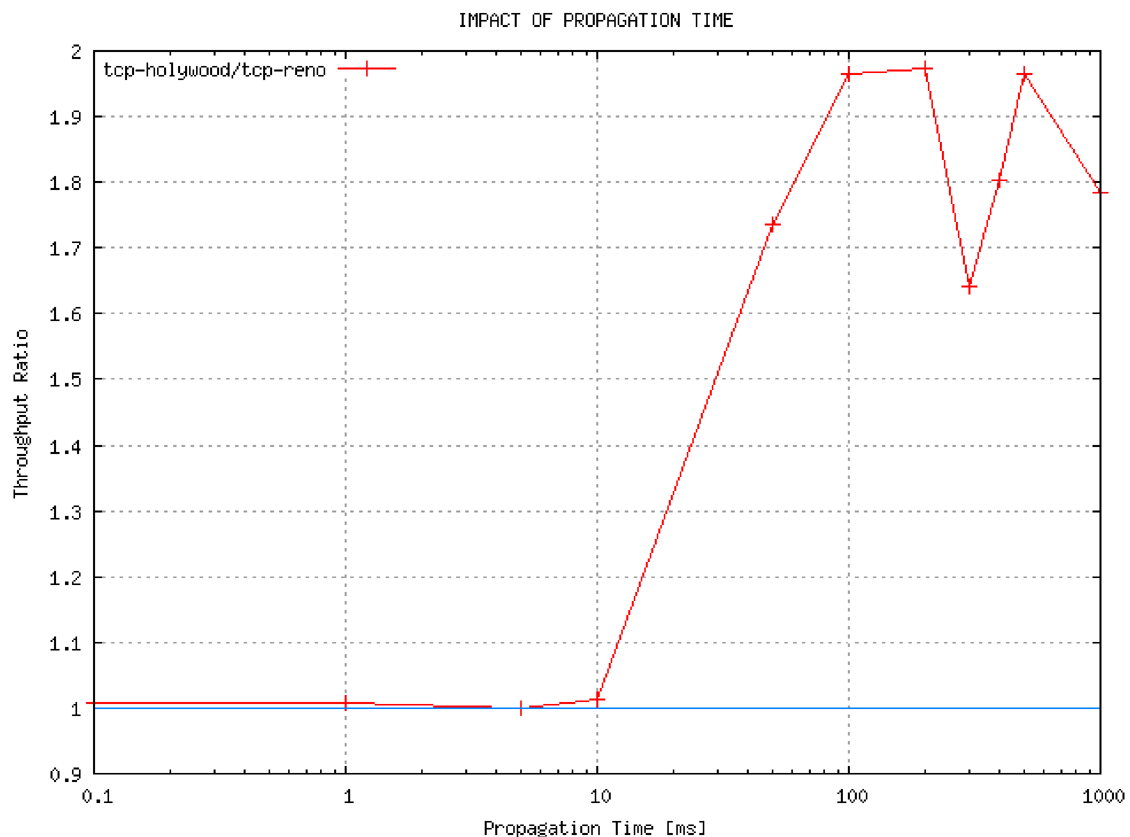

Figure 6.6: Throughput Ratio versus Propagation Time

6.3.3 Impact of Bottleneck Bandwidth in Throughput

In this subsection, we want to know, what was going on with the throughput of TCP HolyWood and TCP Reno, when we varied the Bottleneck Bandwidth from 1 Mbit/s to 100 Mbit/s. In the Figure 6.7, we presented the throughput Versus Time with different Bottleneck Bandwidth values; a different TCP was used in Topology 1 at a time. In all the simulated experiments of this subsection, we used Propagation Time of 35ms and an error rate of 0.1%

We did not take into account as before, the first 10% of the simulation time in our observations, it meant 15 seconds from 150 seconds as a rule of thumb due to warp up time (COUTINHO, 2003)(MACDOUGALL, 1987). In Figure 6.7.a, analyzing the 15 seconds onwards, a steady-state cyclic regime of TCP is attained and TCP is always in Congestion avoidance phase, even in this simulation, TCP HolyWood presented better throughput performance both TCPs oscillate more. The range of TCP HolyWood is from approximately 11 Mbit/s to 2 Mbit/s and the range of TCP Reno went from 7.8 Mbit/s to 1 Mbit/s. In Figure 6.7.b, practically both TCPs had just the same throughput Performance, except that TCP Reno has two small falls. Figure 6.7.b, 6.7.c, 6.7.d had slightly difference, but TCP HolyWood kept always higher than TCP Reno. A detailed statistical Table may be found in the appendix A, Table A.1.3

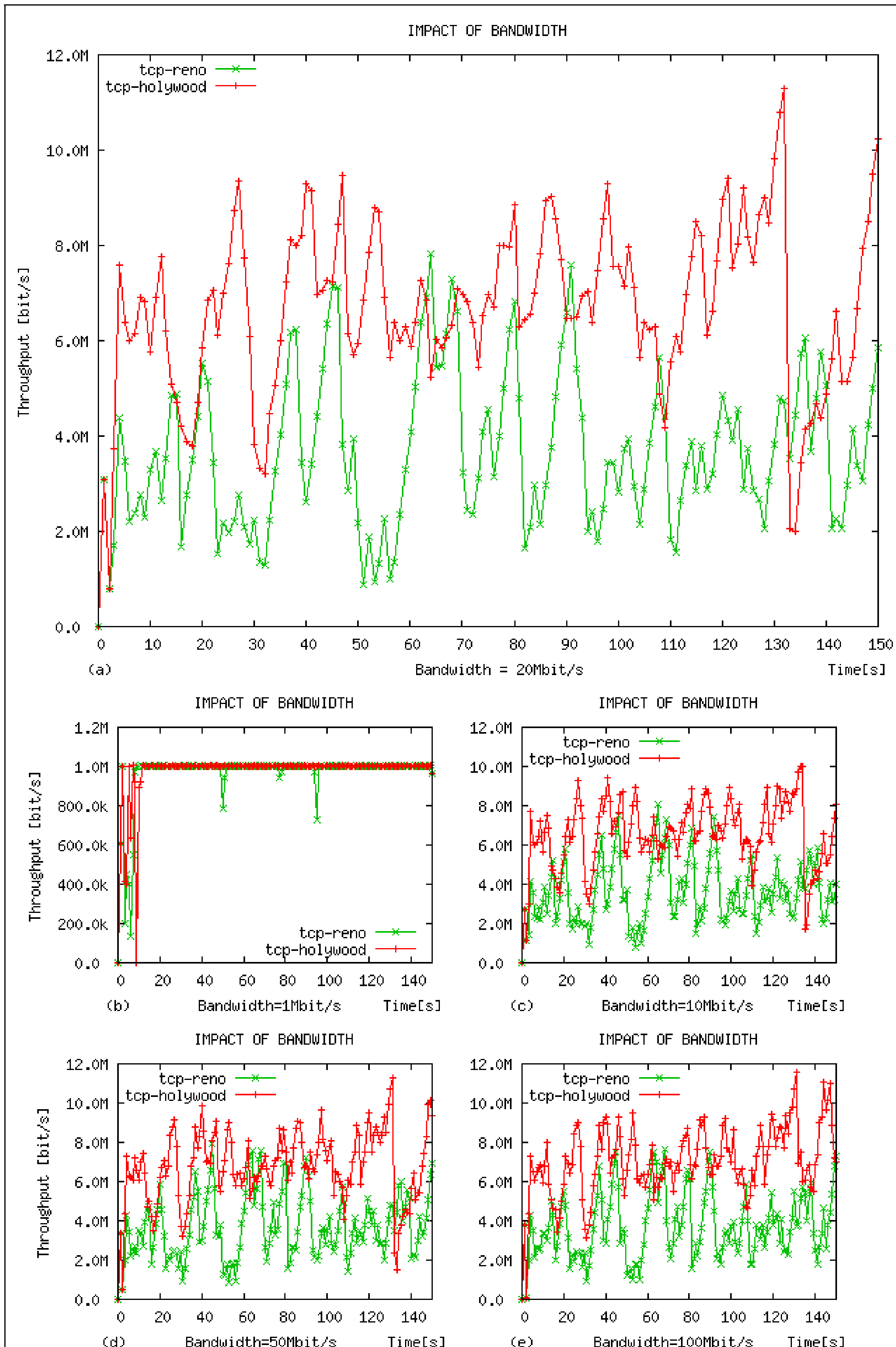

Figure 6.7: Throughput versus Time with different Bottleneck Bandwidth values

In Figure 6.8, we observe the average Throughput Versus Bottleneck Bandwidth, and we may discern that in all the points of bottleneck bandwidth with an error rate of 0.1%, a propagation time of 35 ms and a different TCP being used in Topology 1 at a time the TCP HolyWood outperform higher than TCP Reno.

In all the bandwidth-axis, the Coefficient of Variation of TCP HolyWood was smaller than TCP Reno. This information and the confidence Intervals of the data of figure 6.8 is found in appendix A, Table A.1.3. In the next Figure 6.9 we will see how much is that increase of throughput performance.

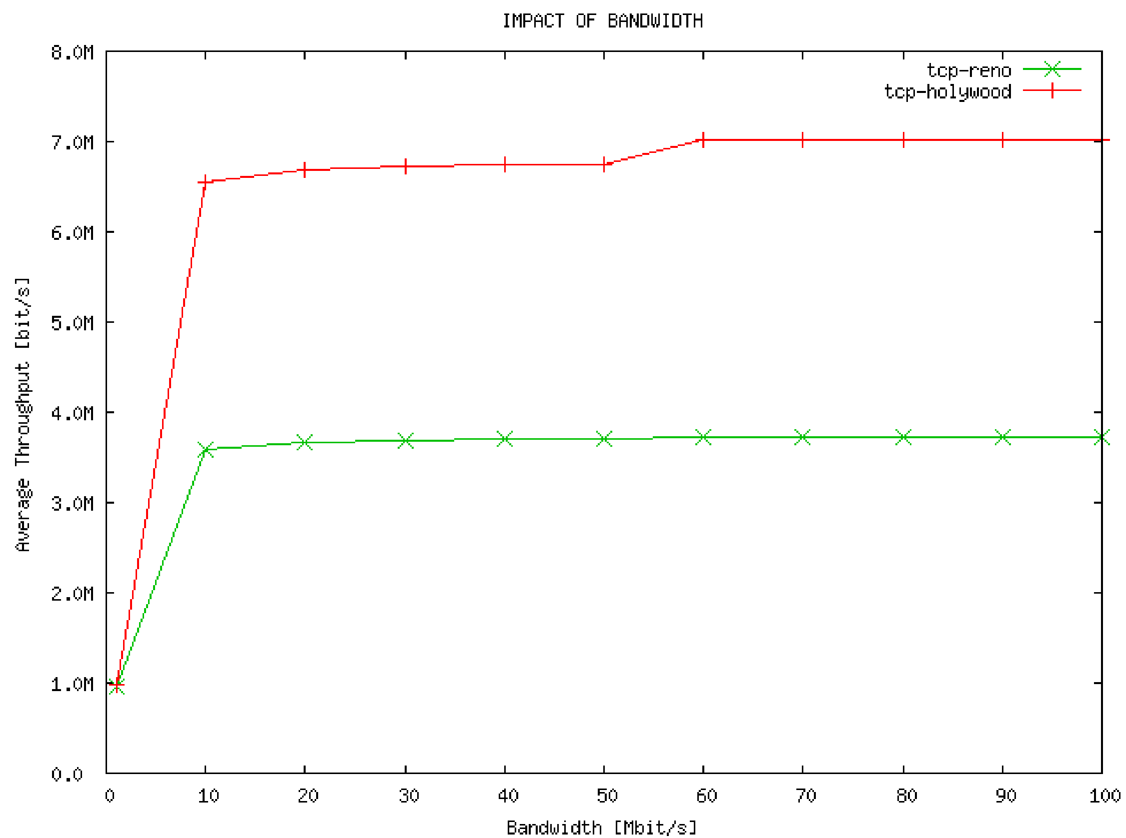

Figure 6.8: Average Throughput versus Bottleneck Bandwidth

In the Figure 6.9, Throughput Ratio Versus Bandwidth, the interval of 1 Mbit/s to 10 Mbit/s the average of throughput of TCP HolyWood to TCP Reno was 41.42% higher than TCP Reno; this value was not so precise since we just had two Values. From 10 Mbit/s to 50 Mbit/s, the average of throughput of TCP HolyWood is 82.04 % higher than TCP Reno. From 50 Mbit/s to 100 Mbit/s, the average of throughput of TCP HolyWood was 87.13% higher than TCP Reno.

A general measurement of all the point of Bottleneck bandwidth in Figure 6.9, gave us an average throughput of TCP HolyWood of 77.49% higher than TCP Reno. In all the cases, we took TCP Reno as a base or denominator. This analytical methodology was taken from Jain (1991, p165-167).

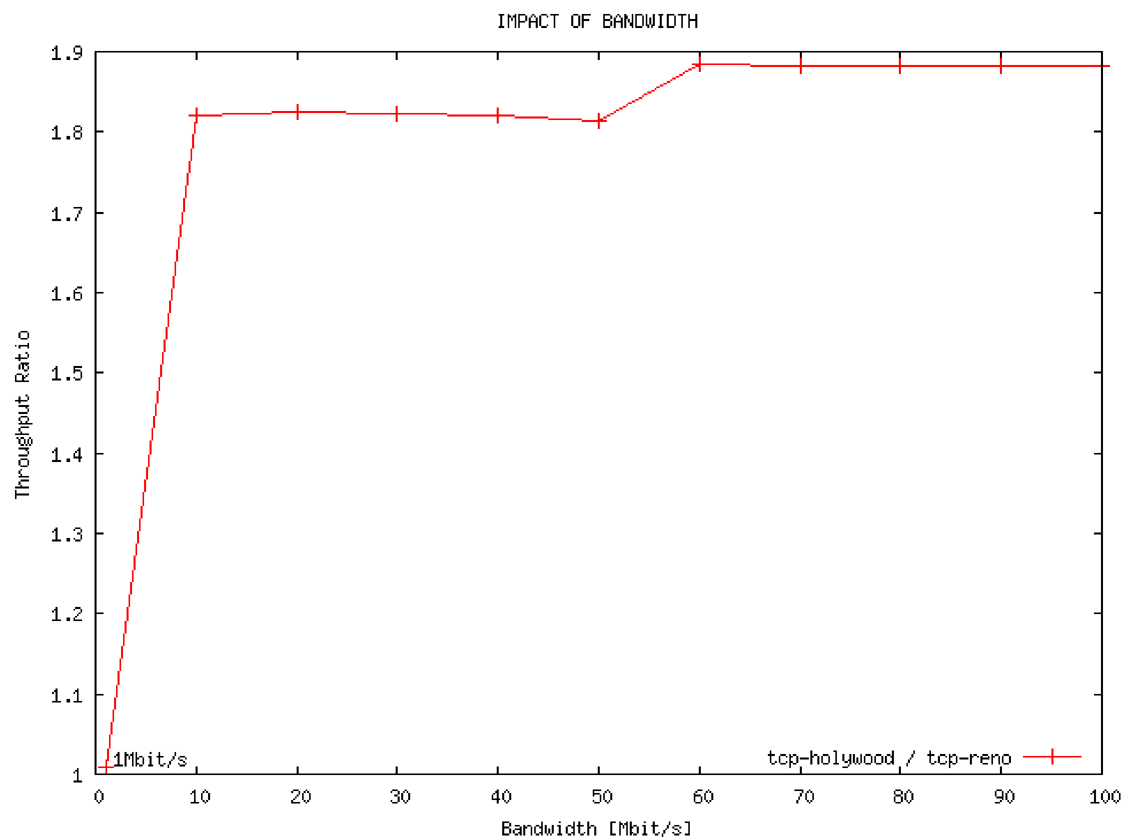

Figure 6.9: Throughput Ratio versus Bandwidth

6.3.4 Impact of Error Rate on Jitter

In this subsection, we analyze how the Jitter was affected by a varying and increasing error rate. When we worked with a jitter metric, what we hope to find was a lower or smaller value and a smooth curve, much better if it is constant, because we may affirm that the jitter is zero, meaning that we would not have variations of delays or expressed in other words constant packet latency.

In the Figure 6.10 we presented the Jitter Versus Sequence Number for different Error Rates, a different TCP was used in Topology 1, one at a time. In all the simulated experiments of this subsection, we used Propagation Time of 35 ms and a bottleneck bandwidth of 5 Mbit/s. We started to analyze Figure 6.10 since a sequence number of 500 or 0.5K due to that approximately after this values the system became stable.

In Figure 6.10.a, we might examine that TCP HolyWood, with color red, displayed less variations than TCP Reno. If it would not be for two positive outlier before the sequence number 20 K. and two small outliers between sequence number 60K and 65K we might declare that our proposal was practically constant, instead, TCP Reno demonstrate several positive outliers, more than 20. Whereas, in figure 6.10.a. We might also discern that the last sequence number of TCP Reno is approx. 45K and TCP HolyWood is 65K.

In Figure 6.10b both TCP were practically constant with the last same sequence number of approx. 65 K., nevertheless our proposal presented slightly less Jitter. In Figure 6.10.c, TCP HolyWood was practically constant and TCP Reno displayed more than 8 outliers. In Figure 6.10.d, both TCPs showed an average jitter of less than 200 ms and TCP Reno revealed a higher sequence number.

Finally, in Figure 6.10e, both TCPs presented positive and negative outliers and the curves were quite rough. In general, we might scrutinize for both TCPs when the Error Rate increased also was increased the Jitter together with the outliers. You may find in appendix C.1.4 the complete set of simulations we made about this subsection.

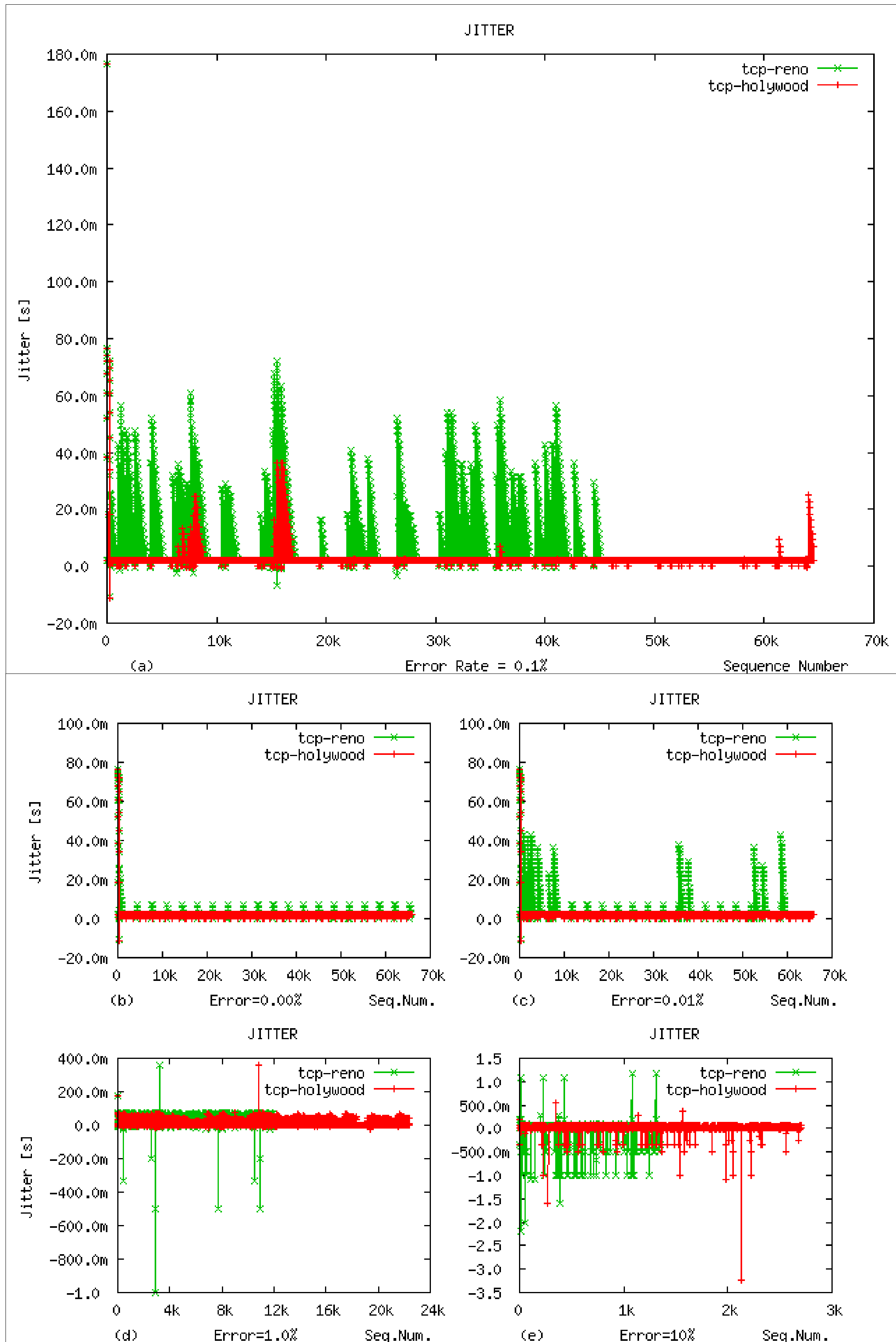

Figure 6.10: Jitter versus Sequence Number for different Error Rates

In Figure 6.11a and 6.11b, average jitter versus error rate, in the error rate-axis, from the interval of 0% to 6%, approximately the Average jitter of TCP HolyWood was lower than TCP Reno; TCP Reno had 32.21% more jitter than TCP HolyWood, using the latter as a base for the calculation of this percentage. From 6% to 60%, TCP Reno was higher than TCP HolyWood with 35.74% more jitter with the latter as a base as we observed in Figure 6.11b, this interval was strongly influence with negative outlier after 10% of error rate. If we analyze in all the measured values from an error Rate of 0% to 60% with TCP HolyWood as a base, TCP Reno had 38.28% more jitter than our proposal.

Due to the presence of negative jitter in the average values of both TCPs and because we were using ratios, we calculated the absolute value of the ratio TCP Reno to HolyWood. So in this way, would be presented as higher those values that would be far away from zero being them positive or negatives and will be presented as lower otherwise.

Moreover, in Figure 6.12, COV of the average jitter versus error rate, and with appropriate scale, we observed that Coefficient of variation of TCP HolyWood was smaller and linear than the TCP Reno with the exception of one point in 0.04% of error rate. About the interval of confidence of both TCPs, you may find them in Appendix A, Table A.1.4.1. To arrive, this analysis we extracted that information from appendix A, Tables A.1.4.1, and A.1.4.2, the latter Table was based in Jain (1991, p.167).

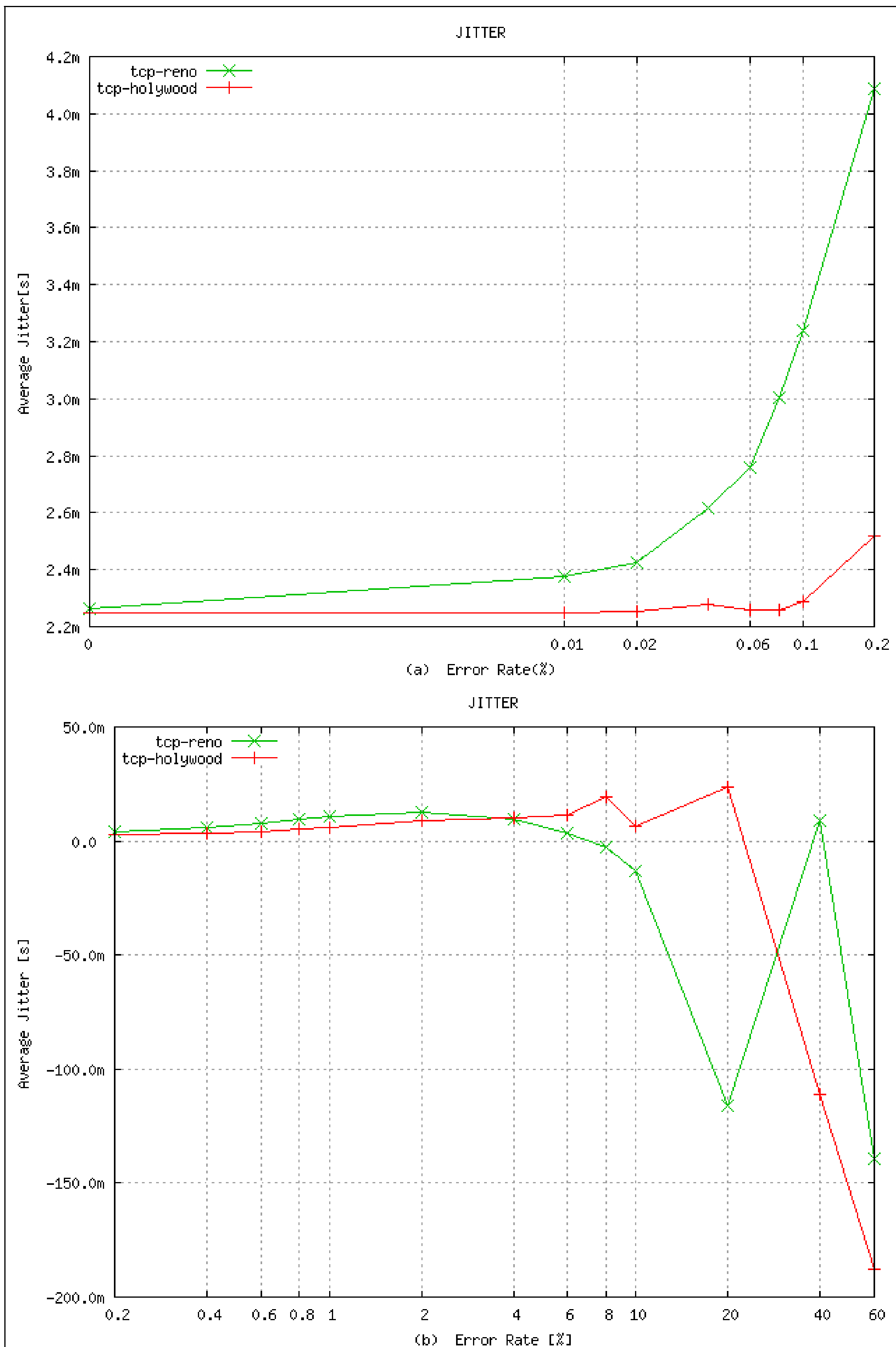

Figure 6.11: the Average Jitter versus Error Rate

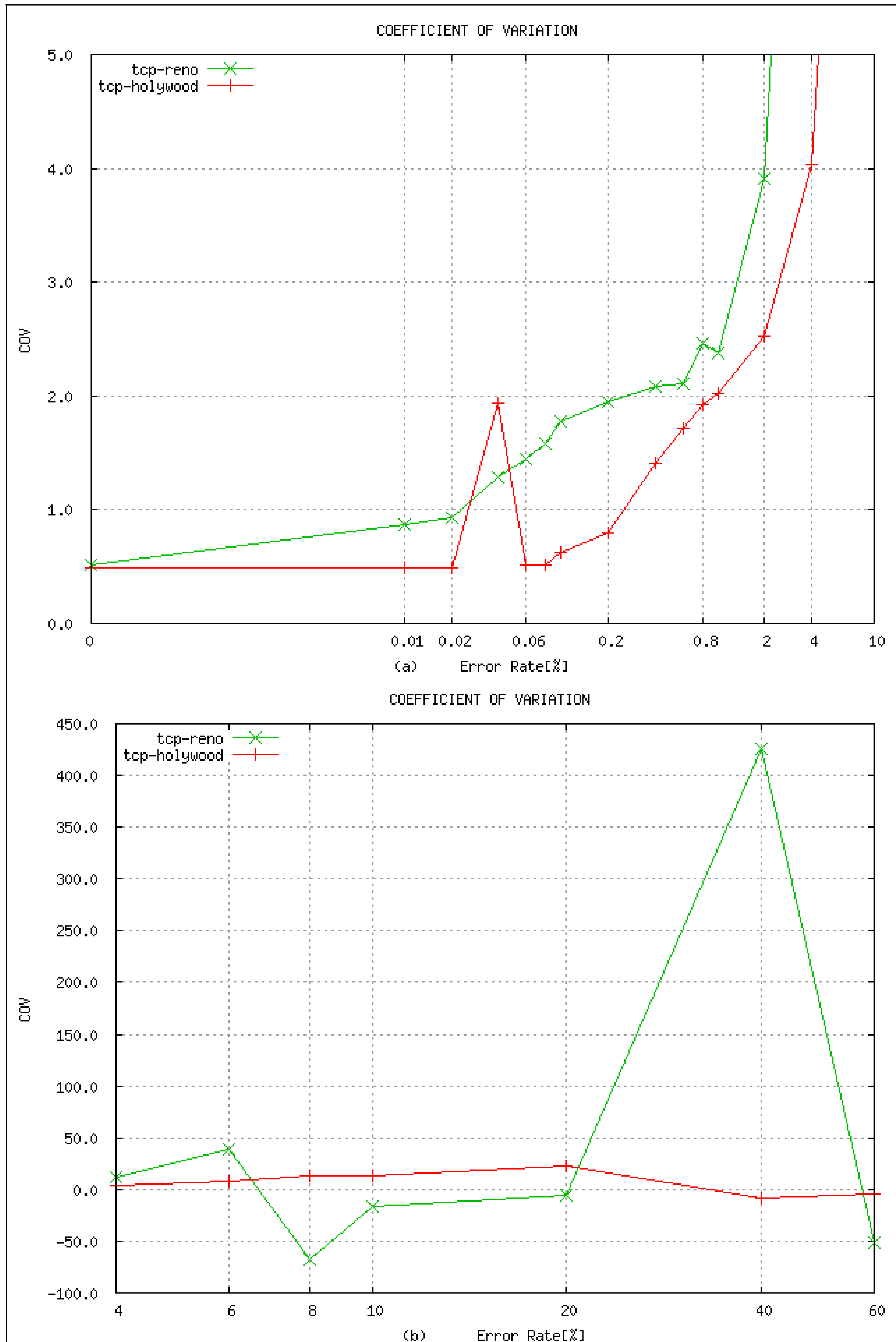

Figure 6.12: COV of the Average Jitter versus Error Rate

6.3.5 Impact of Propagation Time on Jitter

As we observed, in figure 6.13.a, jitter versus sequence number with different propagation time values, with an error rate of 0.1% and a bottleneck bandwidth of 5 Mbit/s, the TCP HolyWood presented lower jitter than TCP Reno as well as produced a higher sequence number. The jitter in both cases were rougher than in Figure 6.10.a due to great number of positive jitter outliers giving the aspect of a saw shaped figure. TCP Reno produced the highest outlier with 100 ms against 60 ms of TCP HolyWood. Mostly the TCP Reno produced higher positive jitter outliers than TCP Reno causing that is average jitter would be higher than our proposal.

According to the increase of the propagation time as we observed in Figure 6.13.b, 6.13.b, 6.13.d and 6.13.e, the jitter increased as well. Besides, in figure 6.13.b for instance the jitter was practically a constant, meant that there was not variation of packet delay; nevertheless in Figure 6.13.e the positive jitter outliers and irregularity in the shape of the figure were the constant. As always, you may find in appendix C.1.5 the complete set of simulations we made about this subsection.

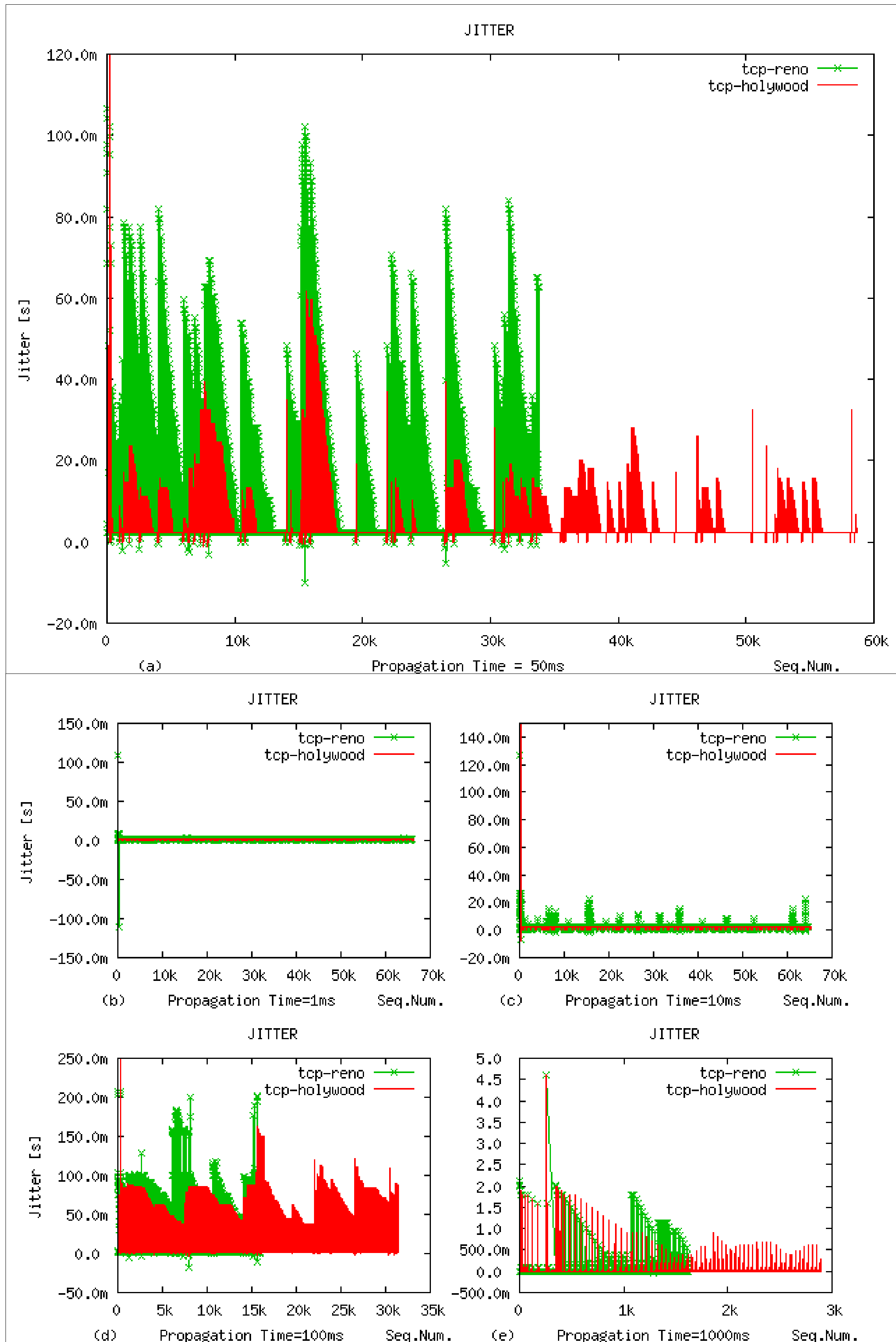

Figure 6.13: Jitter Versus Sequence Number with different Propagation Time Values

We calculate, now, how much greater or lower was the Jitter of TCP HolyWood against the TCP Reno. As we observed, in Figure 6.14, average jitter versus propagation time, in the entire propagation time axis TCP HolyWood showed a lesser jitter than TCP Reno. From the interval of 0.1 ms to 50 ms, with TCP HolyWood as a base, TCP Reno showed 14.84% more jitter than TCP HolyWood. From the interval of 50 ms to 1000 ms, with TCP HolyWood as a base, TCP Reno showed 82.15% more jitter than TCP HolyWood

Generally, in all the route of propagation time, with TCP HolyWood as a base, TCP Reno showed 52.49% more jitter than TCP HolyWood. As before, to obtain this percentages we extracted these information, after a proper processing based in Jain (1991, p.167), from appendix A, Tables A.1.5.1, and A.1.5.2.

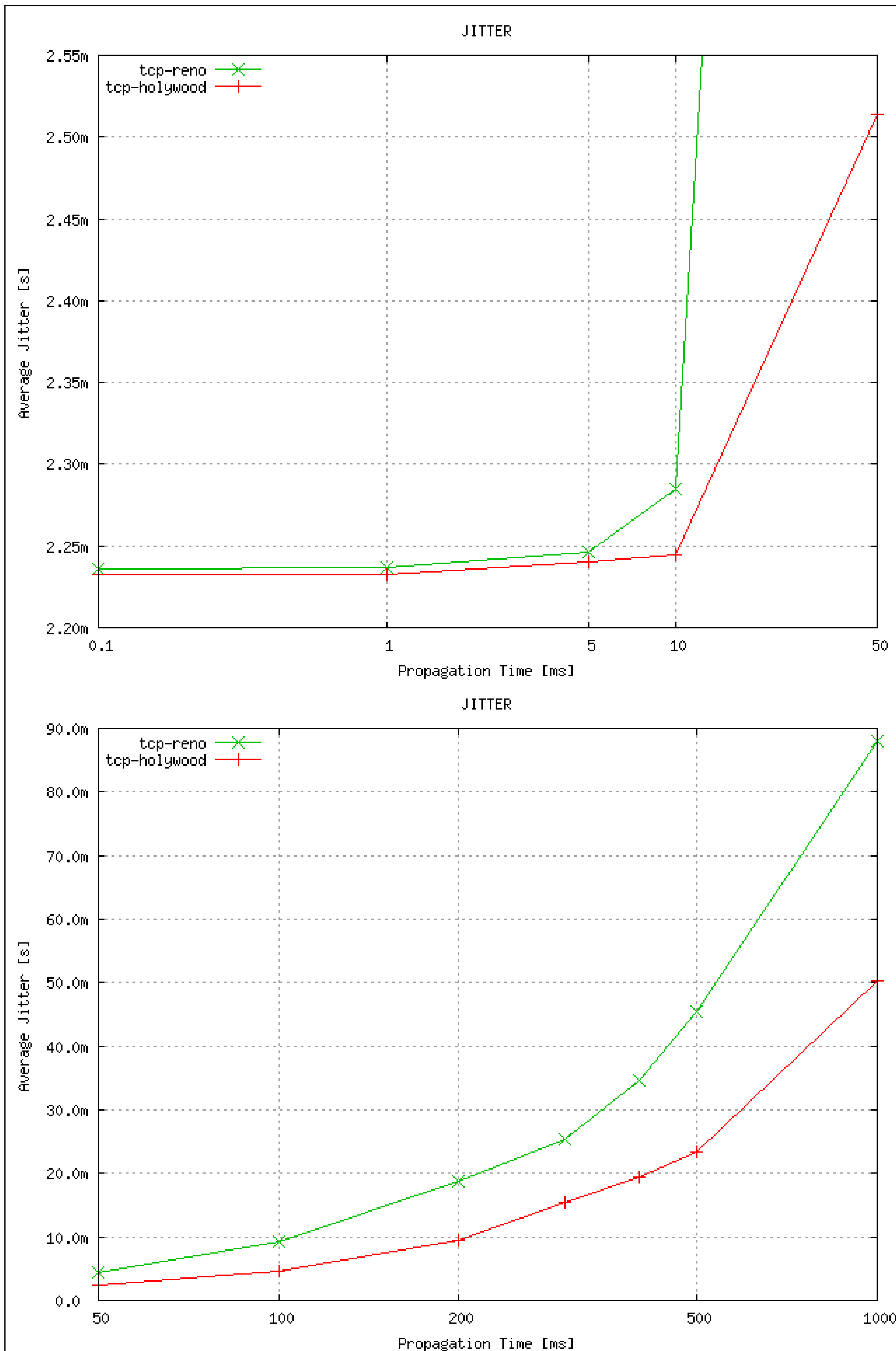

Figure 6.14: Average Jitter versus Propagation Time

About the coefficient of variation of the curves of both TCPs, in figure 6.15, looked like enlaced each other giving the impression of a continuous hug in several point avoiding us to observe which one had more variations, moreover our proposal was a slightly more linear shaped figure than TCP Reno, for all the route of propagation time axis. As a conclusion, we declare that both TCPs had similar Coefficient of variation. A list of the values of the coefficient of variation, as well as, the confidence intervals at 95% are found in appendix A, Table A.15.1.

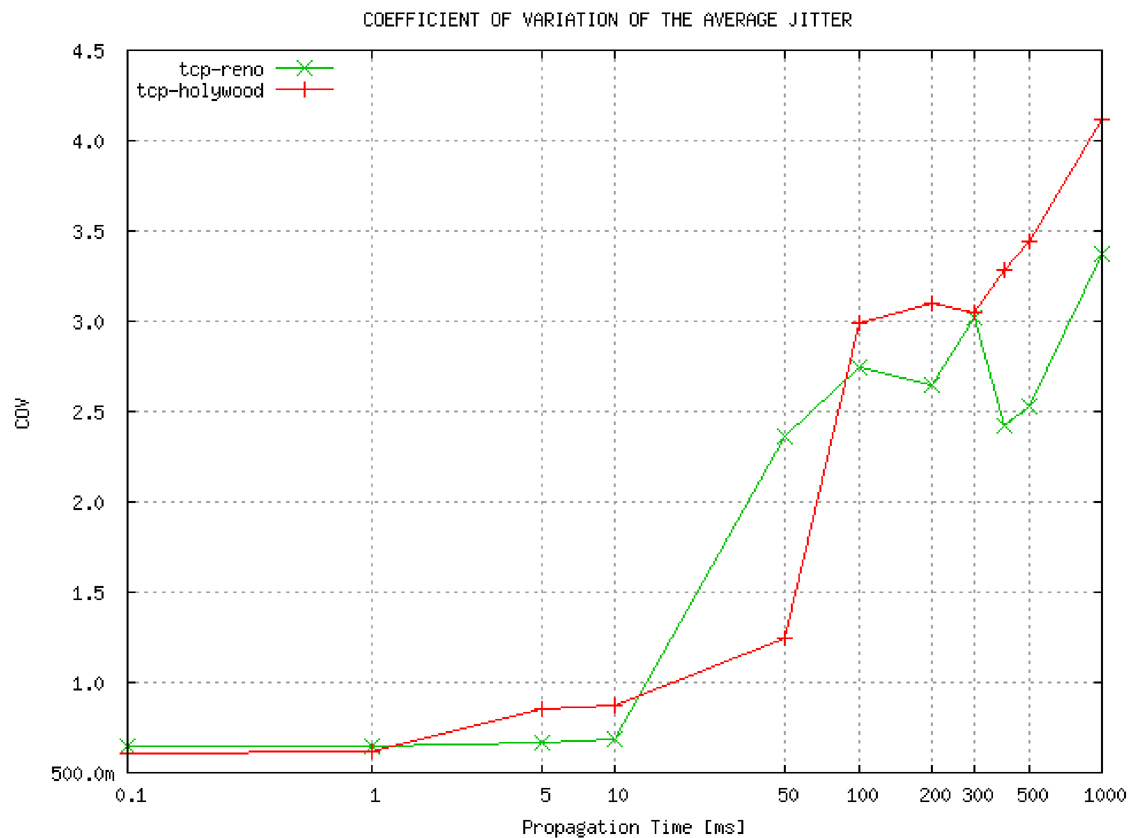

Figure 6.15: COV of Average Jitter versus Propagation Time

6.3.6 Impact of Bottleneck Bandwidth on Jitter

In figure 6.16.a, jitter versus sequence number with different bottleneck bandwidth values, we do not consider the first 500 sequence numbers due to warm up time, with error rate of 0.1%, propagation time of 35 ms, and bottleneck bandwidth of 20 Mbit/s. It was used the Topology 1, each TCP one at a time, we might scrutinize that TCP HolyWood presented lesser positive jitter than TCP Reno, and approximately a double value of Sequence Number. Both TCP presented a rough jitter. Besides, in figure 6.16.b with 1 Mbit/s, we observed that the jitter shift from zero to 10ms in the entire bottleneck bandwidth axis route and with a sequence number of 13 K. Whereas, in figure 6.16.c, TCP Reno displayed more jitter than TCP HolyWood and approximately with a maximum sequence number of the half.

Finally, in Figure 6.16.d and 6.16.e, with a bottleneck bandwidth of 50 Mbit/s and 100 Mbit/s both TCP presented similar jitter being TCP Reno slightly higher in jitter and with a half of the maximum sequence number than our proposal. As always, you may find in appendix C.1.6 the complete set of simulations we made about this subsection.

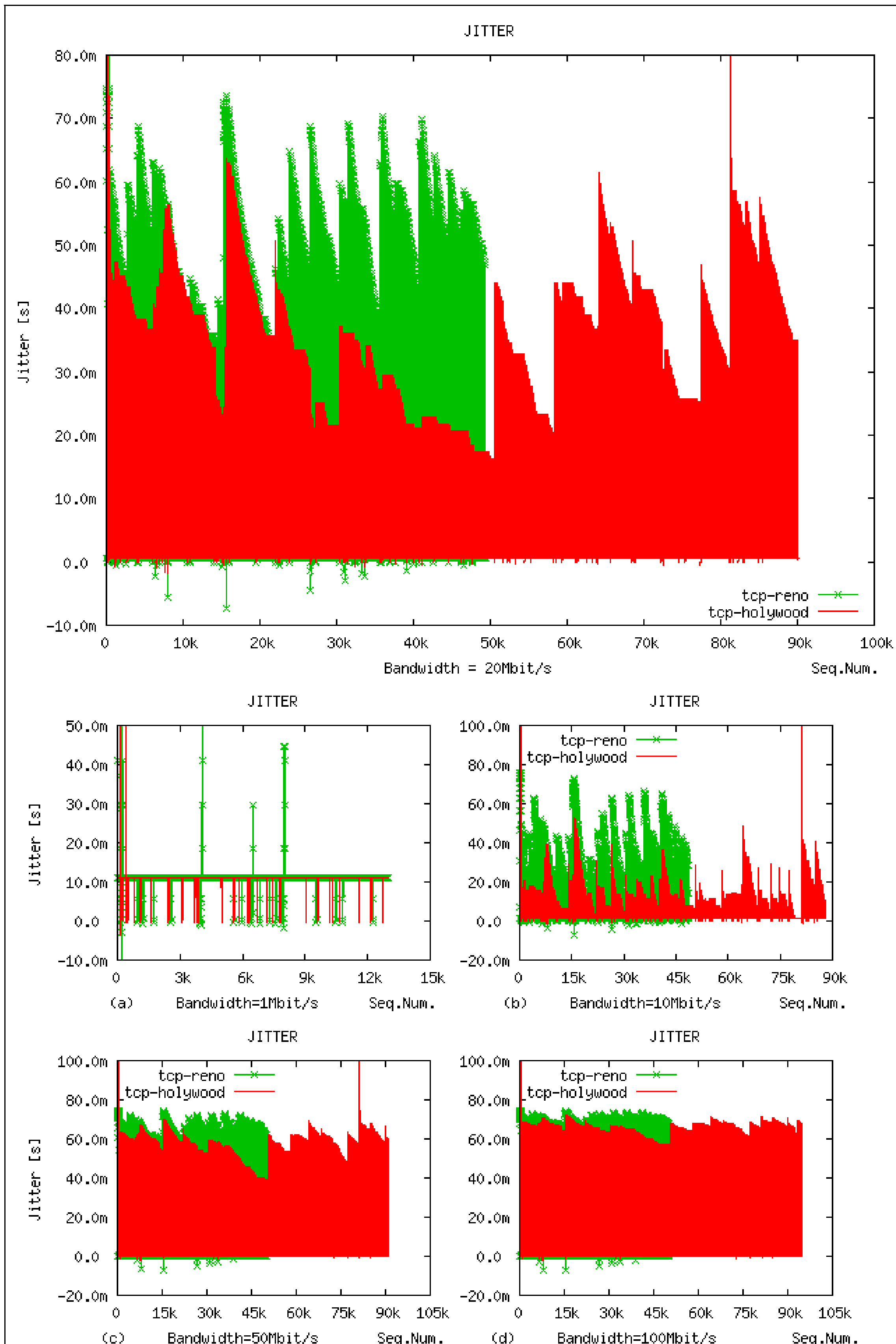

Figure 6.16: Jitter versus Sequence Number with Different Bottleneck Bandwidth values

In the same way, we explained earlier, we quantized how much was the Jitter of TCP HolyWood against the TCP Reno. As we observed, in figure 6.17, average jitter versus bottleneck bandwidth, in the entire propagation time axis TCP HolyWood showed a lesser jitter than TCP Reno.

Analyzing the interval from of 1 Mbit/s to 10 Mbit/s, with TCP HolyWood as a base, TCP Reno showed 40.96% more jitter than TCP HolyWood. We might declare that because of calculating this interval with just two values this average was not so representative. From the interval of 10 Mbit/s to 100 Mbit/s, with TCP HolyWood as a base, TCP Reno showed 84.77% more jitter than TCP HolyWood

Predominantly, in all the route of bottleneck bandwidth, with TCP HolyWood as a base, TCP Reno showed 76.81% more jitter than TCP HolyWood. As before, to obtain this percentages we extracted these information, after a proper processing based in Jain (1991, p.167), from appendix A, Tables A.1.6.1, and A.1.6.2.

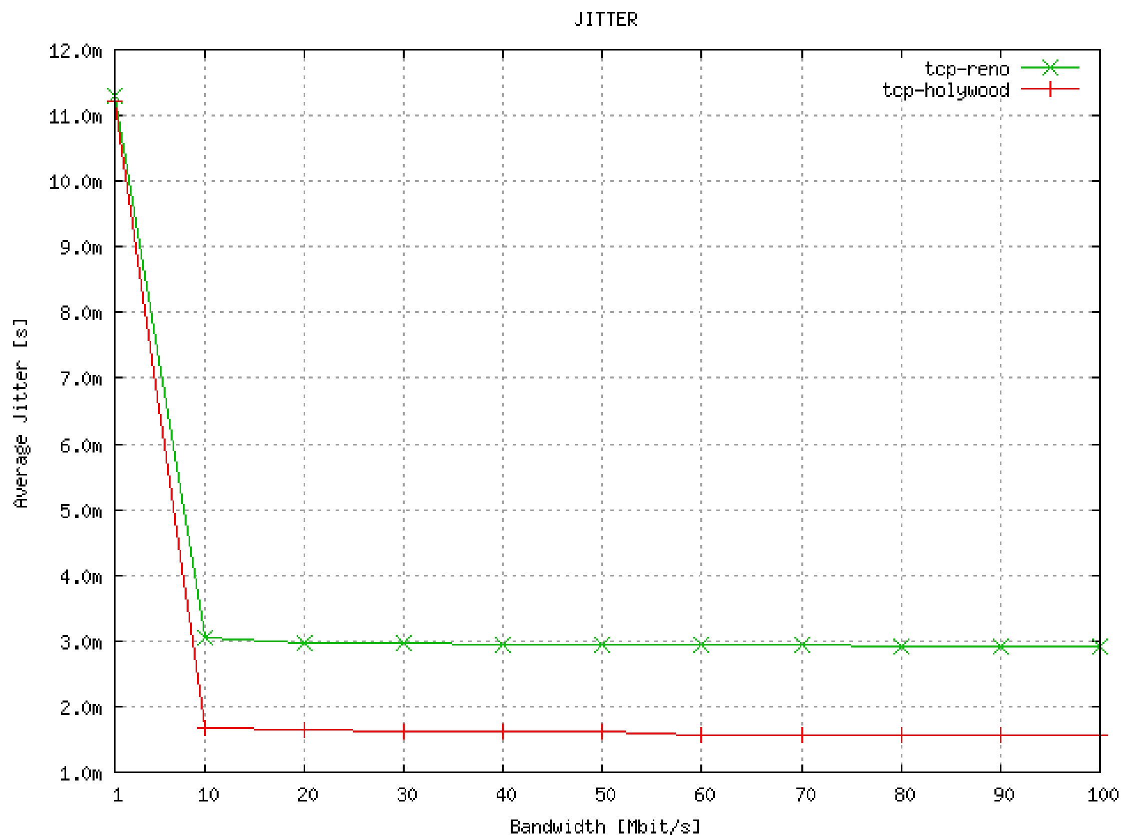

Figure 6.17: Average Jitter versus Bottleneck Bandwidth

However, in this subsection, we had a drawback in the statistics, as it may be seen the coefficient of variation of TCP Reno is lesser than TCP HolyWood, this time the former, in Figure 6.18, COV of the average jitter versus bandwidth was better than our proposal. A list of the values of the coefficient of variation, as well as, the confidence intervals at 95% are found in appendix A, Table A.16.1

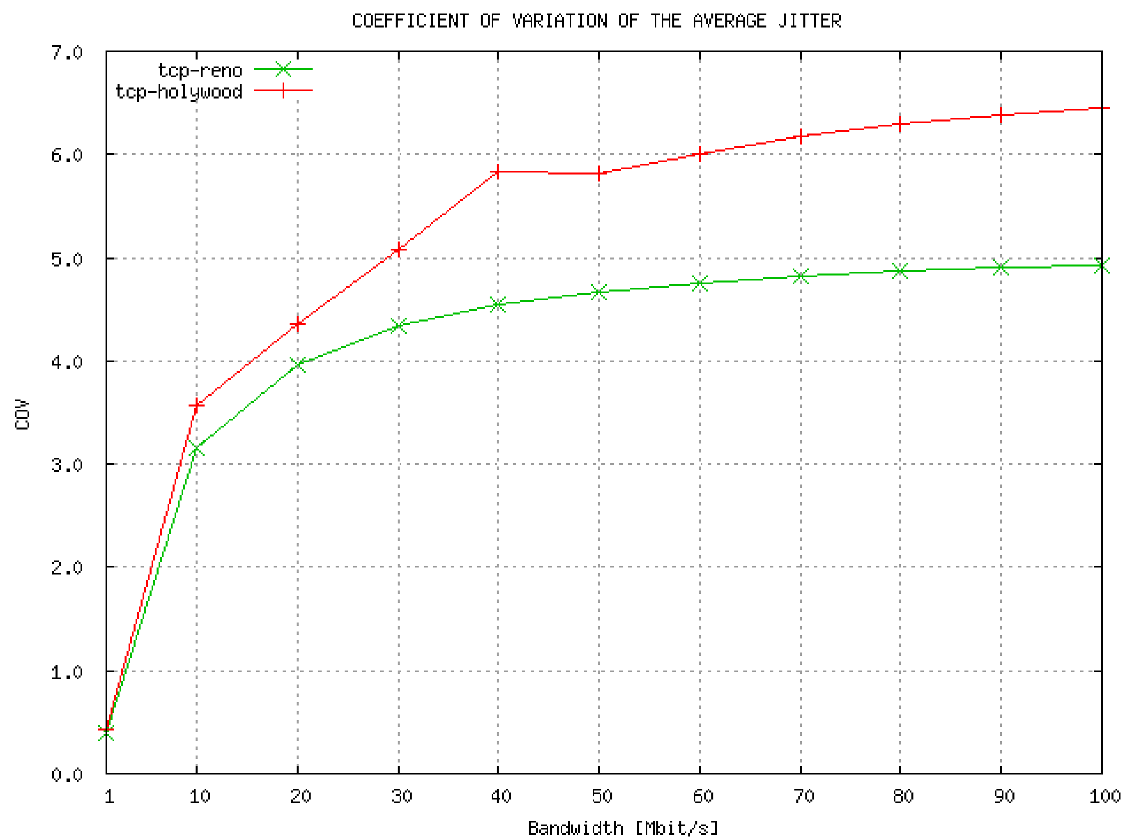

Figure 6.18: COV of the Average Jitter versus Bandwidth

6.3.7 Percentage of lost packets

In figure 6.19.a, percentage of lost packets of TCP HolyWood versus TCP Reno, from an error rate interval of 0% to 0.08% TCP Reno presented a lower percentage of lost packets of 14.82% in comparison with TCP HolyWood, being the latter as a base.

In Figure 6.19.a also, from an error rate interval of 0.08% to 1% TCP Reno presented a lower percentage of lost packets of 7.87% in comparison with TCP HolyWood, being the latter as a base. In Figure 6.19.b from an error rate interval of 1% to 60%, TCP Reno presented a lower percentage of lost packets of 15.31% in comparison with TCP HolyWood, being the latter as a base.

In all the route of error rate axis of Figure 6.19 TCP Reno presented a lower percentage of lost packets of 8.61% in comparison with TCP HolyWood, being the latter as a base. We noticed that both TCP has sent different quantity of packets for the same simulation. We extracted this information from Trace graph, the window of nerwork information, one point for each Error Rate value. Arranged the data and process it as shown in appendix A.3.1 Tables A.3.1.1, A.3.1. Moreover, for finding the final comparison Percentages of TCP HolyWood and TCP Reno we used the methology of Jain (1991, p.165-167).

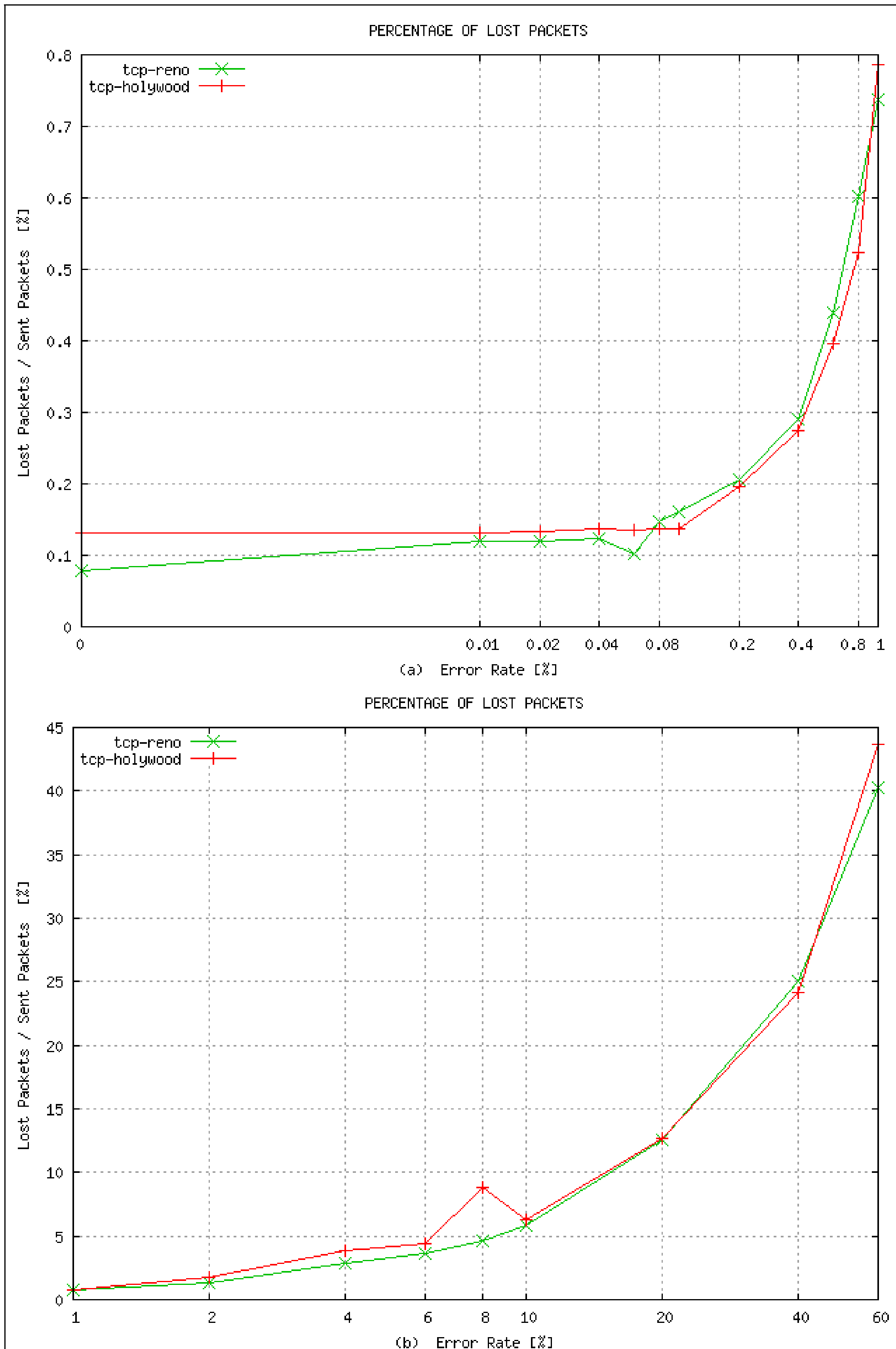

Figure 6.19: Percentage of Lost Packets of TCP HolyWood Vs. TCP Reno

6.3.8 Latency

As stated by Jain (1991) ''*After a few experiments it was clear that throughput and delay were redundant metrics. All schemes that resulted in higher throughput also resulted in higher delay*". To test this statement with a minimum, medium and maximum value of error rate in our topology 1 with 35ms of propagation time, the results were as follows:

Table 6.1: Statistics of Latency of TCP HolyWood and TCP Reno

Error Rate [%]	Protocol [TCP]	Average Latency [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
0	Reno	62.615	25.163	40.187	58.588	66.642
0	HolyWood	75.74	28.936	38.204	71.109	80.37
0.1	Reno	44.243	11.745	26.547	42.363	46.123
0.1	HolyWood	56.519	20.149	35.652	53.294	59.744
10	Reno	39.218	1.982	5.056	38.901	39.535
10	HolyWood	41.854	10.798	25.799	40.126	43.583

In all the tested we accomplished, between TCP Reno and TCP HolyWood, our proposal had the higher average latency or packet delay as shown in the Table with a difference of approximately of 13.6 ms, 12.3 ms and 2.6 ms with error rates of 0.0%, 0.1% and 10% respectively. We observed as well as that, the difference was diminishing while the Error rate of the link increased together with the variability of the data.

6.3.9 Fairness

In Figure 6.20.a, Fairness of TCP HolyWood and TCP Reno, using the Jain's index (JAIN, 1991, p.36). We observed that TCP HolyWood is as fair as TCP Reno both TCP are practically in one, with slight variations.

In order to clarify those variations we did a zoom and obtained Figure 6.20.a and TCP Reno presented more variability than our proposal. The values of the coefficient of variation and the confidence intervals with 95% of the Figure 6.20 may be found in appendix A, Table A.1.7.

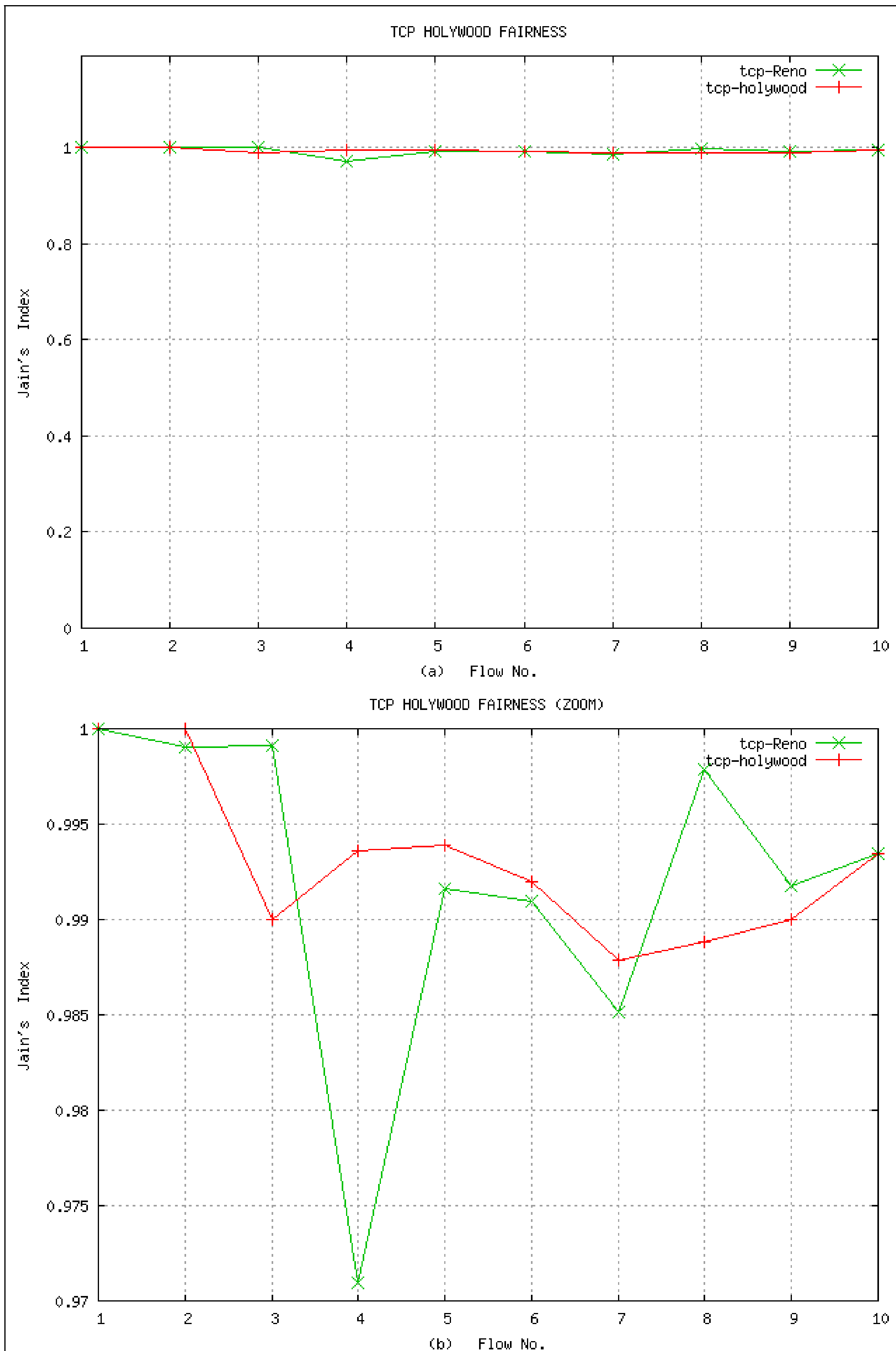

Figure 6.20: Fairness of TCP HolyWood and TCP Reno

In order to have an additional insight of the fairness, in Figure 6.21 average throughput versus increasing number of flows, we observed that the average throughput until three (3) TCP flows of TCP HolyWood was bigger than of the TCPs Reno; but from then to 10 flows both kind of TCPs presented similar average.

We used topology 2 and script B.5 of the appendix B, increasing the number of TCPs in script B.5 from 1 to 10 to get Figures 6.20 to 6.22. We simulate a kind of TCPs at a time.

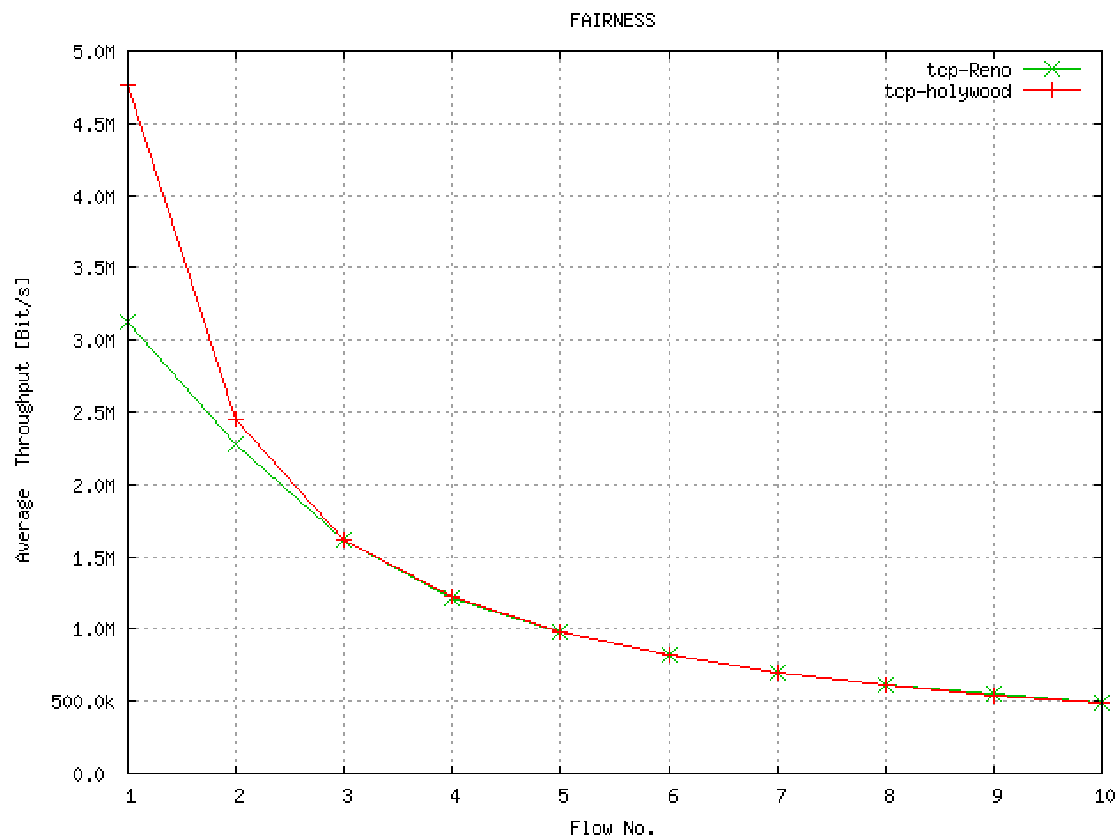

Figure 6.21: Average Throughput versus Increasing Number of Flows

6.3.10 Friendliness

In Figure 6.22, friendliness between TCP HolyWood and TCP Reno, we deployed, a propagation time of 35 ms, a bottleneck bandwidth of 5 Mbit/s, an error rate of 0.1% and the topology 2 with a modification, the first flux of TCP Reno was changed by TCP HolyWood and the others remaining fluxes were TCP Renos. We observed also that accordingly the increase of the number of TCPs Reno competing with a single TCP HolyWood, the latter became friendlier with the other TCP Renos.

Lastly, the coefficient of variation in Figure 6.22 of the Average throughput of TCP Westwoood was always smaller in comparison with the average of the increasing number of TCPs Reno. The values of the coefficient of variation and the confidence intervals with 95% of the Figure 6.22 may be found in appendix A, Table A.1.8.

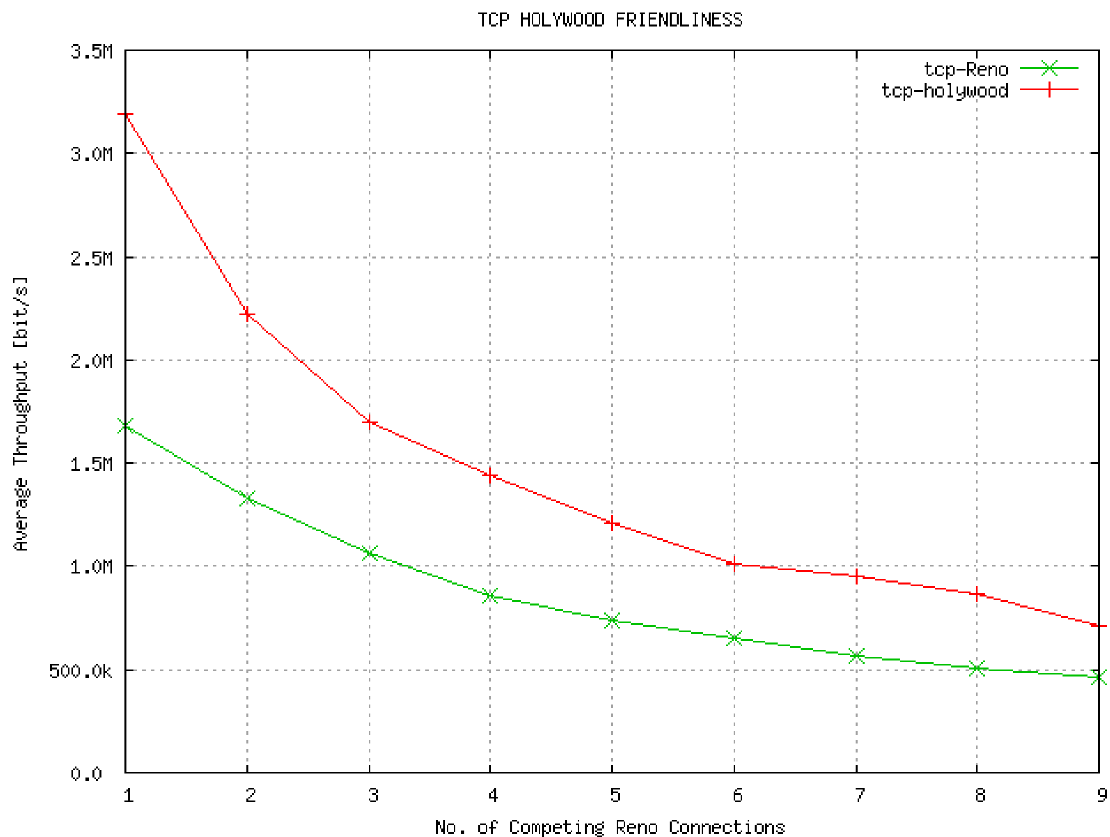

Figure 6.22: Friendliness between TCP HolyWood and TCP Reno

6.4 TCP Holy Wood versus Other TCP Protocols

We present as in subsection 6.3, a set of results of two metrics throughput and jitter using 3 factors, Error Rate, Propagation Time, and Bottleneck Bandwidth to scrutinize the perturbations and interactions of TCP HolyWood in comparison with TCP Westwood and TCP Vegas in a Network Environment.

We did not use replication in our experiments because even if we repeat several times the same simulation and considering a Normal Random Error Rate, and with the same configuration, values apparently there were no changes in the output of the simulation using script the *test-1-simple.tcl*.

The accuracy of our result is limited to what the Network simulator 2, version ns-2.1b8a may offer to us. Only after validating it in a real network with a real implementation of our proposal, we will know how accurate our results will be. We zoomed figures of our results where they were appropriated, using logarithmic and linear scales to obtain through Gnuplot the best possible visual resolution. The round number was of 10 in Trace Graph by default in all its outputs, but in order to fit the results into tables as presented in appendix A, we used integer arithmetics with round numbers of different decimal places, from 1, 2, 3, 4 until 5.

6.4.1 Impact of Error Rate on Throughput

In this subsection, we analyze how the throughput was affected by a varying and increasing error rate. When we analyzed throughput, we hope to find a higher value and as stable as possible. In addition, in the figure 6.23, throughput versus time with different Error rates over TCP HolyWood with other Protocols, in all the simulated experiments of this subsection, a different TCP was used one at a time, we deployed Topology 1; we used propagation time of 35 ms and a bottleneck bandwidth of 5 Mbit/s. We discounted the first 10% of the simulation time in our observations, it meant 15 Sec. from 150 Sec. as a rule of thumb due to warp up time (COUTINHO, 2003) (MACDOUGALL, 1987).

Moreover, in Figure 6.23.a with an error rate of 0.1% we might notice that TCP HolyWood showed a more stable and continuous appearance than TCP Vegas and TCP Westwood with two falls approximately in second 20 and 40. Our proposal is followed closer by TCP Vegas with more falls, and instability in shape. TCP Westwood presented the most variable throughput performance with more than 19 falls diminishing its performance in those points from 5 Mbit/s to less than 1Mbit/s. Besides, in figure 6.23.b without error rate (0%) and 6.23.c with error rate of 0.01% TCP HolyWood and TCP Vegas used the channel link to its maximum values were presenting almost a constant value of 5 Mbit/s. In addition in Figure 6.23.d, with an error rate of 1% the winner was TCP Westwood, presented a better throughput performance, nevertheless our proposal was as good as TCP Vegas. However, TCP HolyWood and TCP Vegas presented lesser variability of the throughput.

Finally, in figure 6.23.e we observed that the Three TCPs were trying to do its better effort, but with a high error rate, we might not distiguish a winner but TCP HolyWood and TCP Westwood seemed to be slightly better in througput performance than TCP Vegas. In all the cases, the variability of the throughput was high. You may find in appendix C.2.1 the complete set of simulations we made about this subsection.

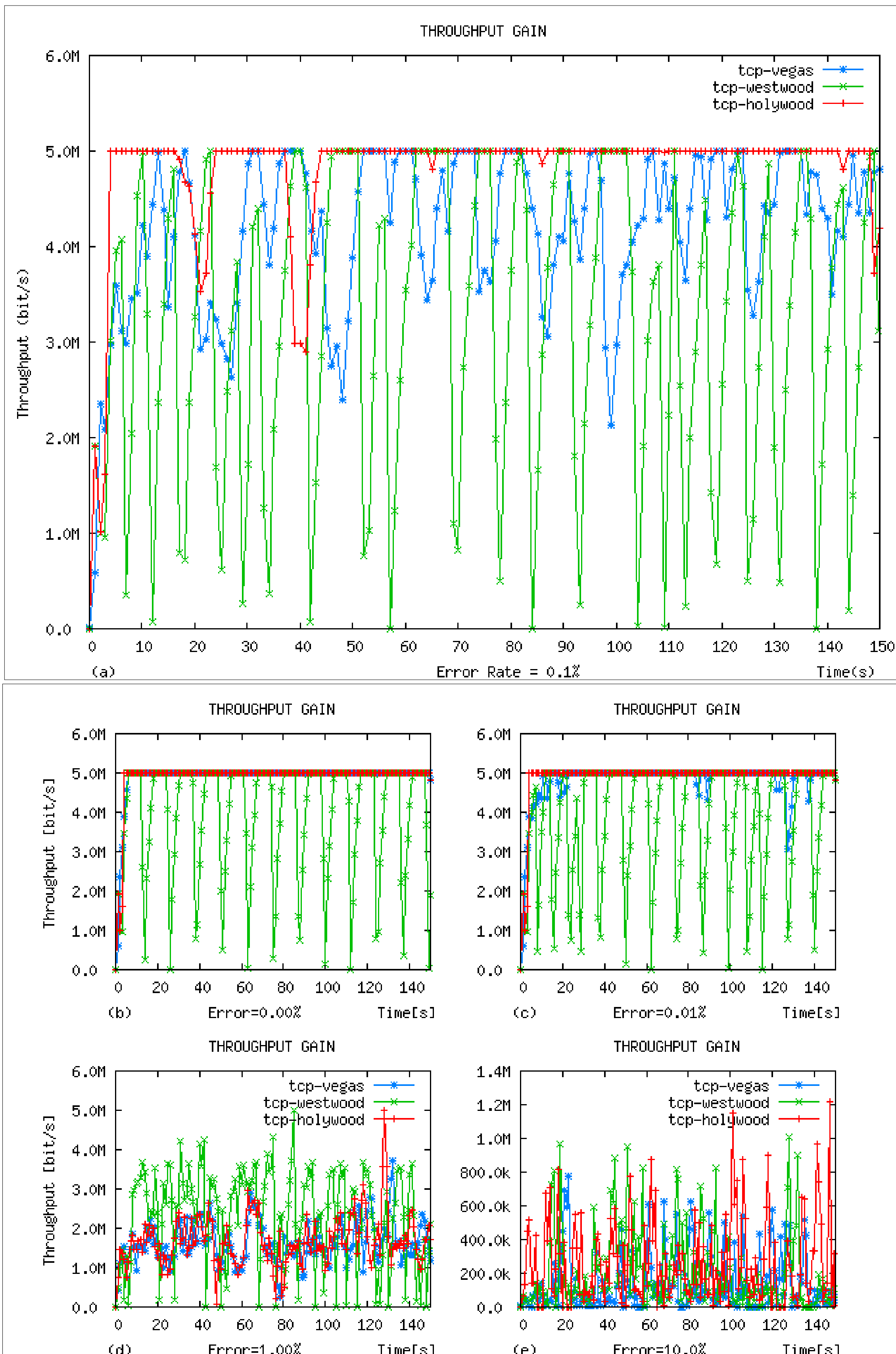

Figure 6.23: Throughput versus Time with different Error Rates over TCP HolyWood with other Protocols

Now we observe, if TCP HolyWood presented better average throughput performance or not against TCP Westwood and TCP Vegas. At a glance in figure 6.24, average throughput versus error rate, from an error rate of Approx. 0.0% to 0.6%, TCP HolyWood outperformed better than TCP Westwood and TCP Vegas; from error rate interval of 0.6% to 8%, TCP Westwood outperformed better nevertheless TCP HolyWood was as good as TCP Vegas. Finally, from the interval of 10% to 60% our proposal outperformed better, nevertheless TCP Westwood was a good as TCP Vegas.

In addition, calculating the appropriated, the coefficient of variation (COV) of TCP HolyWood and TCP Vegas were smaller than the TCP Westwood from Error Rate of 0% to 1%, TCP HolyWood and TCP Vegas presented similar Coefficient of variation. From 2.0% to 60.0% the coefficient of variation of TCP HolyWood and TCP Westwood in almost all the points was bigger than TCP Vegas. The exact percentage of COV and Confidence intervals of amongst TCPs are found in appendix A, Tables A.2.1.1 and A.2.1.2.

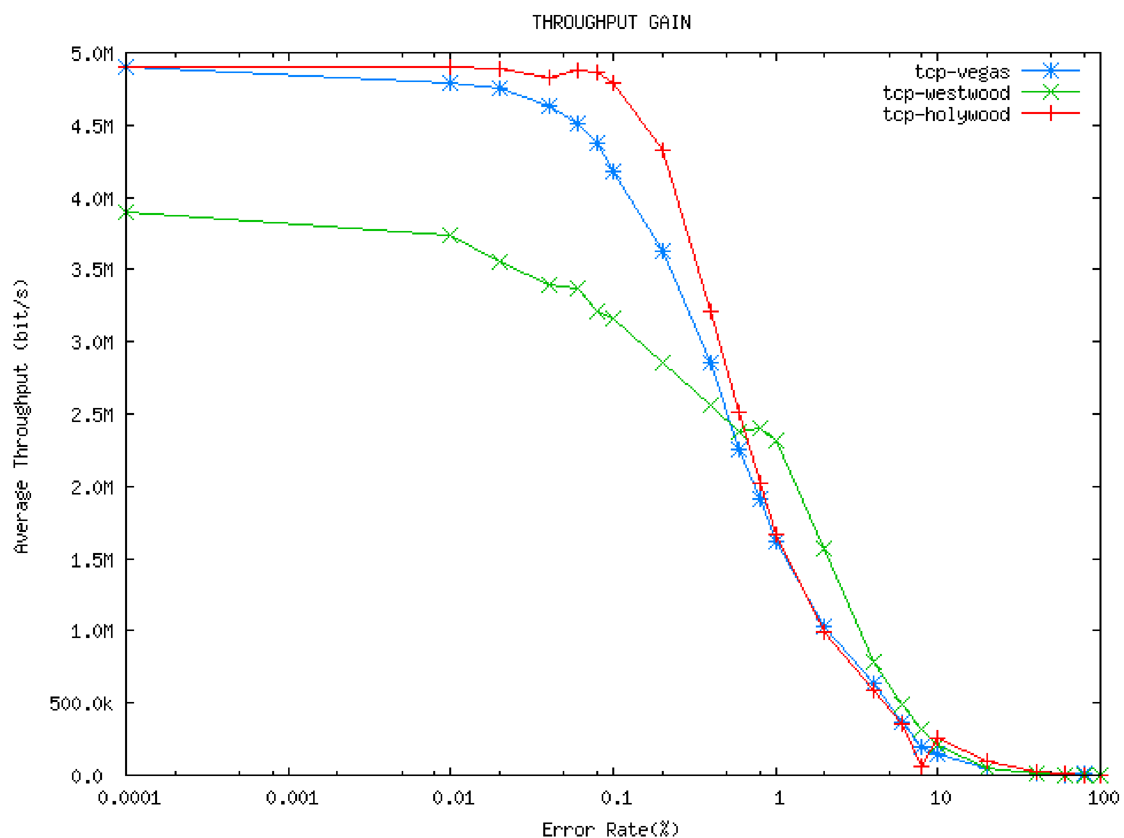

Figure 6.24: Average Throughput versus Error Rate of TCP HolyWood with other Protocols

Besides, in figure 6.25, throughput ratio versus error rate of TCP HolyWood with other Protocols, the results of this subsection in the search of the best throughput performance were as follows:

From 0% to 0.6%, TCP HolyWood presented 33.66% more throughput than TCP Westwood with latter as a base; and 8.57% more throughput than TCP Vegas as this last as a base. From 0.6% to 8%, TCP HolyWood presented 29.77% less throughput than TCP Westwood with latter as a base, and 9.34% less throughput than TCP Vegas as this last as a base. From 10% to 60%, TCP HolyWood presented 115.08% more throughput than TCP Westwood with latter as a base; and 335% more throughput than TCP Vegas

as this last as a base.

At last, in all the route of error rate axis, TCP HolyWood presented 30.65% more throughput than TCP Westwood with latter as a base; and 67.46% more throughput than TCP Vegas as this last as a base.

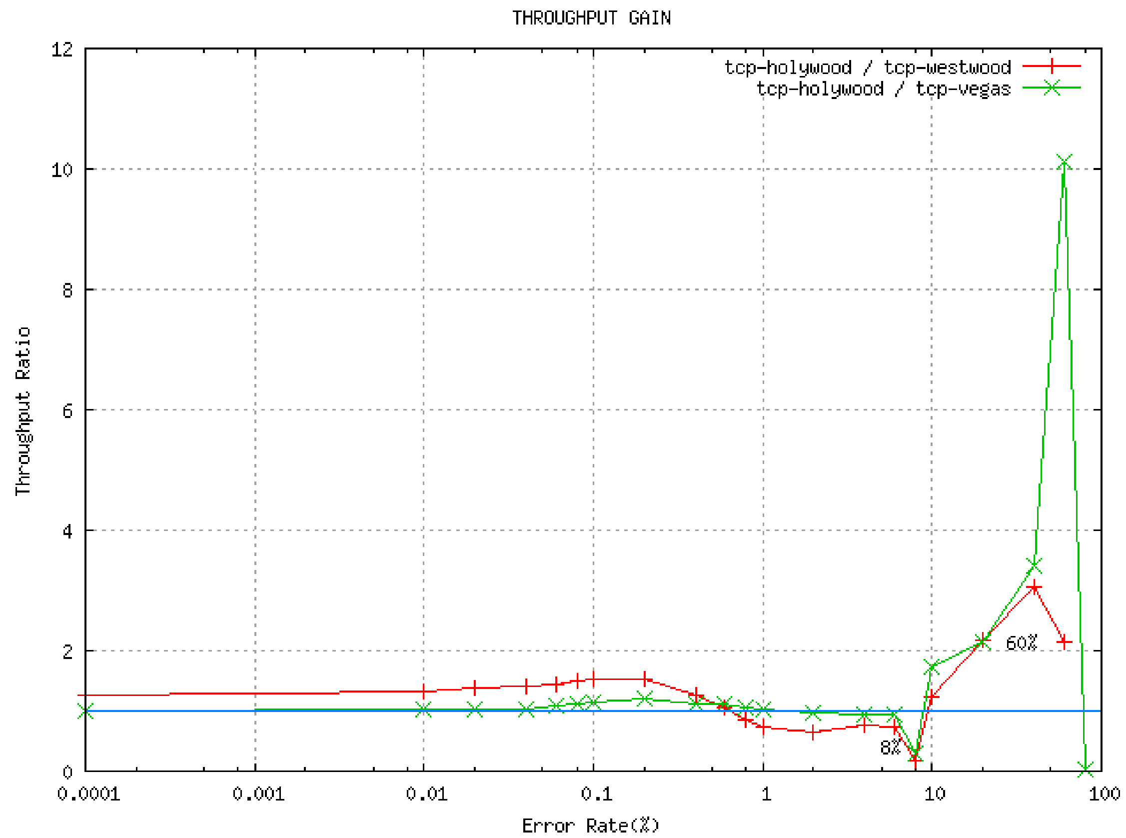

Figure 6.25: Throughput Ratio versus Error Rate of TCP HolyWood with other Protocols

6.4.2 Impact of Propagation Time on Throughput

In this subsection, each TCP was deployed each one at a time. We used an Error Rate of 0.1% and a bottleneck bandwidth of 5 Mbit/s in all the simulated experiments of this subsection. In figure 6.26.a, we worked with a propagation time of 50 ms and we discounted the first 15 seconds of the analysis of Figure 6.26.

In addition, in Figure 6.26.a, we observed that even before 15 seconds, approximately since second 5 of simulation due to warm up time or better explained that 5 seconds onwards a steady-state cyclic regime of TCP is attained. TCP was always in Congestion avoidance phase, but before 5 seconds approximately, we observed a transient behavior in which TCP was in the slow start phase. We also observed that our proposal presented better Throughput performance and with less variations than TCP Westwood and TCP Vegas. Besides, in figure 6.26.b (1 ms) and 6.26.c (5 ms), TCP HolyWood, together with TCP Vegas present an ideal performance using practically the maximum bottleneck bandwidth in the steady state of TCP Westwood displayed several falls in the same simulation decreasing its performance. In Figure 6.26.d (100 ms) TCP HolyWood presented better throughput performance and less variability in the the throughput than TCP Westwood and TCP Vegas.

Lastly, in figure 6.26.e (1000 ms) after second 40 presented better throughput performance and more stable in throughput than TCP Westwood and TCP Vegas. You may find in appendix C.2.2 the complete set of simulations we made about this subsection.

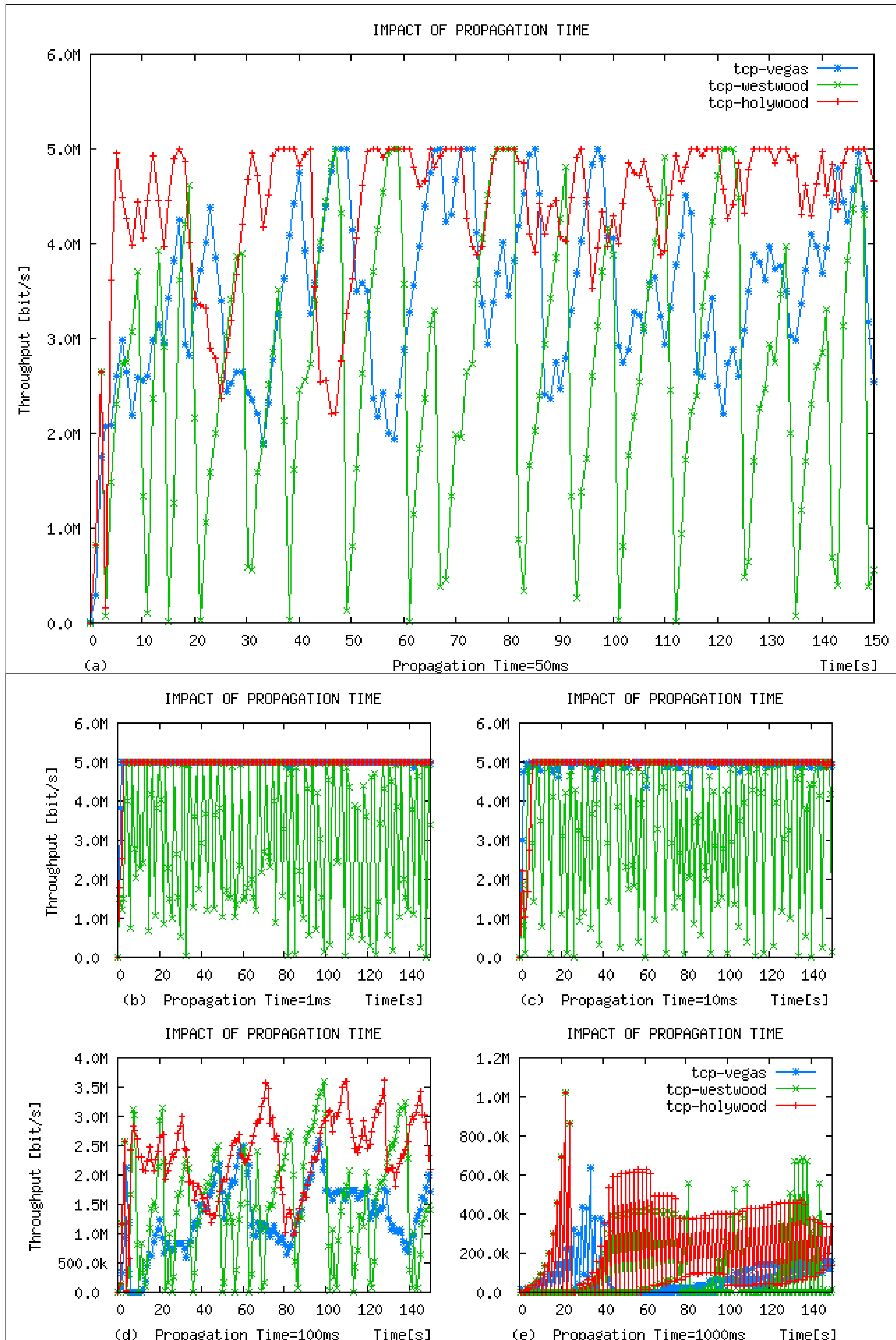

Figure 6.26: Throughput versus Time with Propagation Time of 50ms of TCP HolyWood with other Protocols

In Figure 6.27, average throughput versus propagation time of TCP HolyWood with other protocols, from 0.1 ms to 10 ms TCP HolyWood and TCP Vegas presented much better average throughput performance than TCP Westwood, being TCP Vegas a little greater than our proposal. Besides, in figure 6.27 also, from 10 ms to 1000 ms definitively, presented a better average throughput performance than TCP Westwood and TCP Vegas.

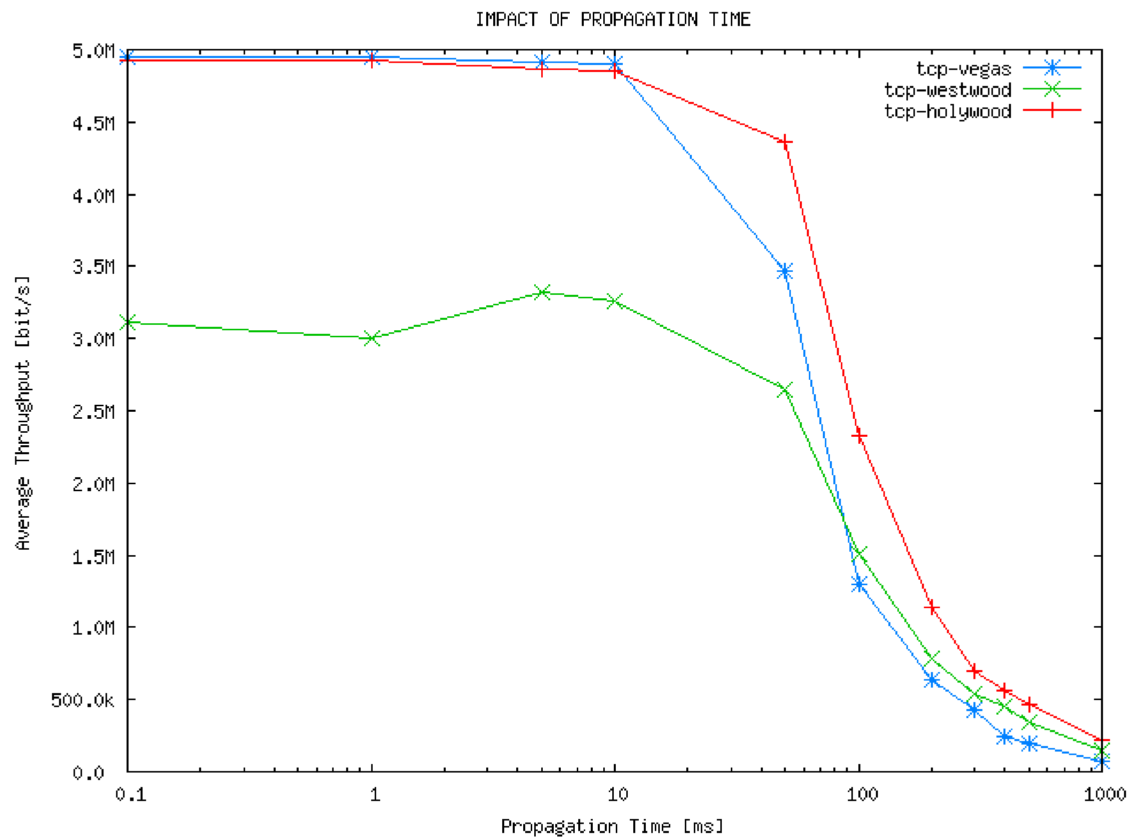

Figure 6.27: Average Throughput versus Propagation Time of TCP HolyWood with other Protocols

Now we observe exactly with what percentage was our proposal better or worse than the TCP Westwood and TCP Vegas. Moreover, in figure 6.28, the throughput ratio versus propagation time of TCP HolyWood with other protocols, from 0.1 ms to 10 ms, TCP HolyWood presented 54.45% more throughput performance than TCP Westwood, with latter as a base, and 0.77% more Throughput than TCP Vegas with the last as a base. Additionally, in figure 6.28, from 10 ms to 1000 ms, TCP HolyWood presented 44.55% more throughput performance than TCP Westwood, with latter as a base, and 91.59% more Throughput than TCP Vegas with the last as a base.

Finally, In all the route of propagation time axis, in Figure 6.28, TCP HolyWood presented 47.76% more throughput performance than TCP Westwood, with latter as a base, and 66.42% more Throughput than TCP Vegas with the last as a base.

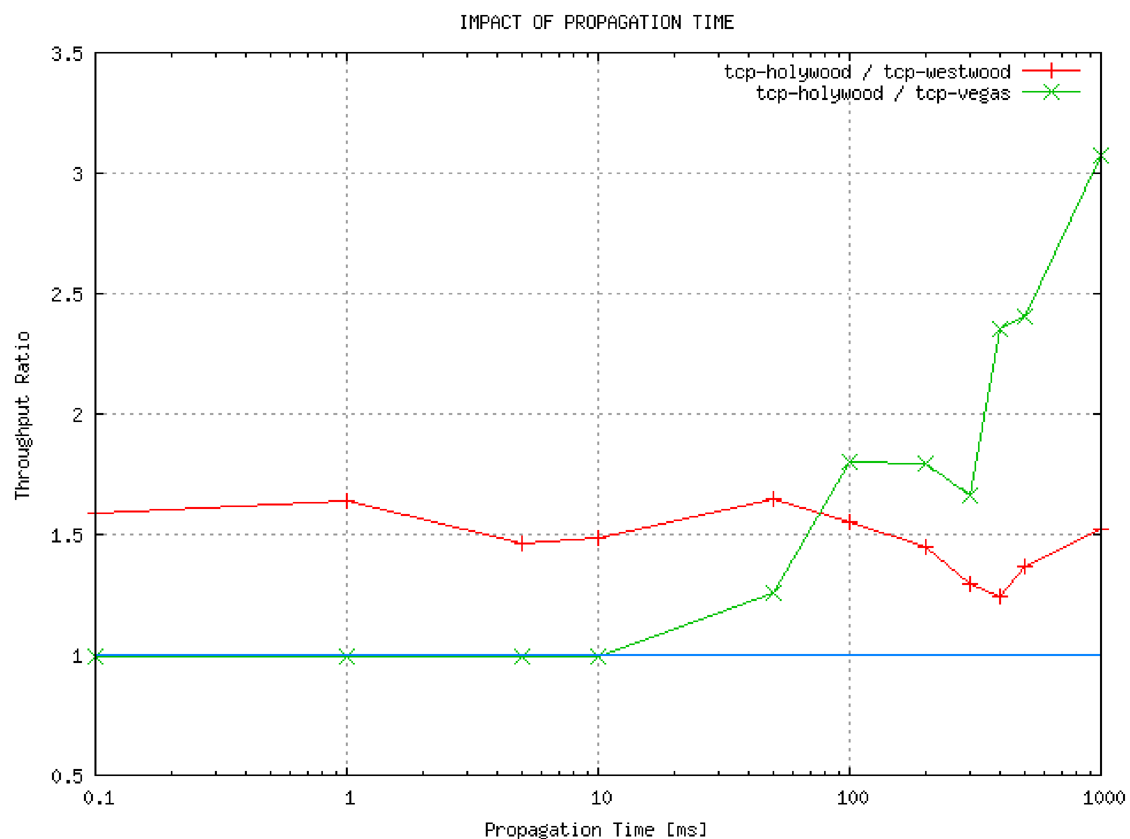

Figure 6.28: Throughput Ratio versus Propagation Time of TCP HolyWood with other Protocols

6.4.3 Impact of Bottleneck Bandwidth on Throughput of TCP Holywood, TCP Westwood, and TCP Vegas

In this subsection, we observe, what happening with the throughput of TCP HolyWood, TCP Westwood and TCP Vegas, when we varied the bottleneck bandwidth from 1 Mbit/s to 100 Mbit/s. In the figure 6.29, we presented the throughput versus time with different bottleneck bandwidth values; a different TCP was used in Topology 1, one at a time. In all the simulated experiments of this subsection, we used propagation time of 35 ms and an error rate of 0.1%. We discounted as before, the first 10% of the simulation time it means 15 Sec. from 150 Sec. as a rule of thumb due to warp up time (COUTINHO, 2003) (MACDOUGALL, 1987).

Additionally, in figure 6.29.a with a bottleneck bandwidth of 20 Mbit/s, we observed that TCP Westwood presented higher outliers of throughput than TCP HolyWood and TCP Vegas, nevertheless TCP HolyWood presented less variations of throughput performance than TCP Westwood and higher throughput than TCP Vegas.

In figure 6.29.b, with a bottleneck bandwidth of 1 Mbit/s after aproximately 15 seconds the three TCPs under test used the entire bandwidth available one each at a time. Besides, in Figures 6.29.c with of bottleneck bandwidth of 10 Mbit/s TCP HolyWood presented more throughputs and less variability of throughput than TCP Westwood and TCP Vegas. Additionally, in Figures 6.29.d and 6.29.e are similar to Figure 6.29.c with the difference that TCP Westwood presented 3 outliers of throughput in Figure 6.29.d and approximately 14 outliers of throughput in Figure 6.29.e.

Lastly, we observed that TCP HolyWood was higher in throughput performance than TCP Vegas in Figures 6.29.c, 6.29.d, and 6.29.e. A detailed list of all the experiments we made, may be found in the appendix C.2.3

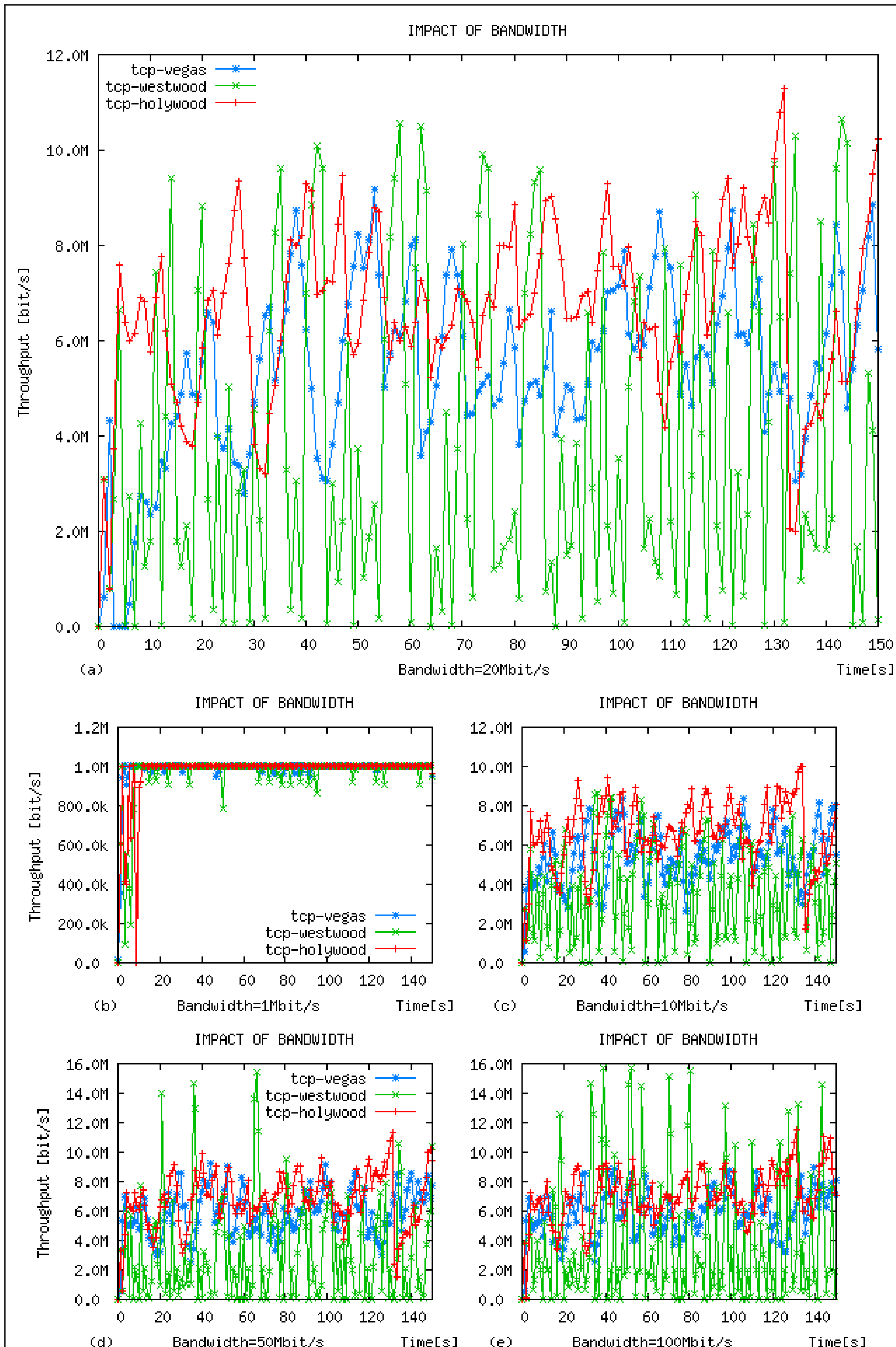

Figure 6.29: Throughput versus Time with different Bottleneck Bandwidth values of TCP HolyWood with other Protocols

In Figure 6.30, average throughput versus bottleneck bandwidth of TCP HolyWood with other protocols, we observed that the average throughput of TCP HolyWood was much better than the average throughput of TCP Westwood and TCP Vegas in all the route of bottleneck bandwidth axis. Additionally, in Figure 6.30, also, coefficient of variation of TCP HolyWood, in all most all the point, is smaller than TCP Westwood and TCP Vegas, we may find in appendix A. Table 2.3, as well as with the confidence intervals. In the next Figure 6.31 we will observe how much is that increase of throughput performance.

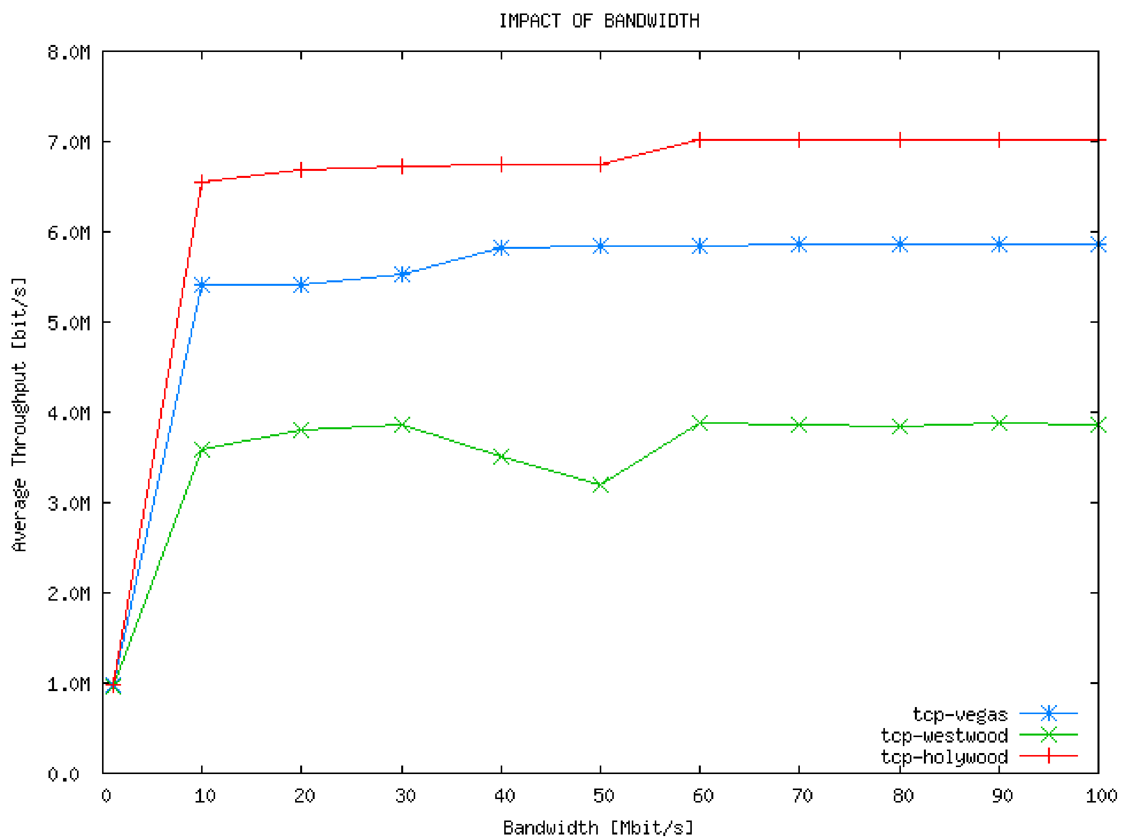

Figure 6.30 - Average Throughput versus Bottleneck Bandwidth of TCP HolyWood with other Protocols

In the Figure 6.31, throughput ratio versus bandwidth of TCP HolyWood with other protocols, we observed than the proportion TCP HolyWood to TCP Westwood was 40% higher than the proportion TCP HolyWood to TCP Vegas. A general measurement of all the point of bottleneck bandwidth in Figure 6.31, TCP HolyWood presented 76.7% higher throughput performance than TCP Westwood, with the latter as a base; and 17.71% higher than TCP Vegas with the last as a base. This analytical methodology was taken from Jain (1991, p.165-167).

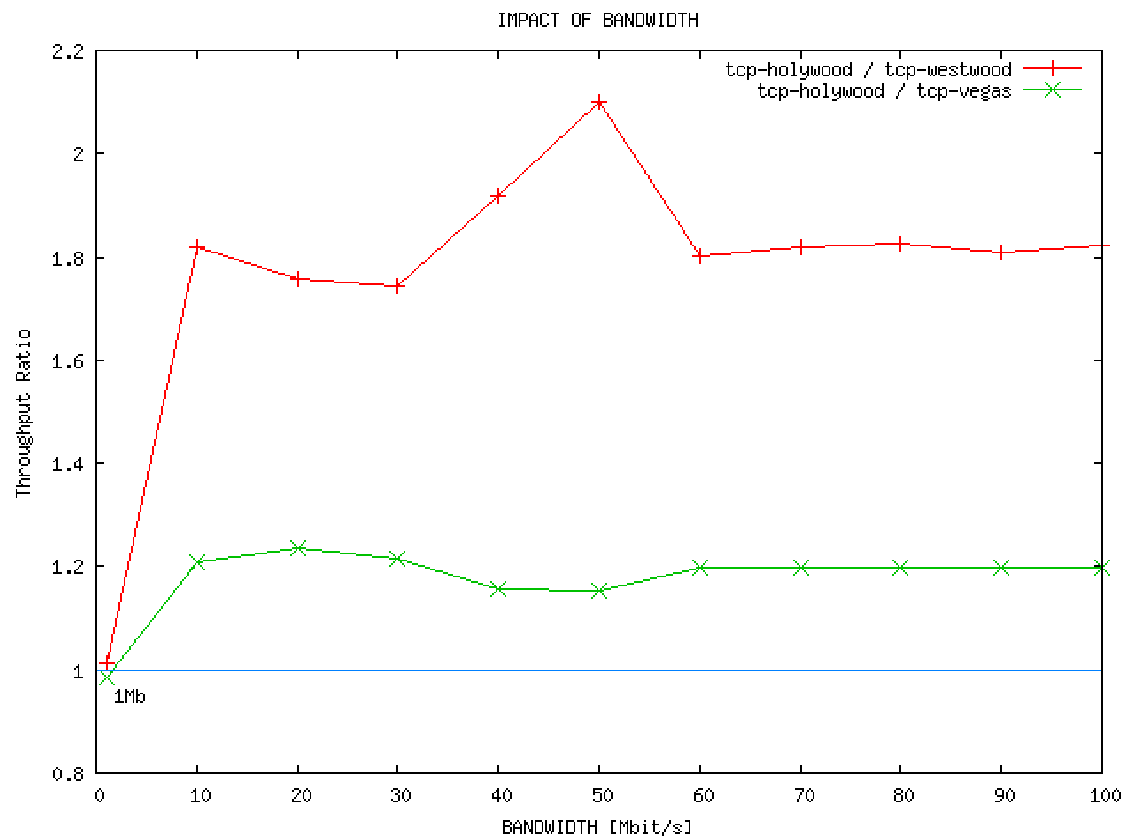

Figure 6.31: Throughput Ratio versus Bandwidth of TCP HolyWood with other Protocols

6.4.4 Impact of Error Rate on Jitter

Recapitulating, in this subsection we analyzed how the Jitter was affected by a varying and increasing error rate. When we worked with a jitter metric, what we hoped to find is a lower or smaller value and a smooth curve, much better if it was constant, because we might declare that the jitter is zero, meaning that we would not have variations of delays or expressed in other words constant packet latency.

Besides, in the figure 6.32, jitter versus sequence number with different error rates, TCP HolyWood with other protocols, a different TCP was used in Topology 1, one at a time. In all the simulated experiments of this subsection, we used Propagation Time of 35 ms and a bottleneck bandwidth of 5 Mbit/s. We started to analyze Figure 6.32, since a sequence number of 500 or 0.5 K due to that approximately after this values the system became stable.

In Figure 6.32.a with error rate of 0.1%, we observed that TCP HolyWood, red colored, presents less jitter than TCP Westwood and TCP Vegas;TCP Westwood presented the highest outliers of the three protocols (3); and our proposal presented the highest sequence number (approx. 65) followed closely by TCP Vegas (Approx. 56 K.) and after it was TCP Westwood (Approx. 40 K.). In Figure 6.32.b, with error rate of 0.0% and 6.32.c, with error rate of 0.01%, TCP HolyWood outperformed better than TCP Westwood and TCP Vegas showing a linear shape figure closest to zero.

Additionally, in Figure 6.32.d, with error rate of 1% TCP Westwood presented highest sequence number (30 K.) of the three TCPs under test, nevertheless the highest outliers as well. On the, other hand TCP HolyWood and TCP Vegas, presented similar jitter and just the same maximum sequence number (approx. 22 K.). Finally, in Figure, 6.3.2.e the three TCPs presented high jitter, but TCP Vegas presented smaller outliers than the other TCPs and the least sequence number (aprox 1.8 K.) in comparison to a sequence number aproximately of 2.8K. As appreciated in apendix C.2.4.

Mostly, we observed that, the three TCPs under test when the error rate increased also increased the jitter together with the outliers. Moreover, the sequence number diminished signicantly. You may find in appendix C.2.4 the complete set of simulations we accomplished about this subsection.

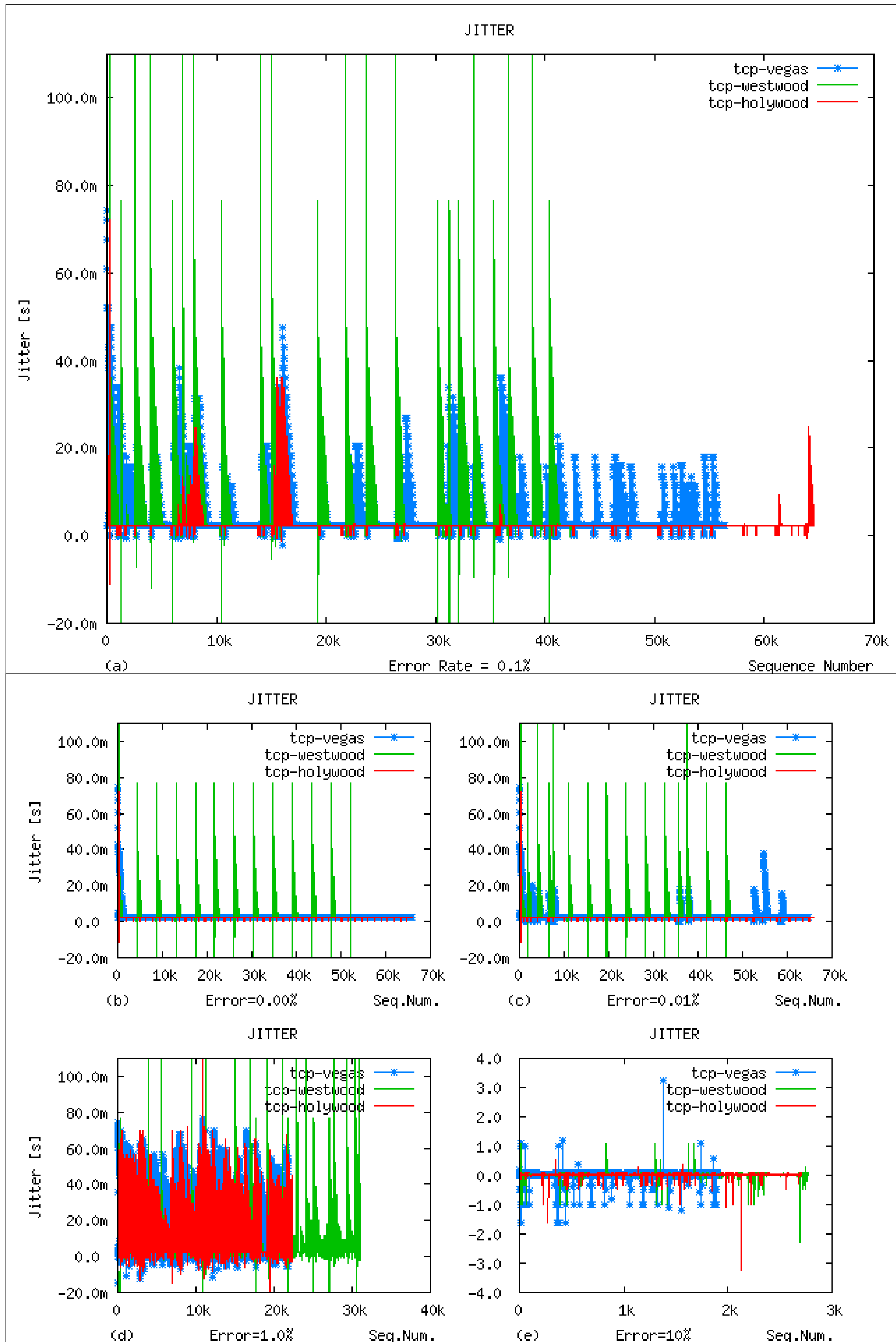

Figure 6.32: Jitter Versus Sequence Number with different Error Rates TCP HolyWood with other Protocols

Following with the analysis of this subsection, in the figure 6.33, we extracted this information from appendix A, Tables A.2.4.1, and A.2.4.2, to get the Table A.2.4.3 that was based in Jain (1991, p.167), and that uses always TCP HolyWood as a base.

Additionally, in figure 6.33, average jitter versus error rate of TCP HolyWood with other protocols, we observed that the error rate interval from 0% to 0.4%, TCP Westwood presented 25.84% more jitter than TCP HolyWood, with latter as a base; and TCP Vegas presented 8.89% more jitter than TCP HolyWood, with TCP HolyWood as a base. In the error rate interval from 0.4% to 10%, TCP Westwood presented 41.63% less jitter than TCP HolyWood, and TCP Vegas presented 12.87% less jitter than TCP HolyWood. In the error rate interval from 10% to 60%, TCP Westwood presented 14.48% more jitter than TCP HolyWood, and TCP Vegas presented 1300.14% more jitter than TCP HolyWood. Finally, for values, greater than 20% of error rate the jitter was extremely high in comparison with the other values of the curve of Figure 6.33.

In all the route of error rate axis from 0% to 60%, TCP Westwood presented 0.11% less jitter than TCP HolyWood, with latter as a base, and TCP Vegas presented 265.32% more jitter than TCP HolyWood, with latter as a base. However, this value is hardly influence by the error rate higher than 20%. **Therefore, we thought more apropriate to finish this analysis with a shorter route of error rate axis from 0.0% to 20% due to the higher jitter outlier from all the TCPs in highly noisy link. In this context, TCP Westwood presented 5.23% more jitter than TCP HolyWood, and TCP Vegas presented 12.67% more jitter than TCP HolyWood, showing us that our proposal outperformed better.**

Due to the presence of negative jitter in the average values of both TCPs and because we were using ratios, we calculate the absolute value of the ratio TCP Reno to HolyWood. So in this way, we would present higher those values that would be far away from zero being them positive or negatives and and would also present lower those values otherwise. Additionally, in figure 6.34.a and 6.34.b, we observed that the Coefficient of Variation of TCP HolyWood is generally smaller than TCP Westwood and TCP Vegas and presented a linear shaped figure with the exception of one point in error rate of 0.04%. A list of the COV of the three TCP together with 95% confidence intervals is in appredix A, Tables A.2.4.1 and A.2.4.2

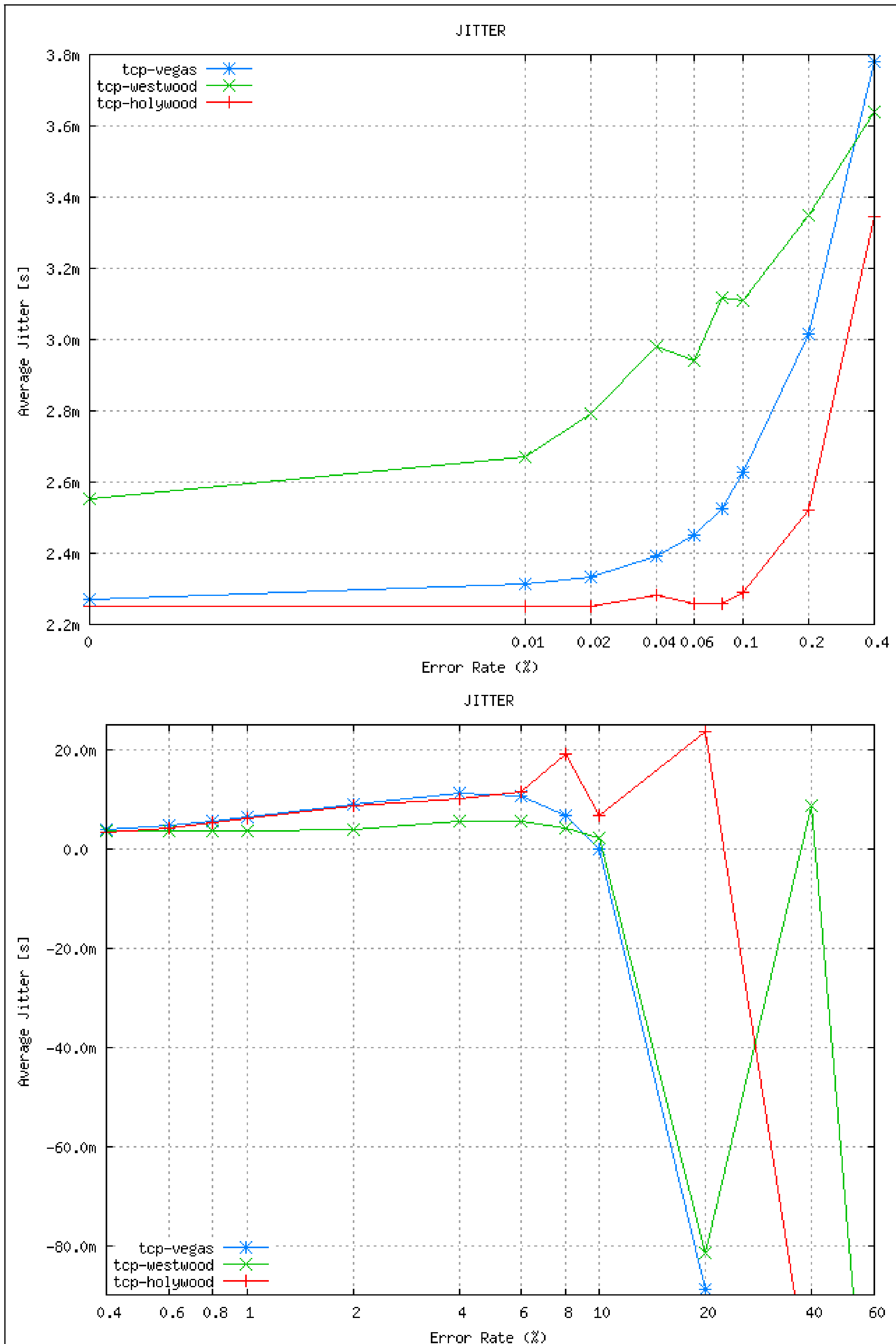

Figure 6.33: Average Jitter versus Error Rate of TCP HolyWood with other Protocols

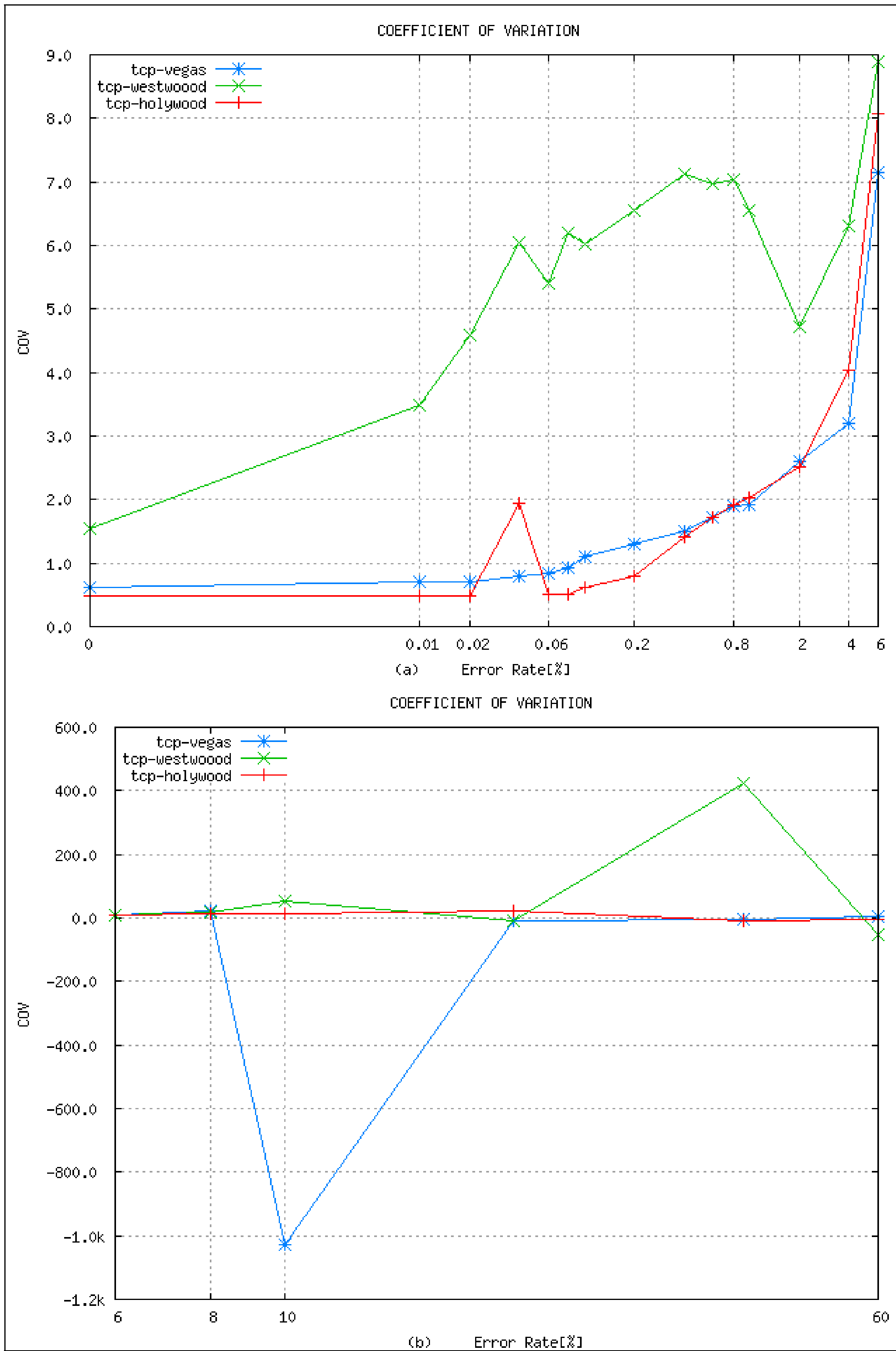

Figure 6.34: COV of the Average Jitter versus Error Rate of TCP HolyWood with Other Protocols

6.4.5 Impact of Propagation Time on Jitter

As we may observe, in Figure 6.35.a, jitter versus sequence number with different propagation time values of TCP HolyWood with other protocols, with an error rate of 0.1%, a bottleneck bandwidth of 5 Mbit/s, and a propagation time of 50 ms, the TCP HolyWood presented lower jitter than TCP Westwood and TCP Vegas as well as produced a higher sequence number. In addition, TCP Vegas outperformed better in jitter than TCP Westwood. The jitter in all the cases were rough due to great number of positive jitter outliers, especially in TCP Westwood

Besides, in figure 6.35.b (1 ms), TCP HolyWood and Vegas outperformed better in jitter than TCP Westwood, presenting both TCPs, a maximum sequence number of Approx. 68 K. and TCP Westwood a maximum sequence number of 39 K. TCP Westwood also displayed several positive outliers. Whereas that the figure 6.35.c (10 ms.) was quite similar to Figure 6.35.b with the difference that TCP Westwood presented several small positive and negative outlier in all its existence. At last, in figure 6.35.d (100 ms) and figure 6.35.e (1000 ms) there was an irregular variation of the jitter of the three TCPs under test, being TCP Vegas and our proposal smaller in jitter than TCP Westwood. TCP Westwood again presented higher outliers, and TCP HolyWood displayed the highest value of sequence number.

According to the increase of the propagation time as we observed in Figure 6.35.b, 6.35.c, 6.35.d, and 6.35.e the jitter increased as well and the maximum sequence number of each TCP diminishes. As always, you may find in appendix C.2.5 the complete set of simulations we made about this subsection.

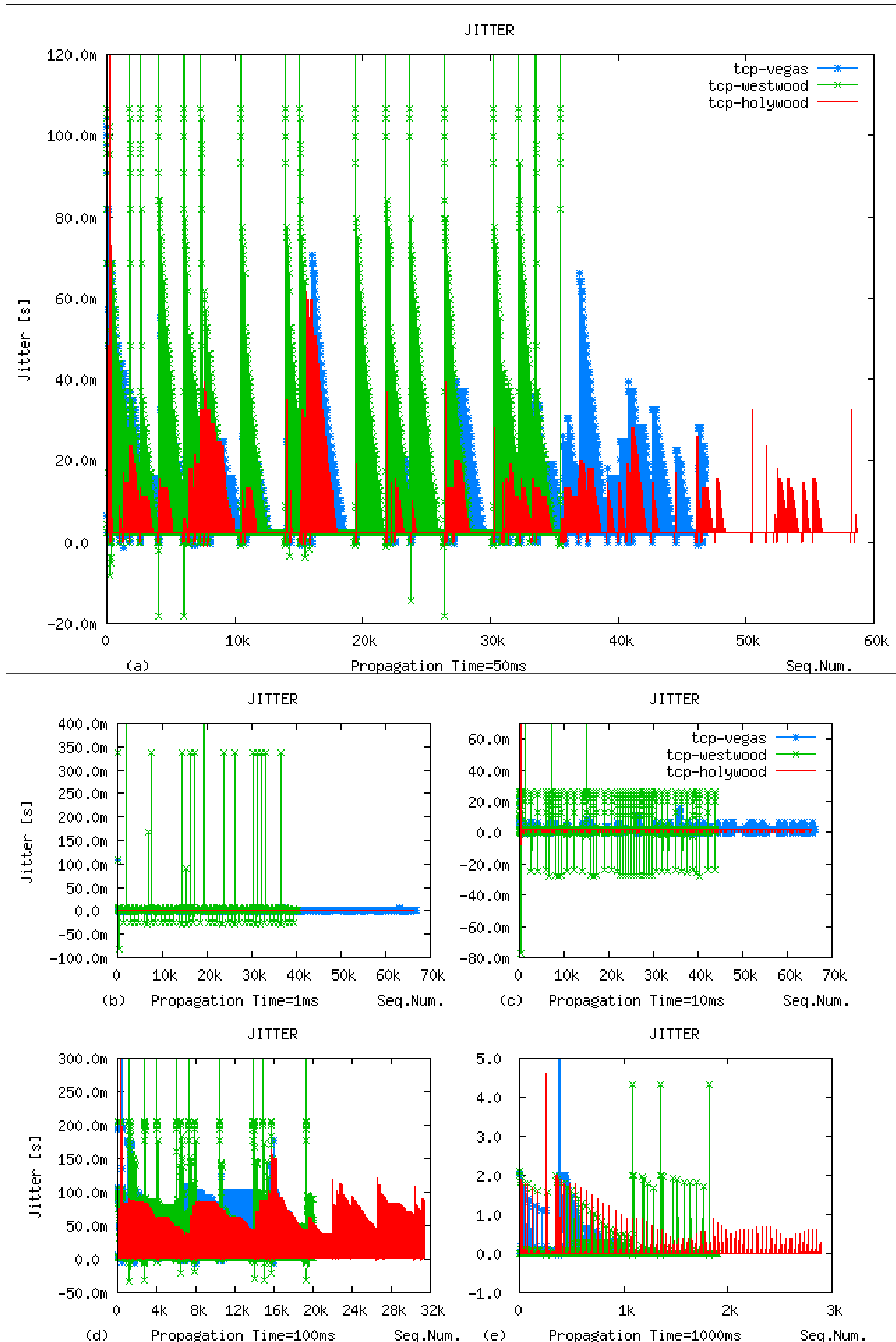

Figure 6.35: Jitter versus Sequence Number with Different Propagation Time Values of TCP HolyWood with other Protocols

Now we calculate, how much bigger or lower was the jitter of TCP HolyWood against the TCP Reno. As we may observe, in figure 6.36, average jitter versus propagation time, in the entire propagation time axis TCP HolyWood showed a lesser jitter than TCP Reno. From the interval of 0.1 ms to 50 ms, with TCP HolyWood as a base, TCP Reno showed 14.84% more jitter than TCP HolyWood. Then from the interval of 0.1 ms to 50 ms, with TCP HolyWood as a base, TCP Westwood showed 16.34% more jitter than TCP HolyWood and TCP Vegas showed 5.46% more jitter than TCP HolyWood too. Finally, from the interval of 50 ms to 1000 ms, with TCP HolyWood as a base, TCP Westwood showed 17.9% more jitter than TCP HolyWood and TCP Vegas showed 100.62% more jitter than TCP HolyWood too.

In general, in all the route of propagation time, with TCP HolyWood as a base, TCP Westwood showed 13.6% more jitter than TCP HolyWood. In addition, TCP Vegas showed 64.17% more jitter than TCP HolyWood; with TCP HolyWood in both cases as a base. It meant that TCP HolyWood outperformed better in jitter than TCP Westwood and TCP Vegas with a lower value. As before, to obtain this percentages we extracted these information, after a proper processing, based in (JAIN 1991, p167), from appendix A, Tables A.2.5.1, and A.2.5.2

At last, analyzing the statistics, in figure 6.37, the coefficient of variation (COV) of average jitter versus propagation time of TCP HolyWood with other Protocols, from the inteval of 0.1 ms to 50 ms, presented slightly bigger value than TCP Vegas. Moreover, both of them were smaller than TCP Westwood. Additionally in the Figure 6.37, also from the interval of 50 ms to 1000 ms, TCP HolyWood presented the smallest coefficient of variation. A list of the values of the coefficient of variation, as well as, the Confidence intervals at 95% are found in appendix A, Table A.2.5.1

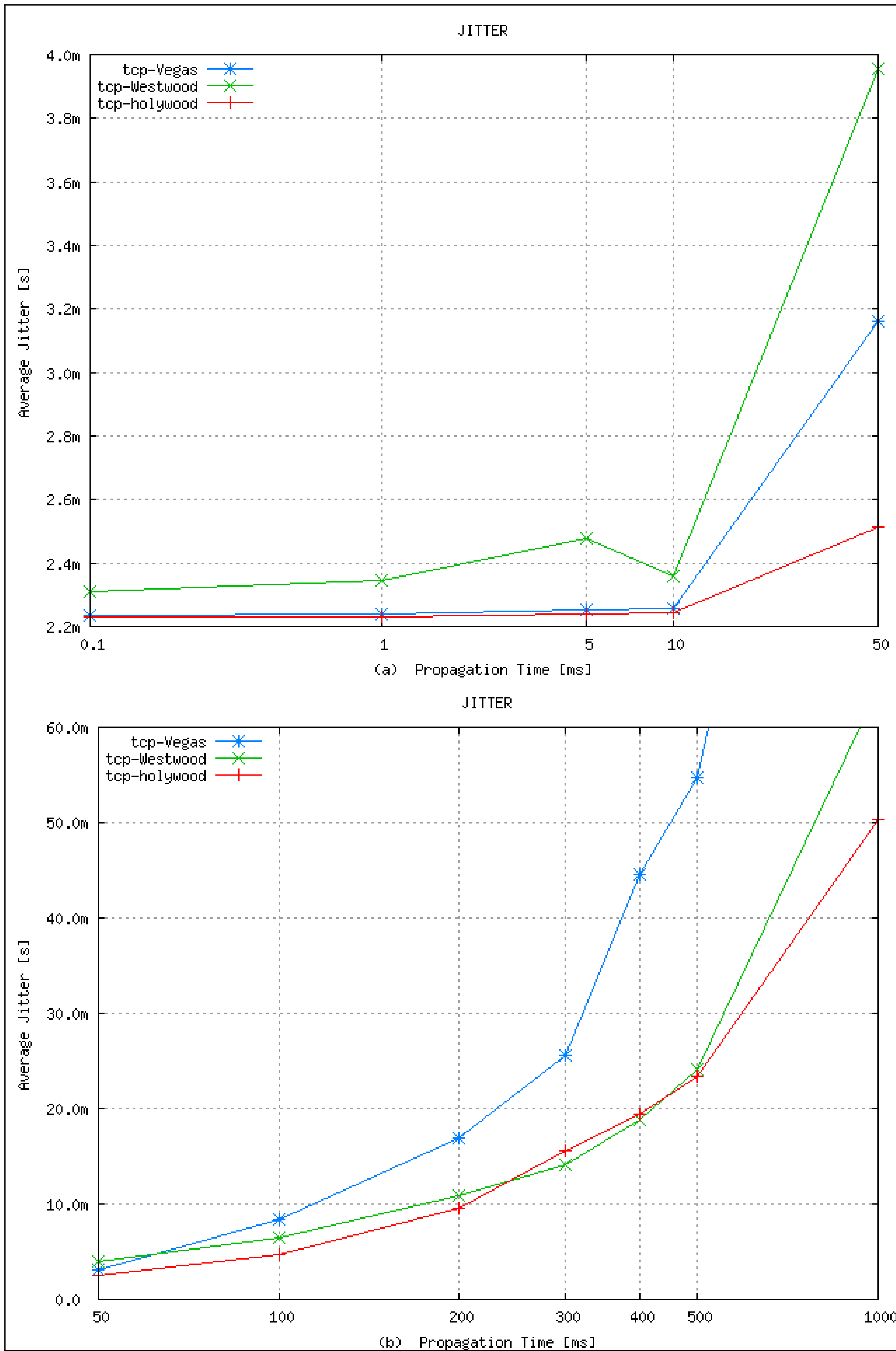

Figure 6.36: Average Jitter versus Propagation Time of TCP HolyWood with other Protocols

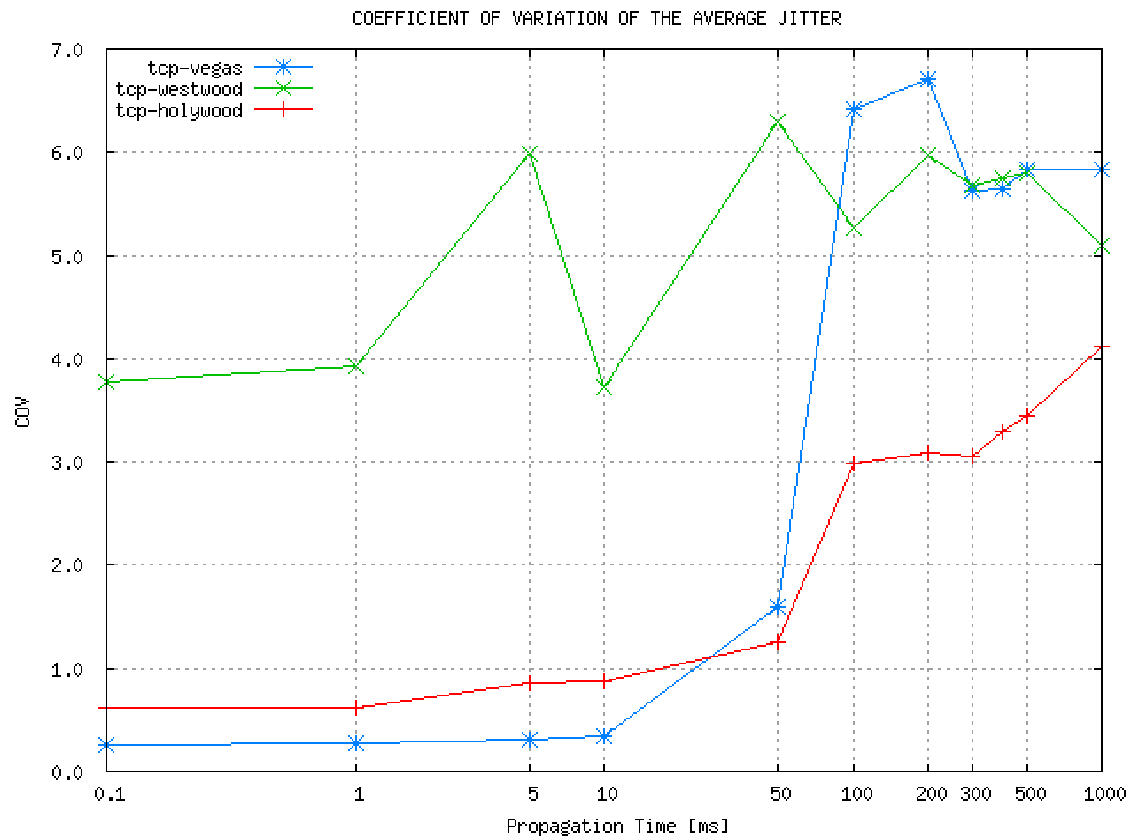

Figure 6.37: COV of Average Jitter versus Propagation Time of TCP HolyWood with other Protocols

6.4.6 Impact of Bottleneck Bandwidth on Jitter

In Figure 6.38.a, jitter versus sequence number with different bottleneck bandwidth values of TCP HolyWood with other Protocol, we did not consider the first 500 sequence numbers due to warm up time, with error rate of 0.1%, propagation time of 35 ms, and bottleneck bandwidth 20 Mbit/s. It was used topology 1, and each TCP one at a time, we observed, also, that TCP HolyWood presented slightly less jitter than TCP Vegas and TCP Westwood, nevertheless TCP Westwood presented several positive and negative outliers. TCP HolyWood showed also the highest value of Sequence number, followed closely by TCP Vegas and after TCP Vegas, TCP Westwood.

Besides, the jitter of TCP HolyWood in Figure 6.38.b (1 Mbit/s) and TCP Westwood and TCP Vegas were stabilized approximately in 11ms, nevertheless TCP Westwood and TCP Vegas presented several positive outliers of aproximately 18ms. Additionally, the figure, 6.38.c (10 Mbit/s) was similar to Figure 6.38.a (20 Mbit/s). Figure 6.38.d (50 Mbit/s) and 6.38.e (100 Mbit/s) were quite similar, whereas in this figures the three TCPs presented jitter of aproximatelly of 65ms. Nevertheless our proposal had the highest value of sequence number (91 K.) followed for TCP Vegas with 80 K., and after it TCP Westwood with approximately 40 K. for Figure 6.38.d and 48 K. for Figure 6.38.e. As always, you may find in appendix C.2.6 the complete set of simulations we made about this subsection.

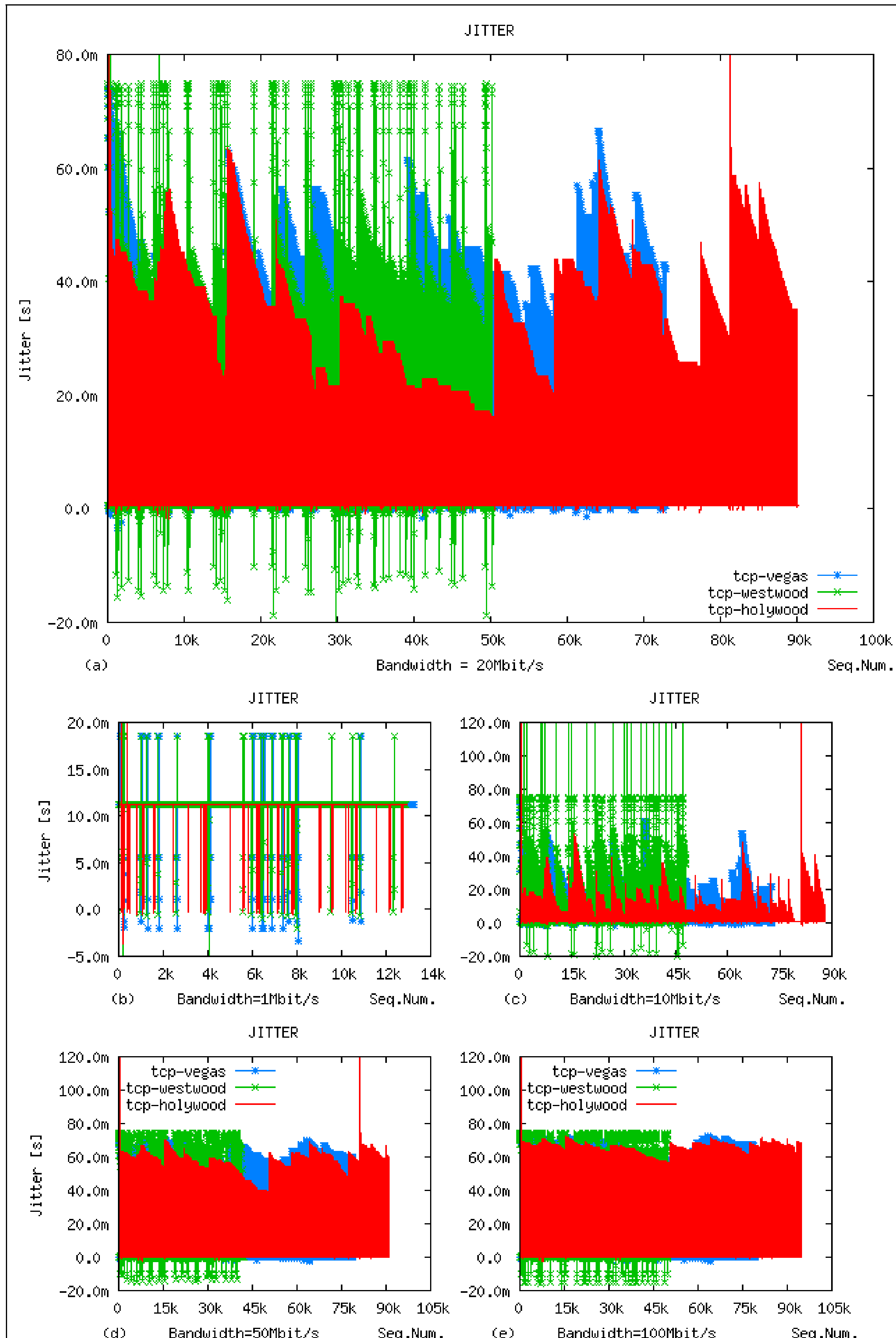

Figure 6.38: Jitter Versus Sequence Number with different Bandwidth values TCP HolyWood with other Protocol

Now we calculate how much was the jitter of TCP HolyWood against the TCP Westwood and TCP Vegas. As we observed, in Figure 6.39, average jitter versus bottleneck bandwidth, in the entire propagation time axis TCP HolyWood showed a lesser jitter than TCP Vegas and a higher jitter than TCP Westwood.

Besides, from the interval of 1 Mbit/s to 10 Mbit/s, with TCP HolyWood as a base, TCP Westwood showed 29.12% more jitter than TCP HolyWood, and TCP Vegas showed 10.52% more jitter than TCP HolyWood. Finally, from the interval of 10 Mbit/s to 100 Mbit/s, with TCP HolyWood as a base, TCP Westwood showed 9.98% less jitter than TCP HolyWood, and TCP Vegas showed 19.29% more jitter than TCP HolyWood too.

In general, in all the route of propagation time, with TCP HolyWood as a base, TCP Westwood showed 8.95% less jitter than TCP HolyWood and TCP Vegas also showed 17.55% more jitter than TCP HolyWood. As before, to obtain this percentages extracting these information, after a proper processing based in (JAIN 1991, p167) from appendix A, Tables A.2.6.1, and A.2.6.2.

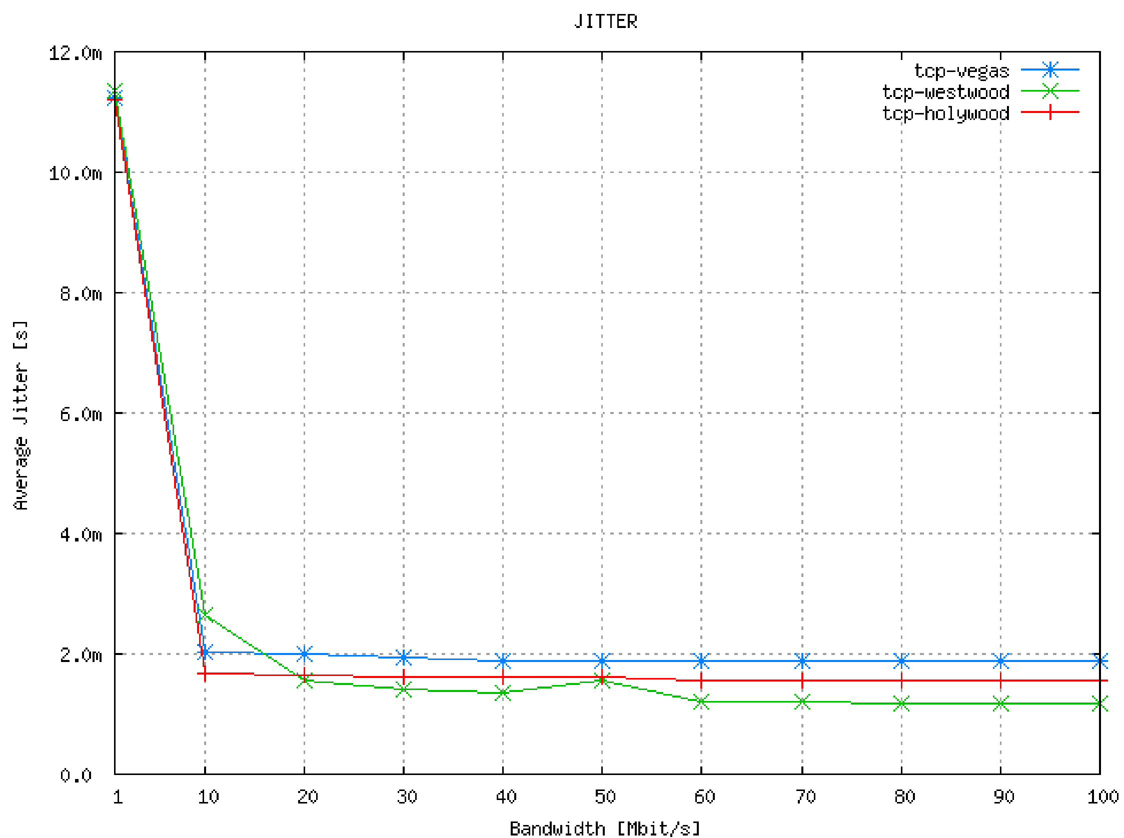

Figure 6.39: Average Jitter versus Bandwidth of TCP HolyWood with other Protocols

Finally, to close this subsection, in figure 6.40, in all the points TCP HolyWood presented a lower coefficient of variation than TCP Westwood and in almost all the points, the coefficient of variation of TCP HolyWood was higher than TCP Vegas. In other words, the Coefficient of Variation of TCP HolyWood was between TCP Westwood and TCP Vegas. A list of the values of the coefficient of variation of this subsection, as well as, the confidence intervals at 95% are found in appendix A, Table A.2.6.1.

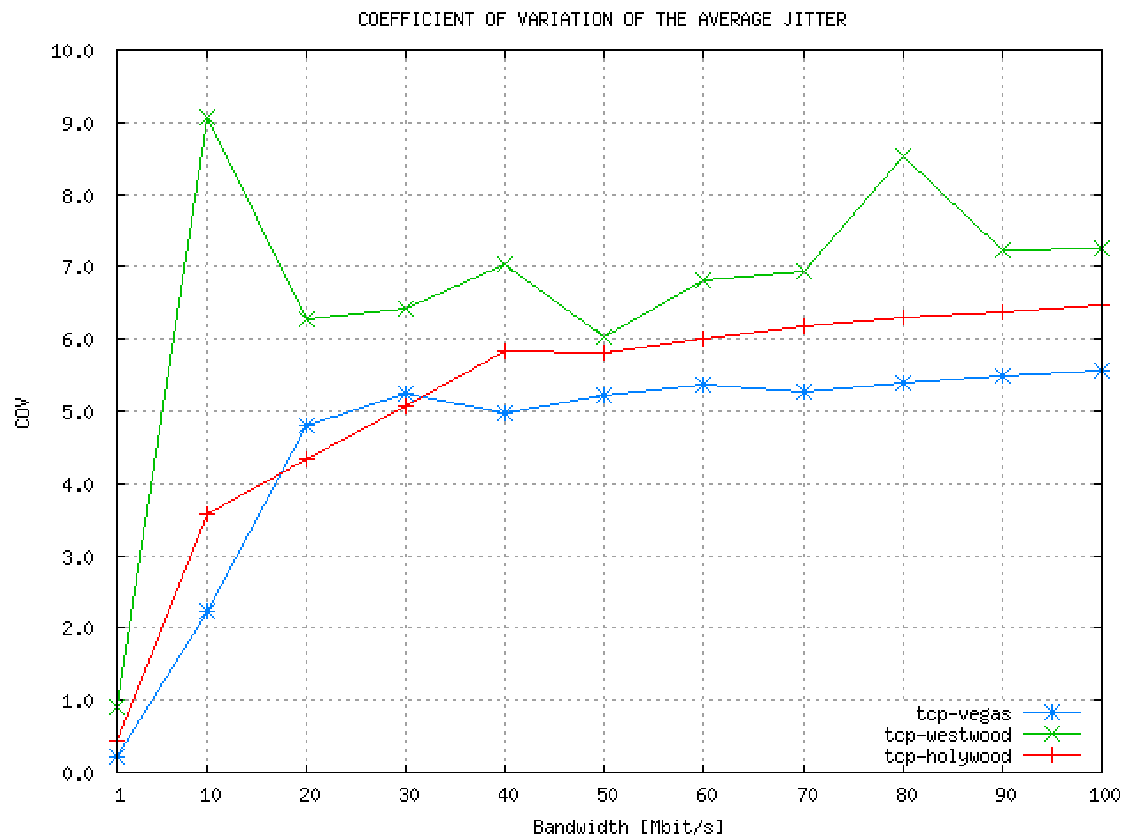

Figure 6.40: COV of the Average Jitter versus Bandwidth of TCP HolyWood with other Protocols

6.4.7 Percentage of lost packets

In Figure 6.41.a, percentage of lost packets of TCP HolyWood versus TCP Westwood and TCP Vegas, from an error rate interval of 0.0% to 1% and TCP HolyWood as a base; TCP Westwood presented 100.06% more Lost packets than TCP HolyWood and TCP Vegas presented 32.71% less lost packets Than TCP HolyWood.

Besides, in figure 6.41.b, from an error rate interval of 1% to 60% and TCP HolyWood as a base, TCP WestWood presented 18.97% less lost packets than TCP HolyWood, and TCP Vegas presented 14.39% less lost packets than TCP HolyWood.

Finally, in all the route of Error Rate axis of figure 6.41 with TCP HolyWood as a base, TCP Westwood presented 57.21% more lost packets than TCP HolyWood and TCP Vegas Presented 25.47% less lost packets than TCP HolyWood.

In other words in figure 6.4.1 TCP HolyWood showed less percentage of lost packets than TCP Westwood and more percentage of lost packets than TCP Vegas in all the route of error rate axis. It is important to remark that TCPs sent different quantity of packets for the same simulation. We extracted this information from Trace graph, the window of network information, and one point for each error rate value. After we arranged the data and processed it as shown in appendix A.3.2 Tables A.3.2.1, A.3.2.2 and for finding the final comparison percentages of TCP HolyWood and TCP Reno we used the methodology of Jain (1991, p.165-167).

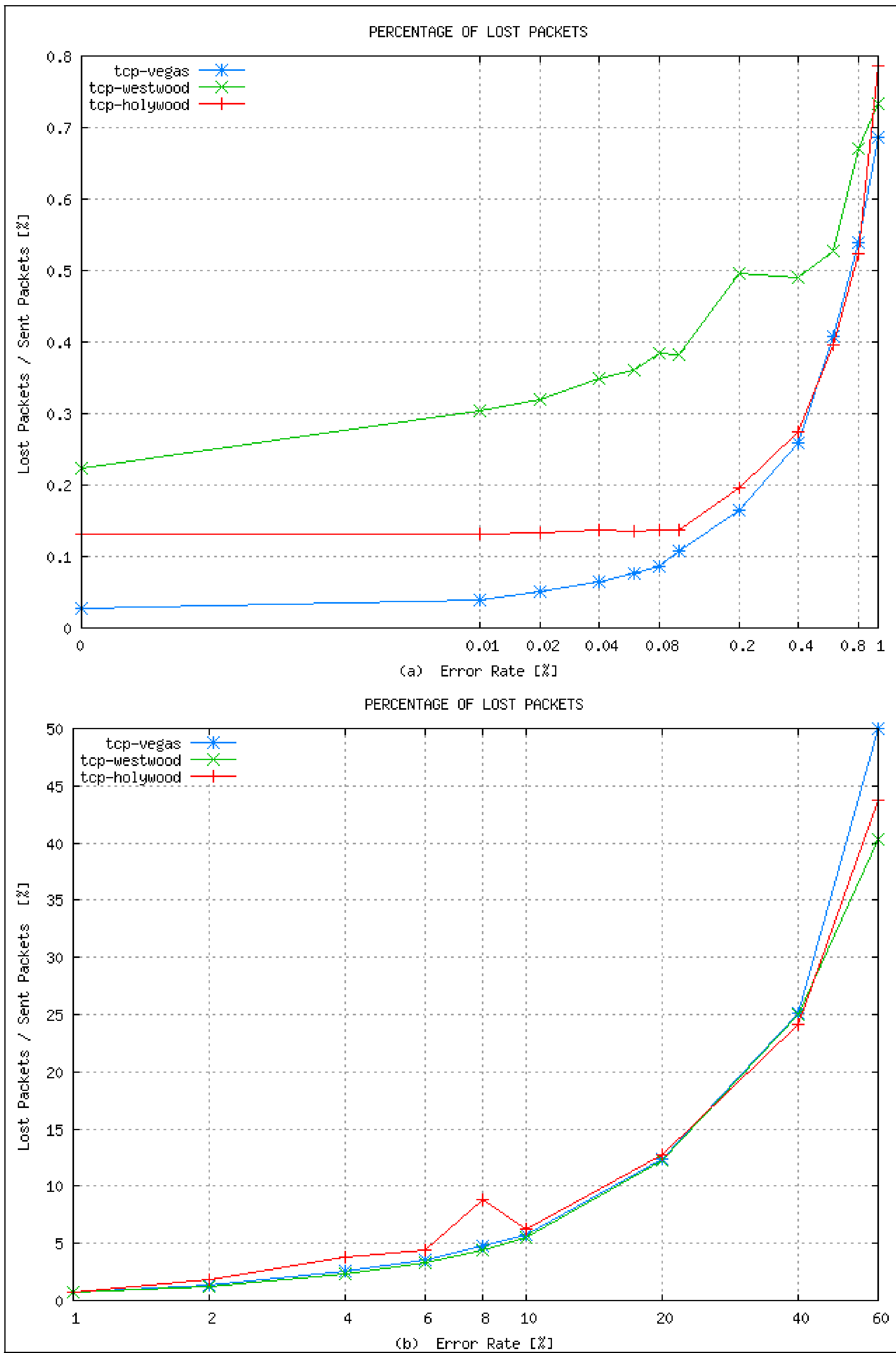

Figure 6.41: Percentage of Lost Packets of TCP HolyWood versus TCP Westwood and TCP Vegas

6.4.8 Latency

To find the latency of TCP HolyWood with TCP Westwood and TCP Vegas, we tested a minimum, medium, and a maximum value of error rate. In our chosen topology, the results were as follows:

Table 6.2: Statistics of Latency of TCP Vegas, TCP Westwood, and TCP HolyWood

Error Rate [%]	Protocol [TCP]	Average Latency [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
0	Vegas	42.06	3.662	8.707	41.474	42.646
	Westwood	57.953	24.274	41.885	54.069	61.838
	HolyWood	75.74	28.936	38.204	71.109	80.37
0.1	Vegas	40.465	3.022	7.469	39.982	40.949
	Westwood	47.644	15.904	33.379	45.099	50.19
	HolyWood	56.519	20.149	35.652	53.294	59.744
10	Vegas	39.347	2.337	5.94	38.973	39.721
	Westwood	40.658	69.256	170.336	29.575	51.742
	HolyWood	41.854	10.798	25.799	40.126	43.583

Even here, in this subsection dealing with other protocols (TCP Vegas and TCP Westwood) Jain (1991) was right. In all the tests, we made among TCP Holywood versus TCP Vegas and TCP Westwood; our proposal had the higher average latency or packet delay as shown in the table with a difference respect of TCP Vegas of Approx. 33.7 ms, 16.1 ms and 2.5 ms; and respect to TCP Westwood of Approx. 17.8 ms, 8.9 ms 1.2 ms with error rates of 0.0 %, 0.1% and 10% respectively. The same observation we obtained here with TCP Reno, the difference was diminishing while the error rate of the link increased. Finally, TCP Westwood and TCP HolyWood had relatively higher coefficient of variation of the average data in comparison with TCP Vegas.

6.4.9 Fairness and Friendliness

We did not do the fairness and friendliness of TCP HolyWood with TCP Westwood and TCP Vegas, simple because they are not the default standard, even though they are excellent proposals. Maybe in the future, their usability will be extended; we do not know, meanwhile, TCP Reno and its enhancements called TCP New Reno and TCP SACK are widely deployed nowadays.

7 Conclusion and Future Works

"Dear children! Today I call you to be my extended hands in this world that puts God in the last place. You, little children, put God in the first place in your life. God will bless you and give you strength to bear witness to Him, the God of love and peace. I am with you and intercede for all of you. Little children, do not forget that I love you with a tender love. Thank you for having responded to my call." February 25, 2005. Bosnia-Herzegovina, Medjugorje.

Blessed Virgin Mary

As stated by Jain (1991, p 171), if your performance metric is a *Higher is Better* for example throughput, it is better to use your opponent as the base. Moreover, if your performance metric is a *Lower is Better* metric, for example jitter; it is better your system as a base. In this line of thinking in our throughput calculations of ratios, we used TCP Reno, Westwood and Vegas as the bases respectively each at a time; and in our Jitter calculations of ratios, we used our proposal, TCP HolyWood as a base. With the supporting evidence collected in this work in a wired network environment, using the Network Simulator 2, version ns-2.1b8a, we arrive to the following conclusions:

About Throughput

With an interval of error rate from 0% to 60%, we get a gain in the average throughput of TCP HolyWood over TCP Reno of 73.46%; over TCP Westwood of 30.65% and over TCP Vegas of 67.46%. In general, in the interval studied of propagation time from 0.1 ms to 1000 ms, we measured the average throughput of TCP HolyWood and found that it outperforms TCP Reno in 53.59%, to TCP Westwood in 47.76% and to TCP Vegas in 66.42%. A General measurement of all the point of bottleneck bandwidth from 1 Mbit/s to 100 Mbit/s gave us an average throughput of TCP HolyWood of 77.49% over TCP Reno, 76.7% over TCP Westwood and 17.71% over TCP Vegas.

We conclude that varying the factors error rate, propagation time and bottleneck bandwidth in a wired network environment, using the nework simulator ns-2, TCP HolyWood presented better throughput perfomance than TCP Reno, TCP Westwood and TCP Vegas.

About Jitter

In all the measured values from an error rate of 0.0% to 60%, TCP Reno had 38.28% more jitter than our Proposal. TCP Westwood presented 0.11% less jitter than

TCP HolyWood, and TCP Vegas presented 265.32% more jitter than TCP HolyWood.

In general, in all the route of propagation time 0.1 ms to 1000 ms, TCP Reno showed 52.49% more jitter than TCP HolyWood, TCP Westwood showed 13.6% more jitter than TCP HolyWood, and TCP Vegas showed 64.17% more jitter than TCP HolyWood. In general, in all the route of bottleneck bandwidth, TCP Reno showed 76.81% more jitter than TCP HolyWood, TCP Westwood showed 8.95% less jitter than TCP HolyWood and TCP Vegas also showed 17.55% more jitter than TCP HolyWood.

We conclude that varying the factor error rate, propagation time and bottleneck bandwidth in a wired network environment, using the nework simulator ns-2, TCP HolyWood presented better average jitter perfomance than TCP Reno, and TCP Vegas and slightly less average jitter performance than TCP Westwood

About Fairness

We conclude that TCP HolyWood is as fair as TCP Reno.

About Friendliness

We conclude that when the number of TCP Reno competing with a single TCP HolyWood increase, the latter become more and more friendly with the other TCP Renos.

About Latency

In all the tests, we made amongst TCP HolyWood versus TCP Reno, TCP Vegas and TCP Westwood; our proposal had the higher average latency or packet delay and the difference was diminishing while the error rate of the link increased.

About Lost packets

In the interval of error rate from 0% to 60%, with TCP HolyWood as a base in all the cases, TCP Reno and TCP Vegas presented 8.61% and 25.47% respectively less percentage of lost packets than TCP HolyWood. Nevertheless, TCP Westwood presented 57.21% more lost packets than our proposal.

In other words, TCP HolyWood showed less percentage of lost packets than TCP Westwood and more percentage of lost packets than TCP Vegas and TCP Reno in all the route of error rate axis. It is important to remark that all TCPs sent different quantity of packets for the same simulation.

As an overall conclusion with all the simulations we made, our TCP HolyWood could work efficiently in a Wired Environment with less than 8% of Error Rate, neverless further tests are required for other network environments.

About future works

We will test TCP HolyWood in other simulated network environment as wireless networks, celular networks, satellite networks and hybrid ones.

We will bring TCP HolyWood to life implementing it in FreeBSD 4.x. or 5.x.

We will fix an apropriate test-bed to make real measurement with the TCP HolyWood in order to validate our future experiments with the results of this dissertation.

Thank you very much.

REFERENCES

ALLMAN, M.; FLOYD,S.; PARTRIDGE,C. **Increasing TCP's Initial Window**: RFC 2414. [S.l.]:. Internet Engineering Task Force, Networking Working Group, 1998. Available at: <ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc2001.txt.pdf >
Visited on: March 2004.

ALLMAN,M.; PAXSON,V.; STEVENS,W. **TCP Congestion Control**: RFC 2581.[S.l.]: Internet Engineering Task Force, Network Working Group, 1999 . Available at: <ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc2581.txt.pdf>**.** Visited on: Apr. 2004.

RAMAKRISHNAN,K.; FLOYD,S. **A Proposal to Add Explicit Congestion Notification (ECN) to IP**: RFC 2481.[S.l]: Internet Engineering Task Force, Network Working Group, 1999. Available at: <ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc2481.txt.pdf >. Visited on: March 2004.

ALTMAN, E.; JIMENEZ, T. **NS Simulator for Beginners**. Merida, Venezuela: Universidad de los Andes; France: ESSI Sophia-Antipolis, 2003. Available at: < http://www-sop.inria.fr/mistral/personnel/Eitan.Altman/COURS-NS/n3.pdf > . Visited on: March 2004.

AKYILDIZ, I. F. et al. InterPlaNetary Internet: State-of-the-art and research challenges. **Computer Networks,** [S.l.], v. 43, p.75-112, 2003. Available at: < http://www.ece.gatech.edu/research/labs/bwn/space.pdf >. Visited on: Apr. 2004.

ALLMAN, M.; FALK, A. On the Effective Evaluation of TCP. **ACM Computer Communication Review**, New York,Oct. 1999. Available at: <http://citeseer.ist.psu.edu/278079.html >. Visited on: Apr. 2004.

BALAKRISHNAN, H. et al .A Comparison of Mechanisms for Improving TCP Performance over Wireless Links**. IEEE/ACM Transactions on Networking,** Atlanta, v. 5, n. 6, p. 756-769, Dec. 1997.

BLESSED VIRGIN MARY. ***Queen of Peace***.- **Messages of Our Lady**. Medjugorje. Available at: <http://www.medjugorje.hr>, <http://www.medjugorje.org>. Visited on: Feb. 25, 2005.

BRADEN,R. **Requirements for Internet Hosts -- Communication Layers**: RFC1122.[S.l.]: Internet Engineering Task Force, Network Working Group Oct.1989. Available at:< ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc793.txt.pdf>**.**
Visited on: March 2004.

BRADEN, R.; BORMAN, D. **TCP Extensions for High Performance:** RFC 1323.[S.l.]: Internet Engineering Task Force, Network Working Group, May 1992. Available at: <ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc1323.txt.pdf >.
Visited on: March 2004.

CERF,V.; KAHN, R. A Protocol for Packet Network Intercommunication. **IEEE Transactions on Communications**, New York, COM-22, n. 5, p.637-648, May 1974.

COUTINHO, M. M. **Network Simulator**: Guia Basico para Iniciantes. 2003. Available at: < http://www.google.com.br/search?q=cache:tDK7vTEmhLAJ:www.stevelacerda.hpg.ig.com.br/Engenharia%2520de%2520Tr%25E1fego/Network%2520Simulator%2520-%2520Guia%2520B%25E1sico.pdf++Guia+Basico+Para+Iniciantes+Coutinho&hl=pt-BR&client=firefox-a >. Visited on: Dec. 2004.

DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION. **Transmission Control Protocol:** RFC 793. [S.l.]: Internet Engineering Task Force**,** Sept. 1981. Available at:< ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc793.txt.pdf >. Visited on: March 2004.

ELAARAG, H. Improving TCP Performance over Mobile Networks. **ACM Computing Surveys**, New York, v.3, n.3, p.357-374, Sept. 2002. Available at: <http://www.acm.org/>. Visited on: Aug. 28, 2002.

FALL, K.;FLOYD, S. Simulation-based Comparison of Tahoe, Reno, And SACK TCP. **Computer Communication Review**, [S.l.], July 1996. Available at: < http://citeseer.ist.psu.edu/fall96simulationbased.html >. Visited on: Apr. 2004.

FALL, K.; VARADHAN, K. The *ns* Manual. VINT Project. Berkeley:UCLA, LBL, USC/ISI, Xerox PARC. 2004. Available at: <http://www.isi.edu/nsnam/ns/ns-documentation>. Visited on: Jan. 2004.

FENG, Z.;MIN, L.; CHUANSHAN, G. An Analytic Throughput Model for TCP Reno over Wireless Networks. In: COMPUTER NETWORKS AND MOBILE COMPUTING, 2001. **Proceedings…** California, USA:[s.n.], 2001. p.111-116. Available at: < http://ieeexplore.ieee.org/>. Visited on: March 2004.

FOROUZAN, B. A. **Data Communication and Networking**. 3rd ed. Boston: McGraw-Hill, 2004.

FLOYD, S. TCP and Explicit Congestion Notification. **Computer Communication Review**, [S.l.], Oct. 1994. Available at: < http://citeseer.ist.psu.edu/floyd94tcp.html >. Visited on: Apr. 2004.

FU, C.P.; LIEW, S.C. TCP Veno: TCP Enhancement for Transmission Over Wireless Access Network. **IEEE Journal Selected Areas in Communication**, [S.l.], v.21, n.2, p. 216-228, Feb. 2003. Available at: < http://citeseer.ist.psu.edu/711567.html>. Visited on: Feb. 2004.

GERLA, M.; SANADIDI, M. Y.; WANG, R.; ZANELLA, A. TCP Westwood: congestion window control using bandwidth estimation. In: GLOBECOM, 2001, San Antonio, Texas, USA. **Proceedings…** NewYork: IEEE, 2001. p.1698-1702.

GOEBEL, G. **A Guided Tour of AWK**. Available at: < http://www.vectorsite.net/tsawk1.html>. Visited on: Feb. 2004.

JAIN, R. Congestion Control in Computer Networks: Issues and trends. **IEEE Network Magazine**, [S.l.], p.24-30, May 1990.

JAIN, R. **The Art of Computer Systems Performance Analysis**: Techniques for Experimental Design, Measurement, Simulation And Modeling. [S.l.]: John Wiley & Sons, 1991.

JACOBSON, V. Congestion Avoidance and Control. SIGCOMM, 1988, Standford, CA. **Proceedings…** Available at: <http://citeseer.ist.psu.edu/jacobson88congestion.html>.

Visited on: Aug. 2004.

JACOBSON, V. Congestion Avoidance and Control. **Computer Communication Review,** [S.l.], v.18, n.4, p.314-329, Aug. 1998. Available at: <http://citeseer.ist.psu.edu/context/4269/15205>. Visited on: Aug. 2004.

JACOBSON, V. Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno. In: THE INTERNET ENGINEER TASK FORCE,18., 1990. **Proceedings…** Vancouver, BC, Canada:[s.n.], 1990.

KARN, P.; PARTRIDGE, C. Improving Round-Trip times Estimates in Reliable Transport Protocols. **Computer Communication Review**, [S.l.], v.17, n.5, p.2-7, Aug. 1987. Available at: <http://zipper.paco.net/~igor/karn-partridge.ps>. Visited on: May 2004.

KAWANO,T. **Tutorial on line of GNUPLOT**.
Available at: <http://t16web.lanl.gov/Kawano/ >. Visited on: Aug. 2004.

KDE:The K Desktop Environment. Available at: < http://www.kde.org/>. Visited on: Aug. 2004

KOJIMA, A. **Window Maker**. Available at: < http://www.windowmaker.org/>. Visited on: Aug. 2004.

LILJA, D. J. **Measuring Computer Performance**: A practitioner's Guide. [S.l.]: Cambridge University Press, 2002.

LOW, S. H.; PETERSON, L. L. ; WANG, L. Understanding TCP Vegas a duality Model. **Journal of the ACM**, New York, v.49, n.2, p.207–235, Mar. 2002. Available at: <http://citeseer.ist.psu.edu/low01understanding.html> Visited on: Aug. 2004

MALEK, J. **TRACE GRAPH**. Poland: Wroclaw University of Technology, 2003. Available at: < http://www.geocities.com/tracegraph/>. Visited on: Jan. 2003.

MATHIS,M.; MAHDAVI,J.; FLOYD,S;ROMANOV, A.**TCP Selective Acknowledgement Options:**RFC 2018. [S.l.]: Internet Engineering Task Force, Networking Working Group, Oct.1996. Available at: <ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc2018.txt.pdf > . Visited on: March 2004.

MATHWORKS, INC. **MATLAB 6.1.** Available at: <http://www.mathworks.com/> . Visited on: Aug. 2004.

MACDOUGALL, M.H. **Simulating Computer System Techniques and Tools**. Cambridge: MIT, 1987.

MATHIS, M.; MAHDAVI, J. Forward Acknowledgement: Refining TCP Congestion Control. In ACM SIGCOM, 2001. **Proceedings…** New York: ACM, 1996.

NAGLE, J. **Congestion Control in IP/TCP Internetworks**: RFC 896. [S.l.]: Internet Engineering Task Force, Network Working Group, Jan. 1984. Available at: <ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc793.txt.pdf >. Visited on: March 2004.

NETWORK Simulator: ns-2. Available at: <http://www.isi.edu/nsnam/ns/>. Visited on: Jan, 2004.

NIEDERREITER, T. **XCDROAST**. Available at: < http://www.xcdroast.org/> . Visited on: Aug. 2004.

OPENOFFICE.ORG. **OpenOffice Word Processor**. Available at: <http://www.openoffice.org/> Visited on: Jan. 2004.

PADHYE, J. et al. Modeling TCP Reno Performance: A simple Model and Its

Empirical Validation. **IEEE/ACM Transactions on Networking**, [S.l.], v.8, n.2, p.133-145, Apr. 2000. Available at: <http://citeseer.ist.psu.edu/padhye00modeling.html>. Visited on Mar. 2004.

PARSA, C.; GARCIA – LUNA - ACEVES, J. J. Differentiating congestion vs. Random Loss: A Method for Improving TCP Performance over Wireless links. In: WIRELESS COMMUNICATIONS AND NETWORKING CONFERENCE, WCNC, 2000. **Proceedings…** Chicago, IL, USA:[s.n.], 2000. v. 1, p. 90-93.

PAXTON, V. Automated Packet Trace Analysis of TCP implementations. In: IGCOMM, 1997. **Proceedings…** Available at: <http://cerberus.sourcefire.com/~jeff/papers/Vern_Paxson/paxson97automated.pdf >. Visited on: Aug. 2004.

STEVENS,W.**TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms:** RFC 2001. [S.l.]: Internet Engineering Task Force, Networking group. Jan. 1997. Available at: < ftp://ftp.rfc-editor.org/in-notes/pdfrfc/rfc2001.txt.pdf >. Visited on: March 2004.

ROBBINS, A.D. **GAWK**: Effective AWK Programming. A User's Guide for GNU Awk. 3rd ed. June 2003. Available at: <http://www.gnu.org/>. Visited on: Feb. 2004.

ROCHOL, J. et al. Plataformas de Simulacao de Software Livre para Redes Fixas e Moveis: Caracteristicas, Suporte, Instalacao e Validação. In: INTERNATIONAL INFORMATION AND TELECOMUNICATION TECHNOLOGIES SYMPOSIUM, 2., 2003. **Proceedings…** [S.l.:s.n.], 2003.

RODRIGUEZ, A. et al.**TCP/IP Tutorial Technical Overview.** Available at: < http:// www.ibm.com/redbooks> . Visited on: Aug. 25, 2003.

SANADIDI, M.Y. Tutorial: Congestion Control for Hybrid High-Speed Global Networks. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, SBRC, 20., 2002, Búzios, RJ. **Anais...** [S.l.:s.n.], 2002.

SEMKE, R.; MAHDAVI, J. **The rate-halving algorithm for TCP congestion control.** Available at: < http://www.psc.edu/networking/ftp/papers/draft-ratehalving.txt >. Visited on: March 2003.

STALLINGS, W. **Data & Computer Communications**. 6th ed. Upper Saddle River: Prentice Hall, 2000.

STALLINGS, W. **Computer Networking with Internet Protocols and Technology.** Upper Saddle River: Prentice Hall, 2004.

STEVENS, W R. **TCP/IP Illustrated**. [S.l.]: Addison-Wesley, 1994. v.1.

UNIVERSITY OF CALIFORNIA. **FreeBSD**. Berkeley. Available at: < http://www.freebsd.org/>. Visited on: Jan. 2005.

UNIVERSITY OF CALIFORNIA. Computer Science Department. **TCP Westwood**. Los Angeles. Available at: <http://www.cs.ucla.edu/NRL/hpi/tcpw/index.html>. Visited on: Feb. 2004.

UNIVERSITY OF CALIFORNIA. Paralell Computing Laboratory. **GloMoSim**: Global Mobile Information Systems Simulation Library. Los Angeles. Available at: <http://pcl.cs.ucla.edu/projects/glomosim/>. Visited on: Dec. 2003.

UNIVERSITY OF BOSTON. Computer Science Department. Boston. Available at: <http://cs-people.bu.edu/guol/bu_ns/buns-faq.html>. Visited on: March

2004.

TSAOUSSIDIS, V.; BADR, H. TCP-Probing: Towards an Error Control Schema with Energy and Throughput Performance gains. In: IEEE INTERNATIONAL CONFERENCE ON NETWORK PROTOCOLS, 2000. **Proceedings…** Osaka, Japan:[s.n.], 2000. p.12-21. Available at: <http://citeseer.ist.psu.edu/tsaoussidis00tcpprobing.html >. Visited on: March 2004.

WANG, S.Y. et al. The Design and Implementation of the NCTUns 1.0 Network Simulator. **Computer Networks**, [S.l.], v. 42, n. 2, p.175-197, June 2003. Available at: <http://www.csie.nctu.edu.tw/~shieyuan/publications/NCTUNSDesign.pdf>. Visited on: Feb. 2004.

WILLIAMS, T.; KELLEY, C. **Gnuplot**. Available at: <http://www.gnuplot.info/>. Visited on: March 2004.

APPENDIX A STATISTICS OF THE RESULTS

As mentioned by Lilja (2000) to determine how much uncertainty exists in our measurements, and, therefore to determine what conclusions we can actually draw from them, we must use the tools and techniques of probability and statistics to quantify the errors. Without more preambles, we start this appendix.

A.1 Statistics of TCP HolyWood versus TCP Reno

A.1.1 Statistics of Average throughput versus Error Rate of TCP HolyWood and TCP Reno

Table A.1.1: Statistics of Average throughput versus Error Rate

Error Rate [%]	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
	Reno	4865326.62	663508.73	13.64	4759143.11	4971510.13
0	HolyWood	4896404.77	633845.59	12.95	4794968.35	4997841.19
	Reno	4627530.6	822065.71	17.76	4495972.68	4759088.51
0.01	HolyWood	4896404.77	633845.59	12.95	4794968.35	4997841.19
	Reno	4528733.25	856467.01	18.91	4391669.98	4665796.51
0.02	HolyWood	4889061.72	636948.72	13.03	4787128.7	4990994.75
	Reno	4186131.92	1052124.63	25.13	4017756.91	4354506.93
0.04	HolyWood	4826237.88	771685.29	15.99	4702742.51	4949733.25
	Reno	3956791.52	1149937.9	29.06	3772763.13	4140819.91
0.06	HolyWood	4872669.67	647185.33	13.28	4769098.45	4976240.89
	Reno	3623016.69	1172662.72	32.37	3435351.57	3810681.81
0.08	HolyWood	4861098.81	650241.49	13.38	4757038.5	4965159.12
	Reno	3352807.42	1249225.86	37.26	3152889.64	3552725.19
0.1	HolyWood	4791376.95	718808.28	15	4676343.67	4906410.24
	Reno	2638529.27	1168046.8	44.27	2451602.85	2825455.69
0.2	HolyWood	4326094.83	838073.81	19.37	4191975.09	4460214.58
	Reno	1799938.54	862173.06	47.9	1661962.12	1937914.97
0.4	HolyWood	3210470.99	932419.57	29.04	3061252.78	3359689.2
	Reno	1341406.09	661411	49.31	1235558.29	1447253.9
0.6	HolyWood	2507837.88	784043.24	31.26	2382364.83	2633310.93
	Reno	1014455.1	538890.88	53.12	928214.6	1100695.6
0.8	HolyWood	2021268.34	686692.64	33.97	1911374.64	2131162.05
	Reno	883615.36	487870.93	55.21	805539.75	961690.97
1	HolyWood	1669247.15	656142.39	39.31	1564242.5	1774251.8
	Reno	533300.13	390551.99	73.23	470798.8	595801.47
2	HolyWood	992129.27	565662.21	57.01	901604.46	1082654.08
	Reno	228823.31	208900.97	91.29	195392.19	262254.43
4	HolyWood	590857.75	420985.86	71.25	523485.98	658229.52
	Reno	168224.64	158803.42	94.4	142810.8	193638.48
6	HolyWood	356399.47	324604.72	91.08	304451.9	408347.04
	Reno	132251.13	131483.6	99.42	111209.37	153292.88
8	HolyWood	56242.29	177354	315.34	27859.73	84624.84
	Reno	101692.19	101083.87	99.4	85515.4	117868.97
10	HolyWood	254931.92	256813.37	100.74	213833.22	296030.62
	Reno	39090.86	54004.73	138.15	30448.3	47733.42
20	HolyWood	98482.29	215278.09	218.6	64030.62	132933.96
	Reno	7409.03	17887.45	241.43	4546.44	10271.62
40	HolyWood	22715.52	81925.11	360.66	9604.78	35826.27

A.1.2 Statistics of Average throughput versus Propagation Time of TCP HolyWood and TCP Reno

Table A.1.2: Statistics of Average throughput versus Propagation Time

Propagation Time [ms]	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
0.1	Reno HolyWood	4895069.67 4929559.74	642956.41 518288.54	13.13 10.51	4792175.21 4846616.29	4997964.12 5012503.18
1	Reno HolyWood	4893289.54 4928966.36	648763.03 519971.77	13.26 10.55	4789465.83 4845753.54	4997113.24 5012179.17
5	Reno HolyWood	4868367.68 4864881.59	690110.44 698594.05	14.18 14.36	4757927.01 4753083.26	4978808.35 4976679.92
10	Reno HolyWood	4793750.46 4857835.23	708770.07 709379.28	14.79 14.6	4680323.63 4744310.9	4907177.3 4971359.56
50	Reno HolyWood	2511175.63 4358656.42	1234030.08 886297.09	49.14 20.33	2313689.69 4216819.35	2708661.57 4500493.5
100	Reno HolyWood	1187202.12 2333904.11	621988.98 699653.65	52.39 29.98	1087663.15 2221936.2	1286741.09 2445872.01
200	Reno HolyWood	579806.09 1143885.56	287302.64 377107.53	49.55 32.97	533828.09 1083535.79	625784.09 1204235.34
300	Reno HolyWood	427975.63 702264.37	221803.43 261031.3	51.83 37.17	392479.69 660490.66	463471.57 744038.08
400	Reno HolyWood	311673.64 561856.42	156516.9 211957.34	50.22 37.72	286625.72 527936.19	336721.56 595776.66
500	Reno HolyWood	238020.66 467583.58	125776.96 157804.92	52.84 33.75	217892.16 442329.53	258149.17 492837.62
1000	Reno HolyWood	121260.8 216460.8	153519.72 227760.44	126.6 105.22	96692.53 180011.54	145829.07 252910.06

A.1.3 Statistics of Average throughput versus Bottleneck Bandwidth of TCP HolyWood and TCP Reno

Table A.1.3: Statistics of Average throughput versus Bottleneck Bandwidth

Bottleneck Bandwidth [Mb]	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
1	Reno	964314.7	149417.1	15.49	940402.99	988226.42
	HolyWood	971954.44	138942.7	14.3	949718.97	994189.9
10	Reno	3592680.26	1590733.87	44.28	3338109.83	3847250.7
	HolyWood	6540060.4	1768192.58	27.04	6257090.65	6823030.14
20	Reno	3662476.29	1612779.86	44.04	3404377.76	3920574.82
	HolyWood	6685957.09	1832154.7	27.4	6392751.27	6979162.9
30	Reno	3689474.97	1645676.51	44.6	3426111.87	3952838.06
	HolyWood	6722672.32	1839144.84	27.36	6428347.85	7016996.79
40	Reno	3703790.2	1653917.34	44.65	3439108.3	3968472.1
	HolyWood	6742105.43	1845212.49	27.37	6446809.93	7037400.93
50	Reno	3714916.03	1663338.75	44.77	3448726.39	3981105.67
	HolyWood	6737432.58	1873226.01	27.8	6437653.99	7037211.18
60	Reno	3719218.01	1647535.16	44.3	3455557.48	3982878.55
	HolyWood	7011795.5	1779664.56	25.38	6726989.85	7296601.14
70	Reno	3726264.37	1660979.91	44.57	3460452.23	3992076.52
	HolyWood	7015578.28	1777549.77	25.34	6731111.07	7300045.49
80	Reno	3730195.5	1664218.13	44.61	3463865.13	3996525.87
	HolyWood	7019286.89	1779583.9	25.35	6734494.15	7304079.63
90	Reno	3730195.5	1666108.96	44.67	3463562.53	3996828.46
	HolyWood	7023069.67	1776242.47	25.29	6738811.67	7307327.67
100	Reno	3734052.45	1664344.9	44.57	3467701.79	4000403.11
	HolyWood	7026407.42	1779231.04	25.32	6741671.15	7311143.68

A.1.4 Statistics of the Average Jitter Versus Error Rate of TCP HolyWood with TCP Reno

Table A.1.4.1: Statistics of the Average Jitter versus Error Rate

Error Rate [%]	Protocol [TCP]	Average Jitter [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
0	Reno	2.265	1.168	51.572	2.078	2.452
	HolyWood	2.251	1.108	49.238	2.074	2.428
0.01	Reno	2.375	2.061	86.756	2.046	2.705
	HolyWood	2.251	1.108	49.249	2.073	2.428
0.02	Reno	2.424	2.271	93.687	2.061	2.788
	HolyWood	2.252	1.114	49.491	2.073	2.43
0.04	Reno	2.615	3.379	129.219	2.075	3.156
	HolyWood	2.281	4.41	193.331	1.575	2.987
0.06	Reno	2.76	4.004	145.063	2.119	3.401
	HolyWood	2.258	1.15	50.928	2.074	2.442
0.08	Reno	3.005	4.752	158.149	2.245	3.766
	HolyWood	2.26	1.168	51.705	2.073	2.447
0.1	Reno	3.238	5.738	177.192	2.32	4.157
	HolyWood	2.29	1.431	62.48	2.061	2.519
0.2	Reno	4.087	7.949	194.527	2.814	5.359
	HolyWood	2.52	2.012	79.848	2.198	2.842
0.4	Reno	5.869	12.222	208.235	3.913	7.825
	HolyWood	3.346	4.707	140.671	2.593	4.1
0.6	Reno	7.807	16.426	210.396	5.179	10.436
	HolyWood	3.346	4.707	140.671	2.593	4.1
0.8	Reno	9.519	23.505	246.93	5.757	13.281
	HolyWood	5.169	9.92	191.905	3.582	6.757
1	Reno	10.654	25.334	237.783	6.6	14.709
	HolyWood	6.059	12.283	202.725	4.093	8.025
2	Reno	12.772	49.884	390.558	4.789	20.756
	HolyWood	8.661	21.869	252.501	5.161	12.161
4	Reno	9.728	113.522	1166.934	-8.439	27.895
	HolyWood	10.169	40.999	403.189	3.608	16.73
6	Reno	3.643	140.253	3849.488	-18.802	26.089
	HolyWood	11.366	91.805	807.699	-3.326	26.058
8	Reno	-2.553	172.285	-6748.516	-30.124	25.018
	HolyWood	19.199	252.426	1314.799	-21.198	59.595
10	Reno	-13.261	218.624	-1648.655	-48.248	21.726
	HolyWood	6.773	91.522	1351.279	-7.874	21.419
20	Reno	-116.062	729.863	-628.858	-232.864	0.741
	HolyWood	23.621	551.257	2333.787	-64.599	111.84
40	Reno	8.691	3696.827	42536.961	-582.925	600.306
	HolyWood	-111.431	872.937	-783.388	-251.13	28.268
60	Reno	-139.133	7202.566	-5176.733	-1291.784	1013.517
	HolyWood	-187.86	921.797	-490.683	-335.378	-40.342

Table A.1.4.2: Ratio with TCP HolyWood as a base for the Average Jitter versus Error Rate of TCP HolyWood and TCP Reno

Error Rate [%]	Avg. Jitter TCP [ms] HolyWood	Avg. Jitter TCP [ms] Reno	TCP [ms] HolyWood As a Base	TCP [ms] abs(Reno / HolyWood)
0.0001 ~ 0	2.2509	2.2647	1	1.0061
0.01	2.2507	2.3753	1	1.0554
0.02	2.2517	2.4243	1	1.0766
0.04	2.2809	2.6153	1	1.1466
0.06	2.2582	2.76	1	1.2222
0.08	2.2596	3.005	1	1.3299
0.1	2.29	3.2383	1	1.4141
0.2	2.5198	4.0865	1	1.6218
0.4	3.3464	5.8694	1	1.7539
0.6	4.2121	7.8073	1	1.8535
0.8	5.1694	9.5191	1	1.8414
1	6.059	10.6544	1	1.7584
2	8.661	12.7724	1	1.4747
4	10.1686	9.7282	1	0.9567
6	11.3662	3.6434	1	0.3205
8	19.1988	-2.5529	1	0.133
10	6.773	-13.2608	1	1.9579
20	23.6207	-116.062	1	4.9136
40	-111.431	8.6909	1	0.078
60	-187.86	-139.133	1	0.7406

A.1.5 Statistics of Average Jitter Versus Propagation Time of TCP HolyWood with TCP Reno

Table A.1.5.1: Statistics of Average Jitter versus Propagation Time

Propagation Time [ms]	Protocol [TCP]	Average Jitter [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
0.1	Reno HolyWood	2.236 2.233	1.437 1.366	64.27 61.173	2.006 2.014	2.466 2.451
1	Reno HolyWood	2.237 2.233	1.442 1.37	64.486 61.374	2.006 2.013	2.467 2.452
5	Reno HolyWood	2.247 2.241	1.487 1.922	66.19 85.773	2.009 1.933	2.485 2.548
10	Reno HolyWood	2.285 2.244	1.574 1.961	68.89 87.368	2.033 1.931	2.537 2.558
50	Reno HolyWood	4.319 2.514	10.19 3.129	235.943 124.46	2.688 2.013	5.949 3.015
100	Reno HolyWood	9.161 4.687	25.164 14.009	274.696 298.859	5.134 2.446	13.188 6.929
200	Reno HolyWood	18.75 9.548	49.552 29.553	264.281 309.516	10.82 4.819	26.68 14.278
300	Reno HolyWood	25.384 15.546	76.676 47.443	302.068 305.183	13.113 7.953	37.655 23.138
400	Reno HolyWood	34.62 19.414	83.67 63.826	241.679 328.766	21.23 9.2	48.01 29.628
500	Reno HolyWood	45.425 23.327	114.798 80.254	252.722 344.038	27.053 10.484	63.796 36.17
1000	Reno HolyWood	88.103 50.311	297.122 207.03	337.243 411.5	40.554 17.179	135.653 83.443

Table A.1.5.2: Ratio with TCP HolyWood as a Base for Statistics of Average Jitter versus Propagation Time

Propagation Time [ms]	Avg. Jitter TCP [ms] HolyWood	Avg. Jitter TCP [ms] Reno	TCP [ms] HolyWood As a Base	TCP [ms] Reno / HolyWood
0.1	2.2325	2.2359	1	1.0015
1	2.2326	2.2365	1	1.0017
5	2.2405	2.2468	1	1.0028
10	2.2443	2.2847	1	1.018
50	2.5142	4.3186	1	1.7177
100	4.6874	9.1607	1	1.9543
200	9.5481	18.7498	1	1.9637
300	15.5459	25.3838	1	1.6328
400	19.4138	34.6201	1	1.7833
500	23.3271	45.4247	1	1.9473
1000	50.3111	88.1032	1	1.7512

A.1.6 Statistics of the Average Jitter Versus Bottleneck Bandwidth of TCP HolyWood with TCP Reno

Table A.1.6.1: Statistics of the Average Jitter versus Bottleneck Bandwidth

Bottleneck Bandwidth [Mb]	Protocol [TCP]	Average Jitter [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
1	Reno	11.311	4.419	39.068	10.604	12.018
	HolyWood	11.217	4.86	43.327	10.439	11.994
10	Reno	3.038	9.611	316.307	1.5	4.576
	HolyWood	1.678	5.997	357.417	0.718	2.638
20	Reno	2.98	11.81	396.281	1.09	4.87
	HolyWood	1.636	7.117	435.027	0.497	2.775
30	Reno	2.958	12.831	433.793	0.904	5.011
	HolyWood	1.626	8.263	508.058	0.304	2.949
40	Reno	2.946	13.378	454.104	0.805	5.087
	HolyWood	1.627	9.478	582.666	0.11	3.144
50	Reno	2.938	13.713	466.8	0.743	5.132
	HolyWood	1.622	9.435	581.595	0.112	3.132
60	Reno	2.935	13.947	475.263	0.703	5.167
	HolyWood	1.564	9.402	601.107	0.06	3.069
70	Reno	2.929	14.107	481.665	0.671	5.187
	HolyWood	1.563	9.643	616.908	0.02	3.106
80	Reno	2.926	14.232	486.404	0.648	5.203
	HolyWood	1.562	9.826	628.943	-0.01	3.135
90	Reno	2.925	14.332	490.023	0.631	5.218
	HolyWood	1.562	9.97	638.392	-0.034	3.157
100	Reno	2.922	14.408	493.033	0.617	5.228
	HolyWood	1.561	10.084	645.897	-0.053	3.175

Table A.1.6.2: Ratio with TCP HolyWood as a Base for Statistics of the Average Jitter versus Bottleneck Bandwidth

Bottleneck Bandwidth [Mb]	Jitter TCP [ms] HolyWood	Jitter TCP [ms] Reno	TCP [ms] HolyWood As a Base	TCP [ms] Reno/ HolyWood
1	11.2165	11.3109	1	1.0084
10	1.6779	3.0384	1	1.8109
20	1.636	2.9803	1	1.8217
30	1.6263	2.9578	1	1.8187
40	1.6267	2.9459	1	1.8109
50	1.6222	2.9377	1	1.811
60	1.5641	2.9346	1	1.8762
70	1.5631	2.9289	1	1.8737
80	1.5624	2.9259	1	1.8727
90	1.5617	2.9248	1	1.8728
100	1.5613	2.9224	1	1.8718

A.1.7 Statistics of Average Throughput Versus Increasing Number of Flows - Fairness

Table A.1.7: Statistics of Average Throughput versus Increasing Number of Flows (Fairness)

Number of Independent Flows	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
	Reno	3124950.464	1247706.15	39.927	2925938.2	3323962.8
1	HolyWood	4766455.099	741493.323	15.556	4648185	4884725.2
	Reno	2284505.43	798191.942	34.939	2194481.1	2374529.8
2	HolyWood	2450539.868	650791.425	26.557	2377140.1	2523939.6
	Reno	1612381.81	661978.185	41.056	1551421	1673342.6
3	HolyWood	1621999.47	575782.618	35.498	1568976.3	1675022.6
	Reno	1212828.609	616188.432	50.806	1163686.8	1261970.4
4	HolyWood	1224659.073	539292.076	44.036	1181649.8	1267668.3
	Reno	976849.801	485416.209	49.692	942224.2	1011475.4
5	HolyWood	978140.397	481300.915	49.206	943808.4	1012472.4
	Reno	817874.083	439888.94	53.784	789230	846518.2
6	HolyWood	819110.287	415002.543	50.665	792086.7	846133.9
	Reno	699763.708	408942.182	58.44	675110.1	724417.3
7	HolyWood	696913.377	387348.306	55.581	673561.6	720265.2
	Reno	612312.052	368278.427	60.146	591543.9	633080.2
8	HolyWood	614815.364	332967.691	54.157	596038.4	633592.3
	Reno	545447.888	360450.329	66.083	526283.6	564612.1
9	HolyWood	542744.724	364115.834	67.088	523385.6	562103.9
	Reno	491480.981	297716.308	60.575	476459.5	506502.5
10	HolyWood	492379.338	297295.677	60.379	477384	507374.7

A.1.8 Statistics Friendliness between TCP HolyWood and TCP Reno

Table A.1.8: Statistics Friendliness between TCP HolyWood and TCP Reno

Number Of Renos	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
	Reno	1681485.562	598564.14	35.597	1586013.1	1776958.1
1	HolyWood	3193114.702	685228.424	21.46	3083819.1	3302410.4
	Reno	1327610.066	607534.063	45.761	1259089.1	1396131
2	HolyWood	2220643.179	733902.616	33.049	2103583.8	2337702.5
	Reno	1065411.391	480227.006	45.074	1021187.8	1109634.9
3	HolyWood	1699064.371	477029.656	28.076	1622976.9	1775151.8
	Reno	862068.344	484704.104	56.226	823412.6	900724.1
4	HolyWood	1437384.9	634145.834	44.118	1336237	1538532.8
	Reno	740746.95	393026.176	53.058	712693.1	768800.8
5	HolyWood	1210418.013	466836.3	38.568	1135956.4	1284879.6
	Reno	647859.073	373961.677	57.723	623507.9	672210.2
6	HolyWood	1012007.417	516640.616	51.051	929601.9	1094412.9
	Reno	565861.722	346493.718	61.233	544972.9	586750.5
7	HolyWood	948367.682	390339.161	41.159	886107.6	1010627.8
	Reno	506748.766	313893.185	61.943	489040.2	524457.4
8	HolyWood	862698.808	366472.824	42.48	804245.5	921152.2
	Reno	465102.929	316005.709	67.943	448301.7	481904.2
9	HolyWood	711684.238	403751.161	56.732	647284.9	776083.6

A.2 Statistics of TCP HolyWood versus Other Protocols

A.2.1 Statistics of Average Throughput versus Error Rate of TCP HolyWood with other Protocols

Table A.2.1.1: Statistics of the Average Throughput versus Error Rate of TCP HolyWood with other Protocols. (1 of 2)

Error Rate [%]	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
0	Vegas	4896105.96	603488.8	12.33	4799527.64	4992684.28
	Westwood	3895673.64	1567463.81	40.24	3644827.19	4146520.1
	HolyWood	4896404.77	633845.59	12.95	4794968.35	4997841.19
0.01	Vegas	4797531.13	652481.41	13.6	4693112.35	4901949.9
	Westwood	3739912.05	1603356.57	42.87	3483321.56	3996502.54
	HolyWood	4896404.77	633845.59	12.95	4794968.35	4997841.19
0.02	Vegas	4752879.47	639777.21	13.46	4650493.79	4855265.15
	Westwood	3549957.09	1668102.71	46.99	3283005.05	3816909.12
	HolyWood	4889061.72	636948.72	13.03	4787128.7	4990994.75
0.04	Vegas	4626490.07	699349.53	15.12	4514570.83	4738409.3
	Westwood	3399239.21	1710892.88	50.33	3125439.32	3673039.09
	HolyWood	4826237.88	771685.29	15.99	4702742.51	4949733.25
0.06	Vegas	4509075.5	764934.23	16.96	4386660.52	4631490.47
	Westwood	3365787.55	1688169.21	50.16	3095624.21	3635950.89
	HolyWood	4872669.67	647185.33	13.28	4769098.45	4976240.89
0.08	Vegas	4369186.75	784669.74	17.96	4243613.44	4494760.07
	Westwood	3210470.99	1709455.63	53.25	2936901.12	3484040.87
	HolyWood	4861098.81	650241.49	13.38	4757038.5	4965159.12
0.1	Vegas	4184646.36	869428.44	20.78	4045508.83	4323783.89
	Westwood	3155954.44	1680946.43	53.26	2886946.98	3424961.89
	HolyWood	4791376.95	718808.28	15	4676343.67	4906410.24
0.2	Vegas	3628354.97	896806.86	24.72	3484835.98	3771873.95
	Westwood	2852516.03	1595126.38	55.92	2597242.64	3107789.41
	HolyWood	4326094.83	838073.81	19.37	4191975.09	4460214.58
0.4	Vegas	2859782.78	819541.98	28.66	2728628.75	2990936.81
	Westwood	2566285.56	1501043.55	58.49	2326068.56	2806502.56
	HolyWood	3210470.99	932419.57	29.04	3061252.78	3359689.2
0.6	Vegas	2256911.26	691586.92	30.64	2146234.3	2367588.22
	Westwood	2377517.35	1408168.06	59.23	2152163.53	2602871.17
	HolyWood	2507837.88	784043.24	31.26	2382364.83	2633310.93
0.8	Vegas	1909785.43	609636.56	31.92	1812223.26	2007347.6
	Westwood	2399027.28	1327363.91	55.33	2186604.82	2611449.75
	HolyWood	2021268.34	686692.64	33.97	1911374.64	2131162.05
1	Vegas	1612651.66	554736.63	34.4	1523875.31	1701428.01
	Westwood	2312616.69	1239234.29	53.59	2114297.9	2510935.48
	HolyWood	1669247.15	656142.39	39.31	1564242.5	1774251.8

Table A.2.1.2: Statistics of Average Throughput versus Error Rate of TCP HolyWood with other Protocols (2 of 2)

Error Rate [%]	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
2	Vegas	1026543.05	438971.51	42.76	956292.97	1096793.12
	Westwood	1565925.3	962055.37	61.44	1411964.37	1719886.22
	HolyWood	992129.27	565662.21	57.01	901604.46	1082654.08
4	Vegas	631353.64	340312	53.9	576892.38	685814.9
	Westwood	782060.8	611963.59	78.25	684126.23	879995.37
	HolyWood	590857.75	420985.86	71.25	523485.98	658229.52
6	Vegas	373605.3	288355.51	77.18	327458.81	419751.79
	Westwood	488055.1	429660.17	88.04	419295.15	556815.05
	HolyWood	356399.47	324604.72	91.08	304451.9	408347.04
8	Vegas	197075.5	201534.36	102.26	164823.28	229327.71
	Westwood	317186.13	341407.43	107.64	262549.56	371822.7
	HolyWood	56242.29	177354	315.34	27859.73	84624.84
10	Vegas	147899.34	181887.99	122.98	118791.2	177007.48
	Westwood	207684.24	249262.14	120.02	167793.99	247574.49
	HolyWood	254931.92	256813.37	100.74	213833.22	296030.62
20	Vegas	45986.75	60246.59	131.01	36345.29	55628.22
	Westwood	45395.5	68306.81	150.47	34464.13	56326.86
	HolyWood	98482.29	215278.09	218.6	64030.62	132933.96
40	Vegas	6675.5	16213.78	242.89	4080.75	9270.24
	Westwood	7409.03	17887.45	241.43	4546.44	10271.62
	HolyWood	22715.52	81925.11	360.66	9604.78	35826.27
60	Vegas	605.41	2549.88	421.19	197.34	1013.47
	Westwood	2865.78	7794.78	272	1618.35	4113.2
	HolyWood	6133.72	25079.13	408.87	2120.23	10147.22

A.2.2 Statistics of Average Throughput versus Propagation Time of TCP HolyWood with other Protocols

Table A.2.2: Statistics of Average Throughput Versus Propagation Time of TCP HolyWood with other Protocols

Propagation Time [ms]	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
0.1	Vegas	4955740.4	414772.83	8.37	4889362.92	5022117.87
	Westwood	3107297.48	1767982.42	56.9	2824361.37	3390233.6
	HolyWood	4929559.74	518288.54	10.51	4846616.29	5012503.18
1	Vegas	4955592.05	416390.58	8.4	4888955.68	5022228.42
	Westwood	3007832.58	1795747.13	59.7	2720453.19	3295211.98
	HolyWood	4928966.36	519971.77	10.55	4845753.54	5012179.17
5	Vegas	4914945.7	427608.18	8.7	4846514.13	4983377.26
	Westwood	3321210.07	1757099.42	52.91	3040015.6	3602404.54
	HolyWood	4864881.59	698594.05	14.36	4753083.26	4976679.92
10	Vegas	4905748.34	442948.29	9.03	4834861.85	4976634.83
	Westwood	3264839.21	1842878.26	56.45	2969917.26	3559761.15
	HolyWood	4857835.23	709379.28	14.6	4744310.9	4971359.56
50	Vegas	3472445.03	940492.29	27.08	3321934.92	3622955.15
	Westwood	2642682.91	1506907.23	57.02	2401527.53	2883838.3
	HolyWood	4358656.42	886297.09	20.33	4216819.35	4500493.5
100	Vegas	1295936.42	590935.02	45.6	1201367.12	1390505.72
	Westwood	1501395.5	979031.17	65.21	1344717.88	1658073.11
	HolyWood	2333904.11	699653.65	29.98	2221936.2	2445872.01
200	Vegas	637287.42	311136.34	48.82	587495.23	687079.6
	Westwood	790199.47	751145.72	95.06	669991.12	910407.82
	HolyWood	1143885.56	377107.53	32.97	1083535.79	1204235.34
300	Vegas	423523.18	221096.83	52.2	388140.32	458906.04
	Westwood	543980.93	604603.79	111.14	447224.17	640737.69
	HolyWood	702264.37	261031.3	37.17	660490.66	744038.08
400	Vegas	239131.13	149388.57	62.47	215223.98	263038.28
	Westwood	451636.56	457936.94	101.4	378351.38	524921.73
	HolyWood	561856.42	211957.34	37.72	527936.19	595776.66
500	Vegas	194479.47	126412.78	65	174249.21	214709.73
	Westwood	342010.07	364174.91	106.48	283729.94	400290.19
	HolyWood	467583.58	157804.92	33.75	442329.53	492837.62
1000	Vegas	70410.67	100287.86	142.43	54361.27	86460.07
	Westwood	142167.47	217395.3	152.91	107376.97	176957.96
	HolyWood	216460.8	227760.44	105.22	180011.54	252910.06

A.2.3 Statistics of Throughput Versus Bottleneck Bandwidth of TCP HolyWood with other Protocols

Table A.2.3: Statistics of Throughput versus Bottleneck Bandwidth of TCP HolyWood with other Protocols

Bottleneck Bandwidth [Mb]	Protocol [TCP]	Average [bps] Throughput	Standard Dev. [bps]	COV [%]	C1 (95%) [bps]	C2 (95%) [bps]
1	Vegas	985006.62	96161.75	9.76	969617.54	1000395.71
	Westwood	959048.48	145950.91	15.22	935691.47	982405.49
	HolyWood	971954.44	138942.7	14.3	949718.97	994189.9
10	Vegas	5409154.97	1472850.61	27.23	5173449.78	5644860.15
	Westwood	3592086.89	2395508.68	66.69	3208725.66	3975448.12
	HolyWood	6540060.4	1768192.58	27.04	6257090.65	6823030.14
20	Vegas	5409377.48	1873395.21	34.63	5109571.81	5709183.16
	Westwood	3804367.68	3394824.23	89.23	3261082.66	4347652.71
	HolyWood	6685957.09	1832154.7	27.4	6392751.27	6979162.9
30	Vegas	5537769.54	1850213.62	33.41	5241673.69	5833865.38
	Westwood	3856362.38	3863139.08	100.18	3238131.37	4474593.39
	HolyWood	6722672.32	1839144.84	27.36	6428347.85	7016996.79
40	Vegas	5829933.77	1606934.64	27.56	5572770.67	6087096.88
	Westwood	3515021.99	4039414.66	114.92	2868581.01	4161462.97
	HolyWood	6742105.43	1845212.49	27.37	6446809.93	7037400.93
50	Vegas	5841282.12	1605638.26	27.49	5584326.48	6098237.76
	Westwood	3205427.28	3451029.48	107.66	2653147.54	3757707.02
	HolyWood	6737432.58	1873226.01	27.8	6437653.99	7037211.18
60	Vegas	5849663.58	1599060.15	27.34	5593760.66	6105566.5
	Westwood	3888404.77	4402173.39	113.21	3183910.3	4592899.23
	HolyWood	7011795.5	1779664.56	25.38	6726989.85	7296601.14
70	Vegas	5853965.56	1615523.32	27.6	5595427.99	6112503.14
	Westwood	3855620.66	4472059.89	115.99	3139942.03	4571299.3
	HolyWood	7015578.28	1777549.77	25.34	6731111.07	7300045.49
80	Vegas	5858341.72	1601653.08	27.34	5602023.85	6114659.6
	Westwood	3842492.19	4292188.91	111.7	3155598.9	4529385.47
	HolyWood	7019286.89	1779583.9	25.35	6734494.15	7304079.63
90	Vegas	5859825.17	1615703.18	27.57	5601258.81	6118391.52
	Westwood	3880542.52	4372591.87	112.68	3180782.08	4580302.95
	HolyWood	7023069.67	1776242.47	25.29	6738811.67	7307327.67
100	Vegas	5862643.71	1603188.46	27.35	5606080.12	6119207.3
	Westwood	3853618.01	4400213.27	114.18	3149437.23	4557798.8
	HolyWood	7026407.42	1779231.04	25.32	6741671.15	7311143.68

A.2.4 Statistics of Average Jitter Versus Error Rate of TCP HolyWood with other Protocols

Table A.2.4.1: Statistics of Average Jitter versus Error Rate of TCP HolyWood with other Protocols. (1 of 2)

Error Rate [%]	Protocol [TCP]	Average Jitter [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
0	Vegas	2.272	1.399	61.586	2.048	2.496
	Westwood	2.554	3.957	154.943	1.921	3.187
	HolyWood	2.251	1.108	49.238	2.074	2.428
0.01	Vegas	2.315	1.618	69.907	2.056	2.574
	Westwood	2.671	9.297	348.092	1.183	4.159
	HolyWood	2.251	1.108	49.249	2.073	2.428
0.02	Vegas	2.334	1.624	69.581	2.074	2.593
	Westwood	2.794	12.844	459.747	0.738	4.849
	HolyWood	2.252	1.114	49.491	2.073	2.43
0.04	Vegas	2.392	1.911	79.889	2.086	2.698
	Westwood	2.982	18.026	604.503	0.097	5.867
	HolyWood	2.281	4.41	193.331	1.575	2.987
0.06	Vegas	2.452	2.045	83.41	2.125	2.779
	Westwood	2.941	15.886	540.084	0.399	5.484
	HolyWood	2.258	1.15	50.928	2.074	2.442
0.08	Vegas	2.525	2.321	91.906	2.154	2.896
	Westwood	3.116	19.303	619.502	0.027	6.205
	HolyWood	2.26	1.168	51.705	2.073	2.447
0.1	Vegas	2.627	2.896	110.235	2.164	3.091
	Westwood	3.111	18.76	603.088	0.108	6.113
	HolyWood	2.29	1.431	62.48	2.061	2.519
0.2	Vegas	3.015	3.899	129.308	2.391	3.639
	Westwood	3.35	21.954	655.406	-0.164	6.863
	HolyWood	2.52	2.012	79.848	2.198	2.842
0.4	Vegas	3.782	5.699	150.688	2.87	4.694
	Westwood	3.639	25.92	712.293	-0.509	7.787
	HolyWood	3.346	4.707	140.671	2.593	4.1
0.6	Vegas	4.743	8.174	172.328	3.435	6.051
	Westwood	3.657	25.486	696.997	-0.422	7.735
	HolyWood	3.346	4.707	140.671	2.593	4.1
0.8	Vegas	5.511	10.513	190.746	3.829	7.194
	Westwood	3.625	25.52	703.976	-0.459	7.709
	HolyWood	5.169	9.92	191.905	3.582	6.757
1	Vegas	6.447	12.373	191.913	4.467	8.427
	Westwood	3.504	22.931	654.396	-0.166	7.174
	HolyWood	6.059	12.283	202.725	4.093	8.025

Table A.2.4.2: Statistics of Average Jitter versus Error Rate of TCP HolyWood with other Protocols. (2 of 2)

Error Rate [%]	Protocol [TCP]	Average Jitter [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
2	Vegas	8.87	23.08	260.2996	5.17	12.56
	Westwood	3.95	18.62	471.2044	0.97	6.93
	HolyWood	8.66	21.87	252.5013	5.16	12.16
4	Vegas	11.32	36.26	320.23	5.52	17.13
	Westwood	5.49	34.59	630.49	-0.05	11.02
	HolyWood	10.17	41	403.19	3.61	16.73
6	Vegas	10.64	76.08	714.77	-1.53	22.82
	Westwood	5.68	50.57	889.76	-2.41	13.78
	HolyWood	11.37	91.8	807.7	-3.33	26.06
8	Vegas	6.82	139.79	2050.45	-15.55	29.19
	Westwood	4.07	68.74	1687.64	-6.93	15.07
	HolyWood	19.2	252.43	1314.8	-21.2	59.6
10	Vegas	-0.18	181.46	-102705.9	-29.22	28.86
	Westwood	2.1	109.26	5206.03	-15.39	19.58
	HolyWood	6.77	91.52	1351.28	-7.87	21.42
20	Vegas	-89.01	702.38	-789.09	-201.42	23.39
	Westwood	-81.51	607.29	-745.06	-178.7	15.68
	HolyWood	23.62	551.26	2333.79	-64.6	111.84
40	Vegas	-1121.35	7418.68	-661.58	-2308.59	65.88
	Westwood	8.69	3696.83	42536.96	-582.92	600.31
	HolyWood	-111.43	872.94	-783.39	-251.13	28.27
60	Vegas	8025.55	36700.85	457.3	2152.19	13898.91
	Westwood	-139.13	7202.57	-5176.73	-1291.78	1013.52
	HolyWood	-187.86	921.8	-490.68	-335.38	-40.34

Table A.2.4.3: Ratio with TCP HolyWood as a Base for Statistics of Average Jitter versus Error Rate of TCP HolyWood with other Protocols.

Error Rate [%]	Jitter TCP [ms] HolyWood	Jitter TCP [ms] Westwood	Jitter TCP [ms] Vegas	TCP [ms] HolyWood as a Base	TCP [ms] abs(Westwood/ HolyWood)	TCP [ms] abs(Vegas/ HolyWood)
0.0001~0	2.2509	2.5538	2.2718	1	1.1346	1.0093
0.01	2.2507	2.6709	2.3147	1	1.1867	1.0284
0.02	2.2517	2.7937	2.3335	1	1.2407	1.0363
0.04	2.2809	2.9819	2.3923	1	1.3074	1.0488
0.06	2.2582	2.9413	2.4521	1	1.3025	1.0859
0.08	2.2596	3.1158	2.5249	1	1.3789	1.1174
0.1	2.29	3.1107	2.6274	1	1.3584	1.1473
0.2	2.5198	3.3497	3.0153	1	1.3294	1.1966
0.4	3.3464	3.639	3.7822	1	1.0874	1.1302
0.6	4.2121	3.6565	4.7433	1	0.8681	1.1261
0.8	5.1694	3.6252	5.5114	1	0.7013	1.0662
1	6.059	3.5041	6.4471	1	0.5783	1.0641
2	8.661	3.9515	8.8672	1	0.4562	1.0238
4	10.1686	5.4855	11.3241	1	0.5395	1.1136
6	11.3662	5.6831	10.6435	1	0.5	0.9364
8	19.1988	4.0732	6.8175	1	0.2122	0.3551
10	6.773	2.0987	-0.1767	1	0.3099	0.0261
20	23.6207	-81.5089	-89.0115	1	3.4507	3.7684
40	-111.431	8.6909	-1121.35	1	0.078	10.0632
60	-187.86	-139.133	8025.55	1	0.7406	42.7209

A.2.5 Statistics of Average Jitter Versus Propagation Time of TCP HolyWood with other Protocols

Table A.2.5.1: Statistics of Average Jitter versus Propagation Time of TCP HolyWood with other Protocols

Propagation Time [ms]	Protocol [TCP]	Average Jitter [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
0.1	Vegas	2.237	0.586	26.214	2.143	2.33
	Westwood	2.312	8.719	377.06	0.917	3.708
	HolyWood	2.233	1.366	61.173	2.014	2.451
1	Vegas	2.238	0.597	26.671	2.142	2.333
	Westwood	2.347	9.229	393.309	0.87	3.824
	HolyWood	2.233	1.37	61.374	2.013	2.452
5	Vegas	2.254	0.679	30.125	2.145	2.362
	Westwood	2.478	14.859	599.589	0.1	4.856
	HolyWood	2.241	1.922	85.773	1.933	2.548
10	Vegas	2.258	0.768	34.03	2.135	2.381
	Westwood	2.357	8.774	372.309	0.953	3.761
	HolyWood	2.244	1.961	87.368	1.931	2.558
50	Vegas	3.16	5.053	159.884	2.352	3.969
	Westwood	3.957	24.928	629.949	-0.032	7.946
	HolyWood	2.514	3.129	124.46	2.013	3.015
100	Vegas	8.421	54.024	641.549	-0.225	17.066
	Westwood	6.404	33.692	526.154	1.012	11.795
	HolyWood	4.687	14.009	298.859	2.446	6.929
200	Vegas	16.961	113.853	671.261	-1.259	35.181
	Westwood	10.849	64.701	596.362	0.495	21.204
	HolyWood	9.548	29.553	309.516	4.819	14.278
300	Vegas	25.56	143.804	562.61	2.547	48.574
	Westwood	14.051	79.733	567.475	1.291	26.81
	HolyWood	15.546	47.443	305.183	7.953	23.138
400	Vegas	44.543	251.774	565.241	4.251	84.835
	Westwood	18.764	107.875	574.909	1.5	36.028
	HolyWood	19.414	63.826	328.766	9.2	29.628
500	Vegas	54.687	318.583	582.558	3.703	105.671
	Westwood	24.184	140.332	580.266	1.726	46.642
	HolyWood	23.327	80.254	344.038	10.484	36.17
1000	Vegas	147.442	860.313	583.492	9.763	285.121
	Westwood	63.886	325.262	509.131	11.833	115.939
	HolyWood	50.311	207.03	411.5	17.179	83.443

Table A.2.5.2: Ratio with TCP HolyWood as a Base for Statistics Average Jitter versus Propagation Time of TCP HolyWood with other Protocols

Propagation Time [ms]	Jitter TCP [ms] HolyWood	Jitter TCP [ms] Westwood	Jitter TCP [ms] Vegas	TCP [ms] HolyWood as a Base	TCP [ms] Westwood/ HolyWood	TCP [ms] Vegas/ HolyWood
0.1	2.2325	2.3122	2.2365	1	1.0357	1.0018
1	2.2326	2.3465	2.2376	1	1.051	1.0022
5	2.2405	2.4782	2.2537	1	1.1061	1.0059
10	2.2443	2.3566	2.2577	1	1.05	1.0059
50	2.5142	3.9571	3.1601	1	1.5739	1.2569
100	4.6874	6.4035	8.4208	1	1.3661	1.7965
200	9.5481	10.8492	16.9611	1	1.1363	1.7764
300	15.5459	14.0505	25.5602	1	0.9038	1.6442
400	19.4138	18.764	44.5427	1	0.9665	2.2944
500	23.3271	24.184	54.6869	1	1.0367	2.3444
1000	50.3111	63.8858	147.442	1	1.2698	2.9306

A.2.6 Statistics of Average Jitter Versus Bandwidth of TCP HolyWood with other Protocols

Table A.2.6.1: Statistics of Average Jitter versus Bandwidth of TCP HolyWood with other Protocols

Bottleneck Bandwidth [Mb]	Protocol [TCP]	Average Jitter [ms]	Standard Dev. [ms]	COV [%]	C1 (95%) [ms]	C2 (95%) [ms]
1	Vegas	11.238	2.525	22.468	10.834	11.642
	Westwood	11.366	10.236	90.062	9.728	13.004
	HolyWood	11.217	4.86	43.327	10.439	11.994
10	Vegas	2.027	4.527	223.289	1.303	2.752
	Westwood	2.633	23.847	905.763	-1.184	6.449
	HolyWood	1.678	5.997	357.417	0.718	2.638
20	Vegas	1.999	9.615	480.924	0.461	3.538
	Westwood	1.564	9.828	628.477	-0.009	3.137
	HolyWood	1.636	7.117	435.027	0.497	2.775
30	Vegas	1.955	10.238	523.802	0.316	3.593
	Westwood	1.414	9.075	641.654	-0.038	2.867
	HolyWood	1.626	8.263	508.058	0.304	2.949
40	Vegas	1.879	9.338	497.106	0.384	3.373
	Westwood	1.354	9.518	703.029	-0.169	2.877
	HolyWood	1.627	9.478	582.666	0.11	3.144
50	Vegas	1.874	9.781	521.828	0.309	3.44
	Westwood	1.557	9.385	602.772	0.055	3.059
	HolyWood	1.622	9.435	581.595	0.112	3.132
60	Vegas	1.872	10.068	537.874	0.261	3.483
	Westwood	1.213	8.274	682.399	-0.112	2.537
	HolyWood	1.564	9.402	601.107	0.06	3.069
70	Vegas	1.872	9.878	527.65	0.291	3.453
	Westwood	1.216	8.428	692.983	-0.133	2.565
	HolyWood	1.563	9.643	616.908	0.02	3.106
80	Vegas	1.871	10.084	539.048	0.257	3.485
	Westwood	1.167	9.96	853.836	-0.427	2.761
	HolyWood	1.562	9.826	628.943	-0.01	3.135
90	Vegas	1.87	10.249	547.941	0.23	3.511
	Westwood	1.176	8.516	724.03	-0.187	2.539
	HolyWood	1.562	9.97	638.392	-0.034	3.157
100	Vegas	1.869	10.378	555.274	0.208	3.53
	Westwood	1.19	8.636	725.558	-0.192	2.572
	HolyWood	1.561	10.084	645.897	-0.053	3.175

Table A.2.6.2: Ratio with TCP HolyWood as a Base for Statistics of Average Jitter versus Bandwidth of TCP HolyWood with other Protocols

Bottleneck Bandwidth [Mb]	Jitter TCP [ms] HolyWood	Jitter TCP [ms] Westwood	Jitter TCP [ms] Vegas	TCP [ms] HolyWood as a Base	TCP [ms] Westwood/ HolyWood	TCP [ms] Vegas/ HolyWood
1	11.2165	11.3657	11.2383	1	1.0133	1.0019
10	1.6779	2.6328	2.0275	1	1.5692	1.2084
20	1.636	1.5639	1.9993	1	0.9559	1.222
30	1.6263	1.4142	1.9546	1	0.8696	1.2018
40	1.6267	1.3538	1.8785	1	0.8322	1.1548
50	1.6222	1.5571	1.8743	1	0.9599	1.1554
60	1.5641	1.2125	1.8718	1	0.7752	1.1967
70	1.5631	1.2161	1.872	1	0.778	1.1976
80	1.5624	1.1665	1.8707	1	0.7467	1.1974
90	1.5617	1.1762	1.8704	1	0.7531	1.1977
100	1.5613	1.1902	1.869	1	0.7623	1.1971

A.3.1 Percentage of Lost Packets of TCP HolyWood versus TCP Reno

Table A.3.1.1: List of Packets Sent and Lost of TCP HolyWood and TCP Reno.

Error Rate [%]	TCP HolyWood Sent Packets	TCP HolyWood Lost Packets	TCP Reno Sent Packets	TCP Reno Lost Packets
0.0001~0	100658	132	99161	78
0.01	100683	133	94408	112
0.02	100565	134	92442	111
0.04	99350	136	85546	105
0.06	100401	136	80924	82
0.08	100223	138	74206	109
0.1	98928	136	68747	111
0.2	89778	176	54305	112
0.4	67152	184	37269	108
0.6	52811	209	27977	123
0.8	42777	224	21233	128
1	35759	281	18596	137
2	21950	391	11386	147
4	14235	547	5041	144
6	8765	389	3753	137
8	1539	136	2999	138
10	6614	417	2359	138
20	2789	355	1023	129
40	748	181	200	50
60	270	118	77	31

Table A.3.1.2: Ratio with TCP HolyWood as a Base Percentage of Lost Packets of TCP HolyWood versus TCP Reno

Error Rate [%]	TCP HolyWood Lost Packets [%]	TCP Reno Lost Packets [%]	TCP HolyWood As a base [%]	TCP (Reno/ HolyWood) [%]
0.0001~0	0.1311	0.0787	1	0.5998
0.01	0.1321	0.1186	1	0.8981
0.02	0.1332	0.1201	1	0.9011
0.04	0.1369	0.1227	1	0.8966
0.06	0.1355	0.1013	1	0.7481
0.08	0.1377	0.1469	1	1.0668
0.1	0.1375	0.1615	1	1.1745
0.2	0.196	0.2062	1	1.0521
0.4	0.274	0.2898	1	1.0576
0.6	0.3958	0.4396	1	1.1109
0.8	0.5236	0.6028	1	1.1512
1	0.7858	0.7367	1	0.9375
2	1.7813	1.2911	1	0.7248
4	3.8426	2.8566	1	0.7434
6	4.4381	3.6504	1	0.8225
8	8.8369	4.6015	1	0.5207
10	6.3048	5.8499	1	0.9279
20	12.7286	12.61	1	0.9907
40	24.1979	25	1	1.0331
60	43.7037	40.2597	1	0.9212

A.3.2 Percentage of Lost Packets of TCP HolyWood versus other Protocols

Table A.3.2.1: List of Sent andLost Packets of TCP HolyWood versus other Protocols

Error Rate [%]	TCP HolyWood Sent Packets	TCP HolyWood Lost Packets	TCP Westwood Sent Packets	TCP WestWood Lost Packets	TCP Vegas Sent Packets	TCP Vegas Lost Packets
0.0001~0	100658	132	79753	179	99035	28
0.01	100683	133	76750	234	97249	39
0.02	100565	134	73078	234	96535	50
0.04	99350	136	70119	245	94184	61
0.06	100401	136	69525	251	91935	70
0.08	100223	138	66536	256	89303	77
0.1	98928	136	65547	251	85700	93
0.2	89778	176	59838	297	74781	123
0.4	67152	184	54535	267	59481	154
0.6	52811	209	51199	270	47333	193
0.8	42777	224	52395	351	40295	217
1	35759	281	51270	376	34250	235
2	21950	391	35749	446	22295	303
4	14235	547	18511	435	14132	356
6	8765	389	11926	398	8593	310
8	1539	136	7850	346	4627	220
10	6614	417	5248	290	3547	203
20	2789	355	1196	147	1204	149
40	748	181	200	50	211	53
60	270	118	77	31	16	8

Table A.3.2.2: Ratio with TCP HolyWood as a Base Percentage of Lost Packets of TCP HolyWood Vs. Other Protocols

Error Rate [%]	TCP HolyWood Lost Packets [%]	TCP Westwood Lost Packets [%]	TCP Vegas Lost Packets [%]	TCP HolyWood As a base [%]	TCP (Westwood/ HolyWood) [%]	TCP (Vegas/ HolyWood) [%]
0.0001	0.1311	0.2244	0.0283	1	1.7115	0.2156
0.01	0.1321	0.3049	0.0401	1	2.308	0.3036
0.02	0.1332	0.3202	0.0518	1	2.4031	0.3887
0.04	0.1369	0.3494	0.0648	1	2.5525	0.4731
0.06	0.1355	0.361	0.0761	1	2.6652	0.5621
0.08	0.1377	0.3848	0.0862	1	2.7943	0.6262
0.1	0.1375	0.3829	0.1085	1	2.7855	0.7894
0.2	0.196	0.4963	0.1645	1	2.5318	0.839
0.4	0.274	0.4896	0.2589	1	1.7868	0.9449
0.6	0.3958	0.5274	0.4077	1	1.3325	1.0303
0.8	0.5236	0.6699	0.5385	1	1.2793	1.0284
1	0.7858	0.7334	0.6861	1	0.9333	0.8731
2	1.7813	1.2476	1.3591	1	0.7004	0.7629
4	3.8426	2.35	2.5191	1	0.6115	0.6556
6	4.4381	3.3373	3.6076	1	0.752	0.8129
8	8.8369	4.4076	4.7547	1	0.4988	0.5381
10	6.3048	5.5259	5.7232	1	0.8765	0.9077
20	12.7286	12.291	12.3754	1	0.9656	0.9723
40	24.1979	25	25.1185	1	1.0331	1.038
60	43.7037	40.2597	50	1	0.9212	1.1441

APPENDIX B COMPUTER CODES

B.1 Main Code of TCP HolyWood for ns-2.1b8a

TCP-holywood.cc

```
// AD MAJOREM DEI GLORIAM
//
//   TCP HOLYWOOD ns-2.1b8a Code
//
//   TCP-holywood.cc
//
//\\   Oscar Núñez Mori (PERU)
//   Copyright (c) 2004, 2005.
//   All rights reserved.
//\\
//   Copyright (c) 1990, 1997 Regents of the University of California.
//   All rights reserved.
//   Redistribution and use in source and binary forms are permitted
// \\   provided that the above copyright notice and this paragraph are
//   duplicated in all such forms and that any documentation,
//   advertising materials, and other materials related to such
//\\   distribution and use acknowledge that the software was developed
//   by the University of California, Lawrence Berkeley Laboratory,
//   Berkeley, CA.  The name of the University may not be used to
//\\   endorse or promote products derived from this software without
//   specific prior written permission.
//   THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY
//   EXPRESS  OR IMPLIED WARRANTIES, INCLUDING, WITHOUT
//   LIMITATION, THE  IMPLIED WARRANTIES OF MERCHANTIBILITY
//   AND FITNESS FOR A  PARTICULAR  PURPOSE.

#ifndef lint
static const char rcsid[] =
"@(#) $Header: /usr/home/oscar/ns-allinone-2.1b8a/ns-2.1b8a/TCP-holywood.cc,v
00.00 2004/08/15 00:13 ONmori
#endif

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
#include "ip.h"
#include "TCP.h"
#include "flags.h"
#include "TCP-holywood.h"

static class HolyWoodTCPClass : public TclClass {
public:
        HolyWoodTCPClass() : TclClass("Agent/TCP/HolyWood") {}
        TclObject* create(int, const char*const*) {
                return (new HolyWoodTCPAgent());
        }
} class_holywood;

int HolyWoodTCPAgent::window()
{
           // the same as TCP-reno.cc
          // reno: inflate the window by dupwnd_
          //      dupwnd_ will be non-zero during fast recovery,
         //        at which time it contains the number of dup acks
        //
        int win = int(cwnd_) + dupwnd_;
        if (win > int(wnd_))
                win = int(wnd_);
        return (win);
}

double HolyWoodTCPAgent::windowd()
{
          //The same as TCP-reno.cc
          // reno: inflate the window by dupwnd_
         //       dupwnd_ will be non-zero during fast recovery,
        //              at which time it contains the number of dup acks
       //
        double win = cwnd_ + dupwnd_;
        if (win > wnd_)
                win = wnd_;
        return (win);
}

HolyWoodTCPAgent::HolyWoodTCPAgent() : TCPAgent(), dupwnd_(0)
{
}

 //
//

void HolyWoodTCPAgent::recv(Packet *pkt, Handler*)
{
        hdr_TCP *TCPh = hdr_TCP::access(pkt);
#ifdef notdef
        if (pkt->type_ != PT_ACK) {
                fprintf(stderr,
```

```
                        "ns: confiuration error: TCP received non-ack\n");
                exit(1);
        }
#endif
        ++nackpack_;
        ts_peer_ = TCPh->ts();

        if (hdr_flags::access(pkt)->ecnecho() && ecn_)
                ecn(TCPh->seqno());
        recv_helper(pkt);
        if (TCPh->seqno() > last_ack_) {
                dupwnd_ = 0;
                recv_newack_helper(pkt);
                if (last_ack_ == 0 && delay_growth_) {
                        cwnd_ = initial_window();
                }
        } else if (TCPh->seqno() == last_ack_) {
                if (hdr_flags::access(pkt)->eln_ && eln_) {
                        TCP_eln(pkt);
                        return;
                }
                if (++dupacks_ == numdupacks_) {
                        dupack_action();
                        dupwnd_ = numdupacks_;
                } else if (dupacks_ > numdupacks_) {
                        ++dupwnd_;   // fast recovery
                } else if (dupacks_ < numdupacks_ && singledup_ ) {
                        send_one();
                }
        }
        Packet::free(pkt);
#ifdef notyet
        if (trace_)
                plot();
#endif

         //
        // Try to send more data
         //

        if (dupacks_ == 0 || dupacks_ > numdupacks_ - 1)
                send_much(0, 0, maxburst_);
}

 //
//

int
HolyWoodTCPAgent::allow_fast_retransmit(int last_cwnd_action_)
{
        return (last_cwnd_action_ == CWND_ACTION_DUPACK);
}
```

```
//
// Dupack-action: what to do on a DUP ACK.  After the initial check
//   of 'recover' below, this function implements the following truth
//    Table:
//
//\\  bugfix   ecn   last-cwnd == ecn     action
//
//      0      0      0              HolyWood_action
//\\    0      0      1              HolyWood_action    [impossible]
//      0      1      0              HolyWood_action
//      0      1      1              retransmit, return
//\\    1      0      0              nothing
//      1      0      1              nothing       [impossible]
//      1      1      0              nothing
//      1      1      1              retransmit, return


void
HolyWoodTCPAgent::dupack_action()
{
        int recovered = (highest_ack_ > recover_);
        int allowFastRetransmit = allow_fast_retransmit(last_cwnd_action_);
    if (recovered || (!bug_fix_ && !ecn_) || allowFastRetransmit) {
                goto HolyWood_action;
        }

        if (ecn_ && last_cwnd_action_ == CWND_ACTION_ECN) {
                last_cwnd_action_ = CWND_ACTION_DUPACK;
                //
                 // What if there is a DUPACK action followed closely by ECN
                // followed closely by a DUPACK action?
               // The optimal thing to do would be to remember all
              // congestion actions from the most recent window
             // of data.  Otherwise "bugfix" might not prevent
            // all unnecessary Fast Retransmits.
           //
                reset_rtx_timer(1,0);
                output(last_ack_ + 1, TCP_REASON_DUPACK);
                return;
        }

        if (bug_fix_) {
                 //
               // The line below, for "bug_fix_" true, avoids
              // problems with multiple fast retransmits in one
             // window of data.
            //
                return;
        }

HolyWood_action:
```

```
        // we are now going to fast-retransmit and will trace that event
        trace_event("RENO_FAST_RETX");
        recover_ = maxseq_;
        last_cwnd_action_ = CWND_ACTION_DUPACK;
         // 50 % is :
           // slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_HALF);
          //  Added
         // 83.33 % is :
        slowdown(CLOSE_SSTHRESH_FIVE_SIXTH|CLOSE_CWND_FIVE_SIXTH
);

        reset_rtx_timer(1,0);
        output(last_ack_ + 1, TCP_REASON_DUPACK);   // from top
        return;
}


void HolyWoodTCPAgent::timeout(int tno)
{
        // retransmit timer
        if (tno == TCP_TIMER_RTX) {

                // There has been a timeout - will trace this event
                trace_event("TIMEOUT");

                  // if (cwnd_ < 1) cwnd_ = 1;
                // added
            if (cwnd_ < 3) cwnd_ = 3;
                if (highest_ack_ == maxseq_ && !slow_start_restart_) {
                           // TCP option:
                       // If no outstanding data, then don't do anything.
                     // Should this return be here?
                   // What if CWND_ACTION_ECN and cwnd < 1?
                 // return;
                } else {
                        recover_ = maxseq_;
                        if (highest_ack_ == -1 && wnd_init_option_ == 2)
                                   //
                                  // First packet dropped, so don't use larger
                                   // initial windows.
                                 //
                              wnd_init_option_ = 1;
                        if (highest_ack_ == maxseq_ && restart_bugfix_)
                                   //
                                    // if there is no outstanding data, don't cut
                                // down ssthresh_.
                               //
                             //  slowdown(CLOSE_CWND_ONE);
                                   //
                   slowdown(CLOSE_CWND_THREE);

                        else if (highest_ack_ < recover_ &&
```

```
                last_cwnd_action_ == CWND_ACTION_ECN) {
                        //
                        // if we are in recovery from a recent ECN,
                        // don't cut down ssthresh_.
                        //
                        //slowdown(CLOSE_CWND_ONE);
                        //
                        slowdown(CLOSE_CWND_THREE);
                }
                else {
                        ++nrexmit_;
                        last_cwnd_action_ = CWND_ACTION_TIMEOUT;

                //
slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_RESTART);
                // 65% of slow start threshold and start for 3

        slowdown(CLOSE_SSTHRESH_THIRTEEN_TWENTIETH|CLOSE_CWND_
THREE);
                }
            }
            // if there is no outstanding data, don't backoff rtx timer
            if (highest_ack_ == maxseq_ && restart_bugfix_) {
                reset_rtx_timer(0,0);
            }
            else {
                reset_rtx_timer(0,1);
            }
            last_cwnd_action_ = CWND_ACTION_TIMEOUT;
            send_much(0, TCP_REASON_TIMEOUT, maxburst_);
        }
        else {
            timeout_nonrtx(tno);
        }
}

void HolyWoodTCPAgent::opencwnd()
{
        double increment;
        if (cwnd_ < ssthresh_) {
            // slow-start (exponential)
            // cwnd_ += 1;
          // fast-starter
          cwnd_ += 1.8;

        } else {
            //   reducing the slope of Congestion Avoidance
            //    increment_num_ default is 1
          //   This is the standard algorithm. */
         //     cwnd_ += increase_num_ / cwnd_;
        //
        cwnd_ += increase_num_ /(4 * cwnd_);
```

```
        }

    //maxcwnd_ is maximum number of congestion window that can ever be
   // if maxcwnd_ is set (nonzero), make it the cwnd limit
  // but maxcwnd_ is 0 (zero) by default in ns-default.tcl. Zero means false
    if (maxcwnd_ && (int(cwnd_) > maxcwnd_))
            cwnd_ = maxcwnd_;

    return;
}

void
HolyWoodTCPAgent::slowdown(int how)
{
    double fiveSixthWin, thirteenTwentiethWin, decreasewin;
    int slowstart = 0;
    // we are in slowstart for sure if cwnd < ssthresh
      if (cwnd_ < ssthresh_) {
        // a switch
        slowstart = 1;
      // Because we are in slowstart - need to trace this event.
        trace_event("SLOW_START");
    }

    if (precision_reduce_) {

            fiveSixthWin = 5 * windowd() / 6;

        thirteenTwentiethWin = 13 * windowd() / 20;

          decreasewin = decrease_num_ * windowd();

      } else  {
              int temp;
                //
                  // --- added
            temp = (int)(5 * window() / 6);
              fiveSixthWin = (double) temp;

            temp = (int)(13 * window() / 20);
              thirteenTwentiethWin = (double) temp;
             //
              // decrease_num_ is a factor for multiplicative decrease
             //  decrease_num_ default is 0.5
          temp = (int)(decrease_num_ * window());
              decreasewin = (double) temp;
         }

  if (how & CLOSE_SSTHRESH_FIVE_SIXTH)
             // For the first decrease, decrease by three Fifth
            // even for non-standard values of decrease_num_.
```

```
            if (first_decrease_ == 1 || slowstart ||
            last_cwnd_action_ == CWND_ACTION_TIMEOUT) {
                     // Do we really want fiveSixthWin  instead of decreasewin
                      // in fast recovery algorithm ?
                    ssthresh_ = (int) fiveSixthWin;
            } else {
                    ssthresh_ = (int) decreasewin;
            }

    if (how & CLOSE_SSTHRESH_THIRTEEN_TWENTIETH)
              // For the first decrease, decrease by thirteen twentienth (65%)
            // even for non-standard values of decrease_num_.
            if (first_decrease_ == 1 || slowstart ||
                    last_cwnd_action_ == CWND_ACTION_TIMEOUT) {
                     // Do we really want thirteen twentieth instead of decreasewin
                      // after a timeout?
                    ssthresh_ = (int) thirteenTwentiethWin;
            } else {
                    ssthresh_ = (int) decreasewin;
            }

    if (how & CLOSE_CWND_FIVE_SIXTH)
              // For the first decrease, decrease by five sixth
            // even for non-standard values of decrease_num_.
                 //  The default value of decrease_num is 0.5 in ns-2
            if (first_decrease_ == 1 || slowstart || decrease_num_ == 0.5) {
                    cwnd_ = fiveSixthWin; // added by oscar Nunez Mori
            } else cwnd_ = decreasewin;

    else if (how & CLOSE_CWND_THREE)
            cwnd_ = 3; // added by Oscar Nunez Mori

     // if (ssthresh_ < 2)
   //       ssthresh_ = 2;
 //  added by Oscar Nunez Mori
if (ssthresh_ < 3)
       ssthresh_ = 3;

    if (how & (CLOSE_CWND_FIVE_SIXTH|CLOSE_CWND_THREE))
        // cong_action: Congestion Action is True to indicate
      // that the sender responded to congestion
         cong_action_ = TRUE;


//fcnt_ and count_ are used in window increment algorithms
    fcnt_ = count_ = 0;
    if (first_decrease_ == 1)
         //  First decrease of congestion window.
        //  Used for decrease_num_ != 0.5.
      // first_decrease_ is 1 by default defined in TCP.cc line 78
            first_decrease_ = 0;
```

```
}

     //
   // Check if the sender has been idle or application-limited for more
  // than an RTO, and if so, reduce the congestion window.
//
void HolyWoodTCPAgent::process_qoption_after_send ()
{
          // we intentionally clean the properties of this function
        // because when the sender transmits again or after the
      // idleness of the sender finishes, we do not want a
    // reduced congestion window, so the performance
  // will not be diminished
}

      // Check if the sender has been idle or application-limited for more
    // than an RTO, and if so, reduce the congestion window, for a TCP sender
  // that "counts RTTs" by estimating the number of RTTs that fit into
 // a single clock tick.
void
HolyWoodTCPAgent::rtt_counting()
{
      // we intentionally clean the properties of this function
     // because when the sender transmits again or after the
    // idleness of the sender finishes, we do not want a
   // reduced congestion window, so the performance
  // will not be diminished
}
```

B.2 Addendum to Main Code of TCP HolyWood for ns-2.1b8a

TCP-holywood.h

```
    //  DEO GRATIAS
  //
 //\\  TCP-holywood.h   2004/08/15
     //
   //    Oscar Núñez Mori (PERU)
  //
//\\   Copyright (c)  2004, 2005. All right reserved
    //
  //    @(#) $Header: /onunezmori/TCP-holywood.h,v 0.1 2004/08/15 00:13:00
onunezmori


#ifndef TCP_holywood_h
#define TCP_holywood_h

#include "TCP.h"
  // TCP HOLY WOOD
 //
// bits to modify the ssthresh and cwnd
```

```
#define CLOSE_SSTHRESH_FIVE_SIXTH                   0x00000800
#define CLOSE_CWND_FIVE_SIXTH                       0x00001000
#define CLOSE_CWND_THREE                            0x00002000
#define CLOSE_SSTHRESH_THIRTEEN_TWENTIETH  0x00004000

class HolyWoodTCPAgent : public virtual TCPAgent {
 public:
        HolyWoodTCPAgent();
        virtual int window();
        virtual double windowd();
        virtual void recv(Packet *pkt, Handler*);
        virtual void timeout(int tno);
        virtual void dupack_action();
        // added and modified
        // these where originally in class TCPAgent (file: TCP.h)
        virtual void opencwnd();
        virtual void slowdown(int how);
        virtual void process_qoption_after_send ();
        virtual void rtt_counting();
 protected:
        int allow_fast_retransmit(int last_cwnd_action_);
        unsigned int dupwnd_;
};

#endif // TCP_holywood_h
```

B.3 Base Script: test-1-simple.tcl

```
# Original Script from TCP-Westwood.  CS Department.
# University of California. Los Angeles
#  Available at: <http://www.cs.ucla.edu/NRL/hpi/tcpw/> . Visited Feb. 2004.
        #
        ### Default Values
        set proto "Westwood"
        set buffer 0
        set psize 1400
        set lrate 0.0

        ### Read command line arguments
        If {$argc > 1} {
        set proto  [lindex $argv 0]
        set buffer [lindex $argv 1]
        set lrate  [lindex $argv 2]
        } else {
        puts "usage: ns test-1-simple.tcl <protocol> <buffer> <error rate>"
        puts " "
        puts "<protocol> is Reno, Newreno, Westwood or WestwoodNR"
        puts "<buffer> is the buffer size, use 0 for automatic pipe size setting"
        puts "<erro rate> is the link error rate (0.001 = 0.1%). Use 0 for no errors"

        exit 1
        }

        set ns [new Simulator]

        $ns color 1 Blue
        $ns color 2 Red

        set tr_f       [open out.tr w]
```

```
set nam_f      [open out.nam w]
set record_f   [open out.data w]

$ns trace-all $tr_f
$ns namtrace-all $nam_f

### Finish proc
proc finish {} {
global ns tr_f record_f nam_f
$ns flush-trace
close $tr_f
close $record_f
close $nam_f

exec awk { { print $1, $2 } } out.data > temp.cwnd
exec awk { { print $1, $3 } } out.data > temp.sst
exec awk { { print $1, $4 } } out.data > temp.bwe

exec xgraph temp.cwnd temp.sst -m -x time -y seq_no -geometry 600x200 &
exec xgraph temp.bwe -m -x time -y bit/s -geometry 600x200 &

exec gnuplot cwnd_and_sst.g &
exec gnuplot bwe.g &

exit 0
}


# Network topology 1 :
#
#
#   100Mbit/s, 1ms      5Mbit/s, 35 ms        100Mbit/s, 1ms
#  n0 --------------- n1------------------- n2 ------------------ n3 TCP-Sink
# TCP Source         30 buffer
#
#
# backlogged FTP sources with 1400 bytes packet size

set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#   object     from  to  bandwith   delay    queue
$ns duplex-link $n0  $n1   100Mb      1ms   DropTail
$ns duplex-link $n2  $n3   100Mb     1ms   DropTail

$ns duplex-link $n1  $n2     5Mb     35ms   DropTail

#### BOTTLENECK BUFFER
set rtd 72
set bneck_bw 5
# Set the queue size (queue size is in packets, default is 50)
# We should set the buffer capacity equal to the pipe size:
# buffer = bneck_bw * round_trip_delay / 8 / bytes_per_packet
set buffer_calc [expr $bneck_bw*1000000 * $rtd/1000 / 8 / $psize]
if {$buffer == 0} {  # no buffer provided
set buffer $buffer_calc
}
puts "Buffer: set:$buffer - calc: $buffer_calc"

$ns queue-limit $n1 $n2 [expr $buffer *1]

$ns duplex-link-op $n1 $n2 queuePos 0.5


#### BOTTLENECK LINK ERRORS
# Add bottleneck link errors
set lossy_link 0
if {$lrate > 0} { set lossy_link 1 }
```

```
if { $lossy_link == 1} {
        set loss_module [new ErrorModel]
        $loss_module unit pkt
        $loss_module set rate_ $lrate
        $loss_module ranvar [new RandomVariable/Uniform]
        $loss_module drop-target [new Agent/Null]

        $ns lossmodel $loss_module $n1 $n2
        puts ">>>ErrorModel: lossy_link: $lossy_link - rate: $lrate"
}

#### TCP AGENT
set TCP [new Agent/TCP/$proto]
puts "Proto used: $proto"
$TCP set fid_ 1
$TCP set tau_ 1.0
$TCP set filter_type_ 3
$TCP set window_ 100
$TCP set maxcwnd_ 2000
$TCP set packetSize_ $psize
$ns attach-agent $n0 $TCP

set sink [new Agent/TCPSink/DelAck]
$ns attach-agent $n3 $sink

$ns connect $TCP $sink

set ftp [new Application/FTP]
$ftp attach-agent $TCP

### Record proc
proc record {} {
global ns TCP sink record_f proto
set now [$ns now]

set time 0.15

set cwin     [$TCP set cwnd_]
set ssthresh [$TCP set ssthresh_]

if { ($proto == "Westwood") || ($proto == "WestwoodNR")} {
        set bwe  [$TCP set current_bwe_]
} else {
        set bwe  0
}

puts $record_f "$now [expr $cwin*1] $ssthresh $bwe"

 $ns at [expr $now+$time] "record "
}

$ns at 0.0 "record"
$ns at 0.0 "$ftp start"

$ns at 40.0 "finish"

$ns run
```

B.4 Script of Network topology 1: test-1-simple-HOLYWOOD-150s.tcl

```
# Script modified by Eng. Oscar  Nunez Mori  (PERU)
# this test  is base in TCP-Westwood script  test1-simple.tcl (B.3)
#
#
### Default Values

set proto "HolyWood"
set buffer 0
set psize 1400

set lrate 0.0

### Read command line arguments
if {$argc > 1} {
   set proto  [lindex $argv 0]
   set buffer [lindex $argv 1]
   set lrate  [lindex $argv 2]
} else {
   puts "usage: ns test-1-simple.tcl <protocol> <buffer> <error rate>"
   puts " "
   puts "<protocol> is Reno, Newreno, Westwood, HolyWood etc."
   puts "<buffer> is the buffer size, use 0 for automatic pipe size setting"
   puts "<erro rate> is the link error rate (0.001 = 0.1%). Use 0 for no errors"

   exit 1
}

set ns [new Simulator]

set tr_f       [open test-1-simple-150s-Err-$lrate-holywood.tr w]
set record_f   [open test-1-simple-150s-Err-0-holywood.data w]

$ns trace-all $tr_f

### Finish proc
proc finish {} {
   global ns tr_f record_f  ;# nam_f
    $ns flush-trace
    close $tr_f
    close $record_f


    exec awk { { print $1, $2 } } test-1-simple-150s-Err-0-holywood.data > test-1-
simple-150s-Err-0-holywood.cwnd
    exec awk { { print $1, $3 } } test-1-simple-150s-Err-0-holywood.data > test-1-
simple-150s-Err-0-holywood.sst
    exec awk { { print $1, $4 } } test-1-simple-150s-Err-0-holywood.data > test-1-
simple-150s-Err-0-holywood.bwe
```

```
    exec xgraph  test-1-simple-150s-Err-0-holywood.cwnd -m -x time -y seq_no -
geometry 600x200 &
    exec xgraph  test-1-simple-150s-Err-0-holywood.sst  -m -x time -y seq_no -geometry
600x200 &
    exec xgraph  test-1-simple-150s-Err-0-holywood.bwe  -m -x time -y bit/s    -
geometry 600x200 &
    exit 0
}

# Network topology 1:
#
#
#   100Mbit/s, 1ms      5Mbit/s, 35 ms      100Mbit/s, 1ms
#  n0 ----------- ---- n1 ------------------- n2 ----------------- n3 TCP-Sink
# TCP Source        30 buffer
#
#
# backlogged FTP sources with 1400 bytes packet size

set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#   object     from  to  bandwith   delay    queue
$ns duplex-link $n0  $n1   100Mb     1ms   DropTail
$ns duplex-link $n2  $n3   100Mb     1ms   DropTail
$ns duplex-link $n1  $n2      5Mb    35ms   DropTail

#### BOTTLENECK BUFFER
set rtd 72
set bneck_bw 5
# Set the queue size (queue size is in packets, default is 50)
# We should set the buffer capacity equal to the pipe size:
# buffer = bneck_bw * round_trip_delay / 8 / bytes_per_packet
set buffer_calc [expr $bneck_bw*1000000 * $rtd/1000 / 8 / $psize]
if {$buffer == 0} {  # no buffer provided
   set buffer $buffer_calc
}
puts "Buffer: set:$buffer - calc: $buffer_calc"

$ns queue-limit $n1 $n2 [expr $buffer *1]

$ns duplex-link-op $n1 $n2 queuePos 0.5


#### BOTTLENECK LINK ERRORS
# Add bottleneck link errors
set lossy_link 0
if {$lrate > 0} { set lossy_link 1 }

if { $lossy_link == 1} {
```

```
        set loss_module [new ErrorModel]
        $loss_module unit pkt
        $loss_module set rate_ $lrate
        $loss_module ranvar [new RandomVariable/Uniform]
        $loss_module drop-target [new Agent/Null]

        $ns lossmodel $loss_module $n1 $n2
        puts ">>>ErrorModel: lossy_link: $lossy_link - rate: $lrate"
}

#### TCP AGENT
set TCP [new Agent/TCP/$proto]
puts "Proto used: $proto"
$TCP set fid_ 1
$TCP set tau_ 1.0
$TCP set filter_type_ 3
$TCP set window_ 100
$TCP set maxcwnd_ 2000
$TCP set packetSize_ $psize
$ns attach-agent $n0 $TCP

set sink [new Agent/TCPSink/DelAck]
$ns attach-agent $n3 $sink

$ns connect $TCP $sink

set ftp [new Application/FTP]
$ftp attach-agent $TCP

### Record proc
proc record {} {
   global ns TCP sink record_f proto
   set now [$ns now]

   set time 0.15

   set cwin     [$TCP set cwnd_]
   set ssthresh [$TCP set ssthresh_]

   if { ($proto == "Westwood") || ($proto == "WestwoodNR")} {
        set bwe  [$TCP set current_bwe_]
   } else {
        set bwe  0
   }

   puts $record_f "$now [expr $cwin*1] $ssthresh $bwe"

   $ns at [expr $now+$time] "record "
}
$ns at 0.0 "record"
$ns at 0.0 "$ftp start"
```

```
$ns at 150.0 "finish"

$ns run
```

B.5 script of network topology 2: test-1-simple-10-flows-150s-Err-0.001-Fairness-x.tcl

```
# Fairness test
# TCP Holy Wood together with increasing number of TCP Reno flows
#
#   Oscar Nuñez Mori (PERU)
#   2005. All Right Reserved.
#
#  [1]  modified from test-1-simple.tcl of
#      TCP-Westwood. CS Department. UCLA. USA.  See (B.3)
#

### Default Values

set stop 150.0 ;# seconds
set flows 10   ;# only for label the trace .tr

set proto Reno
set buffer 0
set psize 1400
set lrate 0.0

### Read command line arguments
if {$argc > 1} {
   set proto  [lindex $argv 0]
   set buffer [lindex $argv 1]
   set lrate  [lindex $argv 2]
} else {

   puts "usage: ns test-1-simple.tcl <buffer> <error rate>"
   puts " "
   puts "<proto> is the protocol. ie Reno Westwood, HolyWood, etc."
   puts "<buffer> is the buffer size, use 0 for automatic pipe size setting"
   puts "<erro rate> is the link error rate (0.001 = 0.1%). Use 0 for no errors"

   exit 1
}

set ns [new Simulator]

# $ns color 1 Blue
# $ns color 2 Red
# $ns color 3 yellow
# $ns color 4 Brown
# $ns color 5 Black
# $ns color 6 Green
```

```
# $ns color 7 Purple
# $ns color 8 Grey
# $ns color 9
# $ns color 10
set tr_f       [open test-1-simple-$flows-flows-$proto-$lrate-150s-Friendliness.tr w]
# set nam_f     [open test-1-simple-$flows-flows-$proto-$lrate-150s-Friendliness.nam
w]

$ns trace-all $tr_f
# $ns namtrace-all $nam_f

### Finish proc
proc finish {} {
   global ns tr_f ;# nam_f
    $ns flush-trace
    close $tr_f
  #  close $nam_f

    exit 0
}

#
#
# Network topology 2:
#
#
#
#
#     (TCP1)                                                   (TCP-sink1)
#      n101                                                        n201
# n102  \                                                        /  n202
#        \ \ 15Mbit/s,1ms                                       /  /
#         \ \  ...                                             /  /
#          \ \                                                /  /
#   .   ..    \ \                                            /  / ...
#   ...        \  | 100Mbit/s     5Mbit/s       100Mbit/s   | /  ...
#               \ |     1ms          35ms              1ms       |/
#   ...           n0 ------- n1 ----------------- n2 ------------- n3    ...
#   ...          /                                               \   ...
#               /                                                 \
#      n110  /                                                     \ n210
#
# backlogged FTP sources with 1400 bytes packet size

set n0 [$ns node] ;# trace node 0
set n1 [$ns node] ;# trace node 1
set n2 [$ns node] ;# trace node 2
set n3 [$ns node] ;# trace node 3

set n101 [$ns node]  ;# trace node 4   TCP Reno #1, SOURCE
set n102 [$ns node]  ;# trace node 5   TCP Reno #2
set n103 [$ns node]  ;# trace node 6   TCP Reno #3
```

```
set n104 [$ns node]  ;# trace node 7   TCP Reno #4
set n105 [$ns node]  ;# trace node 8   TCP Reno #5
set n106 [$ns node]  ;# trace node 9   TCP Reno #6
set n107 [$ns node]  ;# trace node 10  TCP Reno #7
set n108 [$ns node]  ;# trace node 11  TCP Reno #8
set n109 [$ns node]  ;# trace node 12  TCP Reno #9
set n110 [$ns node]  ;# trace node 13  TCP Reno #10

set n201 [$ns node]  ;# trace node 14  TCP Reno #1, SINK
set n202 [$ns node]  ;# trace node 15  TCP Reno #2
set n203 [$ns node]  ;# trace node 16  TCP Reno #3
set n204 [$ns node]  ;# trace node 17  TCP Reno #4
set n205 [$ns node]  ;# trace node 18  TCP Reno #5
set n206 [$ns node]  ;# trace node 19  TCP Reno #6
set n207 [$ns node]  ;# trace node 20  TCP Reno #7
set n208 [$ns node]  ;# trace node 21  TCP Reno #8
set n209 [$ns node]  ;# trace node 22  TCP Reno #9
set n210 [$ns node]  ;# trace node 23  TCP Reno #10

#   object     from  to  bandwith   delay    queue
$ns duplex-link $n0  $n1   100Mb    1ms   DropTail
$ns duplex-link $n2  $n3   100Mb    1ms   DropTail
$ns duplex-link $n1  $n2      5Mb   35ms   DropTail ;# Bottleneck link

#   object     from  to  bandwith   delay    queue
$ns duplex-link $n101 $n0   15Mb     1ms   DropTail
$ns duplex-link $n102 $n0   15Mb     1ms   DropTail
$ns duplex-link $n103 $n0   15Mb     1ms   DropTail
$ns duplex-link $n104 $n0   15Mb     1ms   DropTail
$ns duplex-link $n105 $n0   15Mb     1ms   DropTail
$ns duplex-link $n106 $n0   15Mb     1ms   DropTail
$ns duplex-link $n107 $n0   15Mb     1ms   DropTail
$ns duplex-link $n108 $n0   15Mb     1ms   DropTail
$ns duplex-link $n109 $n0   15Mb     1ms   DropTail
$ns duplex-link $n110 $n0   15Mb     1ms   DropTail

#   object     from  to  bandwith   delay    queue
$ns duplex-link $n3 $n201   15Mb     1ms   DropTail
$ns duplex-link $n3 $n202   15Mb     1ms   DropTail
$ns duplex-link $n3 $n203   15Mb     1ms   DropTail
$ns duplex-link $n3 $n204   15Mb     1ms   DropTail
$ns duplex-link $n3 $n205   15Mb     1ms   DropTail
$ns duplex-link $n3 $n206   15Mb     1ms   DropTail
$ns duplex-link $n3 $n207   15Mb     1ms   DropTail
$ns duplex-link $n3 $n208   15Mb     1ms   DropTail
$ns duplex-link $n3 $n209   15Mb     1ms   DropTail
$ns duplex-link $n3 $n210   15Mb     1ms   DropTail

#### BOTTLENECK BUFFER
set rtd 72
set bneck_bw 5
# Set the queue size (queue size is in packets, default is 50)
```

```
# We should set the buffer capacity equal to the pipe size:
# buffer = bneck_bw * round_trip_delay / 8 / bytes_per_packet
set buffer_calc [expr $bneck_bw*1000000 * $rtd/1000 / 8 / $psize]
if {$buffer == 0} {  # no buffer provided
   set buffer $buffer_calc
}
puts "Buffer: set:$buffer - calc: $buffer_calc"

$ns queue-limit $n1 $n2 [expr $buffer *1]

$ns duplex-link-op $n1 $n2 queuePos 0.5

#### BOTTLENECK LINK ERRORS
# Add bottleneck link errors
set lossy_link 0
if {$lrate > 0} { set lossy_link 1 }

if { $lossy_link == 1} {
        set loss_module [new ErrorModel]
        $loss_module unit pkt
        $loss_module set rate_ $lrate
        $loss_module ranvar [new RandomVariable/Uniform]
        $loss_module drop-target [new Agent/Null]
        $ns lossmodel $loss_module $n1 $n2
        puts ">>>ErrorModel: lossy_link: $lossy_link - rate: $lrate"
}

#### TCP AGENTS

set TCP1 [new Agent/TCP/$proto]
puts "Proto used: $proto #1"
$TCP1 set fid_ 1
$TCP1 set tau_ 1.0
$TCP1 set filter_type_ 3
$TCP1 set window_ 100
$TCP1 set maxcwnd_ 2000
$TCP1 set packetSize_ $psize
$ns attach-agent $n101 $TCP1
set sink1 [new Agent/TCPSink/DelAck]
$ns attach-agent $n201 $sink1
$ns connect $TCP1 $sink1
set ftp1 [new Application/FTP]
$ftp1 attach-agent $TCP1

set TCP2 [new Agent/TCP/$proto]
puts "Proto used: $proto #2"
$TCP2 set fid_ 2
$TCP2 set tau_ 1.0
$TCP2 set filter_type_ 3
$TCP2 set window_ 100
$TCP2 set maxcwnd_ 2000
$TCP2 set packetSize_ $psize
```

```
$ns attach-agent $n102 $TCP2
set sink2 [new Agent/TCPSink/DelAck]
$ns attach-agent $n202 $sink2
$ns connect $TCP2 $sink2
set ftp2 [new Application/FTP]
$ftp2 attach-agent $TCP2

set TCP3 [new Agent/TCP/$proto]
puts "Proto used: $proto #3"
$TCP3 set fid_ 3
$TCP3 set tau_ 1.0
$TCP3 set filter_type_ 3
$TCP3 set window_ 100
$TCP3 set maxcwnd_ 2000
$TCP3 set packetSize_ $psize
$ns attach-agent $n103 $TCP3
set sink3 [new Agent/TCPSink/DelAck]
$ns attach-agent $n203 $sink3
$ns connect $TCP3 $sink3
set ftp3 [new Application/FTP]
$ftp3 attach-agent $TCP3

set TCP4 [new Agent/TCP/$proto]
puts "Proto used: $proto #4"
$TCP4 set fid_ 4
$TCP4 set tau_ 1.0
$TCP4 set filter_type_ 3
$TCP4 set window_ 100
$TCP4 set maxcwnd_ 2000
$TCP4 set packetSize_ $psize
$ns attach-agent $n104 $TCP4
set sink4 [new Agent/TCPSink/DelAck]
$ns attach-agent $n204 $sink4
$ns connect $TCP4 $sink4
set ftp4 [new Application/FTP]
$ftp4 attach-agent $TCP4

set TCP5 [new Agent/TCP/$proto]
puts "Proto used: $proto #5"
$TCP5 set fid_ 5
$TCP5 set tau_ 1.0
$TCP5 set filter_type_ 3
$TCP5 set window_ 100
$TCP5 set maxcwnd_ 2000
$TCP5 set packetSize_ $psize
$ns attach-agent $n105 $TCP5
set sink5 [new Agent/TCPSink/DelAck]
$ns attach-agent $n205 $sink5
$ns connect $TCP5 $sink5
set ftp5 [new Application/FTP]
$ftp5 attach-agent $TCP5
```

```
set TCP6 [new Agent/TCP/$proto]
puts "Proto used: $proto #6"
$TCP6 set fid_ 6
$TCP6 set tau_ 1.0
$TCP6 set filter_type_ 3
$TCP6 set window_ 100
$TCP6 set maxcwnd_ 2000
$TCP6 set packetSize_ $psize
$ns attach-agent $n106 $TCP6
set sink6 [new Agent/TCPSink/DelAck]
$ns attach-agent $n206 $sink6
$ns connect $TCP6 $sink6
set ftp6 [new Application/FTP]
$ftp6 attach-agent $TCP6

set TCP7 [new Agent/TCP/$proto]
puts "Proto used: $proto #7"
$TCP7 set fid_ 7
$TCP7 set tau_ 1.0
$TCP7 set filter_type_ 3
$TCP7 set window_ 100
$TCP7 set maxcwnd_ 2000
$TCP7 set packetSize_ $psize
$ns attach-agent $n107 $TCP7
set sink7 [new Agent/TCPSink/DelAck]
$ns attach-agent $n207 $sink7
$ns connect $TCP7 $sink7
set ftp7 [new Application/FTP]
$ftp7 attach-agent $TCP7

set TCP8 [new Agent/TCP/$proto]
puts "Proto used: $proto #8"
$TCP8 set fid_ 8
$TCP8 set tau_ 1.0
$TCP8 set filter_type_ 3
$TCP8 set window_ 100
$TCP8 set maxcwnd_ 2000
$TCP8 set packetSize_ $psize
$ns attach-agent $n108 $TCP8
set sink8 [new Agent/TCPSink/DelAck]
$ns attach-agent $n208 $sink8
$ns connect $TCP8 $sink8
set ftp8 [new Application/FTP]
$ftp8 attach-agent $TCP8

set TCP9 [new Agent/TCP/$proto]
puts "Proto used: $proto #9"
$TCP9 set fid_ 9
$TCP9 set tau_ 1.0
$TCP9 set filter_type_ 3
$TCP9 set window_ 100
$TCP9 set maxcwnd_ 2000
```

```
$TCP9 set packetSize_ $psize
$ns attach-agent $n109 $TCP9
set sink9 [new Agent/TCPSink/DelAck]
$ns attach-agent $n209 $sink9
$ns connect $TCP9 $sink9
set ftp9 [new Application/FTP]
$ftp9 attach-agent $TCP9

set TCP10 [new Agent/TCP/$proto]
puts "Proto used: $proto #10"
$TCP10 set fid_ 10
$TCP10 set tau_ 1.0
$TCP10 set filter_type_ 3
$TCP10 set window_ 100
$TCP10 set maxcwnd_ 2000
$TCP10 set packetSize_ $psize
$ns attach-agent $n110 $TCP10
set sink10 [new Agent/TCPSink/DelAck]
$ns attach-agent $n210 $sink10
$ns connect $TCP10 $sink10
set ftp10 [new Application/FTP]
$ftp10 attach-agent $TCP10


$ns at 0.0 "$ftp1 start"
$ns at 0.0 "$ftp2 start"
$ns at 0.0 "$ftp3 start"
$ns at 0.0 "$ftp4 start"
$ns at 0.0 "$ftp5 start"
$ns at 0.0 "$ftp6 start"
$ns at 0.0 "$ftp7 start"
$ns at 0.0 "$ftp8 start"
$ns at 0.0 "$ftp9 start"
$ns at 0.0 "$ftp10 start"

$ns at $stop "finish"

$ns run
```

B.6 AWK Script 1, to calculate Average, Standard Deviation, COV and Confidence Intervals: avg-s-cov-TESTED-END.awk

```
#  Author: Oscar Nunez Mori (Republic of Peru)
#  Advisor: Dr. Juergen Rochol
#  UFRGS-RS_BRAZIL
#
#  little Statistical Program to calculate Average, Standard Deviation
# Coefficient of Variation (COV) and Confidence Intervals
#
#
#  Based on (ALTMAN; JIMENEZ, 2003) page 43
#
```

```
#  USAGE:
#  awk -v SDcol=? -v from=? -v to=? -f avg-s-cov-TESTED-END.awk  ?
#
BEGIN {
minColumn = from; # Constants  here
maxColumn = to;
}
{

column = $SDcol; # Standard Deviation Column Variables here

if(NR >= minColumn && NR <= maxColumn ) {

        matrix[tmp]= $SDcol;
        tmp++;
           n++;
        sumData += $SDcol
         }
}

END {
 if(n-1 ==  1) tStudent = 12.706;
 if(n-1 ==  2) tStudent =  4.303;
 if(n-1 ==  3) tStudent =  3.182;
 if(n-1 ==  4) tStudent =  2.776;
 if(n-1 ==  5) tStudent =  2.571;
 if(n-1 ==  6) tStudent =  2.447;
 if(n-1 ==  7) tStudent =  2.365;
 if(n-1 ==  8) tStudent =  2.306;
 if(n-1 ==  9) tStudent =  2.262;
 if(n-1 == 10) tStudent =  2.228;
 if(n-1 == 11) tStudent =  2.201;
 if(n-1 == 12) tStudent =  2.179;
 if(n-1 == 13) tStudent =  2.160;
 if(n-1 == 14) tStudent =  2.145;
 if(n-1 == 15) tStudent =  2.131;
 if(n-1 == 16) tStudent =  2.120;
 if(n-1 == 17) tStudent =  2.110;
 if(n-1 == 18) tStudent =  2.101;
 if(n-1 == 19) tStudent =  2.093;
 if(n-1 == 20) tStudent =  2.086;
 if(n-1 == 21) tStudent =  2.080;
 if(n-1 == 22) tStudent =  2.074;
 if(n-1 == 23) tStudent =  2.069;
 if(n-1 == 24) tStudent =  2.064;
 if(n-1 == 25) tStudent =  2.060;
 if(n-1 == 26) tStudent =  2.056;
 if(n-1 == 27) tStudent =  2.052;
 if(n-1 == 28) tStudent =  2.048;
 if(n-1 == 29) tStudent =  2.045;
 if(n-1 == 30) tStudent =  2.042;
 if(n-1 >  30) tStudent =  1.960;  # for n>30
```

```
avg    =  sumData/n;

tmp=0; # clean up tmp
for(tmp in matrix) {
 diff  = matrix[tmp] - avg;
 var  += diff * diff;
 }

stdDev = sqrt(var/(n-1));
cov    = (stdDev/avg)*100; # COV PERCENT

Confidence_const = tStudent * (stdDev / sqrt(n));
  C1 = avg - Confidence_const;
  C2 = avg + Confidence_const;

printf("\nSTATISTICS\n\n ");
printf("   Number of Samples: n =  %d\n ", n);
printf("   t Student's Distribution Value: n=%d  --->  %f\n\n ",n-1,tStudent);
printf("   Average =  %f\n ", avg);
printf("   Standard Deviation_(n-1)  =  %f\n ", stdDev);
printf("   Coefficient of Variation  =  %f\%%\n ", cov);
printf("   95\%% Confidence Interval: C1  =  %f\n ", C1);
printf("   95\%% Confidence Interval: C2  =  %f\n ", C2);
printf("   Average (+/-)Confidence Constant = %f (+/-)%f\n\n ",avg,Confidence_const);

 } # the End

# Usage
# awk -v SDcol=? -v from=? -v to=? -f avg-s-cov-TESTED-END.awk  ?
#
#  > awk -v SDcol=2 -v from=3 -v to=4 -f avg-s-cov.awk test4.dat
#  > awk -v SDcol=1 -v from=1 -v to=NR -f avg.awk xxxxx.dat
#
# Note the first line we mean from=? In the file could be 0 or 1, AWK,  according to
# our observation consider the first position as 1;
```

B.7 AWK Script 2, To calculate COV and Confidence Intervals with known Average and standard deviation

```
#  Awk program to Calculate Coeficient of Variance (COV) and Confidence Intervals
#
# awk -v TCP=? -v factor=? -v from=? -v to=NR -v avgC=? -v sdC=? -f cov-conf-
intervals-TESTED-Table.awk  ?
#
# awk -v TCP=HOLYWOOD -v factor=ErrorRate -v from=9 -v to=NR -v avgC=2 -v
sdC=5 -f cov-conf-intervals-TESTED-Table.awk  xxxxx.dat

#--------------------------------------------------------
BEGIN {
# Initializing Constants
TCPName = TCP;
```

```
factorName= factor;
minColumn= from;
maxColumn= to
} # Finished BEGIN
#---------------------------------------------------------
#---------------------------------------------------------
{ # Starts Search Pattern 1

n=150; # 150s simulation time each line
tStudent=1.960 # See Table below n --> infinity

if(NR>=from && NR <= to) {

firstCol  = $1
avgCol    = $avgC;
sdCol     = $sdC;

confConst = tStudent * ( sdCol / sqrt(n) );
COV       =  (sdCol /  avgCol)*100; # COV PERCENT

printf("%.4f     \t%.2f \t%.2f \t%.2f \t%.2f \t%.2f \n",firstCol ,avgCol
,sdCol,COV,avgCol - confConst, avgCol + confConst );

        }#Finished if

} # Finished Search Pattern 1
#---------------------------------------------------------
#---------------------------------------------------------
END {
printf("#");
printf("\n#%s[\%%]\tAverage  \tStand.Dev. \tCOV[\%%] \tC1 \t\tC2",factorName);
printf("\n#");
printf("________________________________________________________________________
______________________________________________________");
printf("\n# TCP %s;  Factor: %s;  C1 < C2;  n=150;  t Student's distribuions used: n >
30 --> t = 1.960",TCPName, factorName);
printf("\n#              _");
printf("\n# Author: Oscar Nunez Mori (Republic of Peru)\n");
printf("# Advisor: Dr. Juergen Rochol\n")
printf("# UFRGS-RS- Brazil\n#\n")
}# Finished END section
```

B.8 AWK Script 3, to calculate the lost packet Percent: lost-packet-stat-6O_PERCENT_TESTED_END.awk

```
#   Little Statistic Program  in AWK
#   Author: Oscar Nunez Mori (PERU)
#   Advisor: Dr. Juerge Rochol
#   UFRGS-RS Brazil
#
#  Usage:
#
```

```
#  awk -f lost-packet-stat-TESTED.awk xxx.dat
#
#  xxx.dat = info-simulation-Err-x-150s-HolyWood-PERCENT.dat

BEGIN {
# Initializations
     }

#================SEARCH PATTERN 1==================#
{

sentPackets     = $3;
lostPackets     = $6;
 minColumnOfFile = 5;
 MaxColumnOfFile = 20;

 if(NR  >= minColumnOfFile &&  NR <= MaxColumnOfFile) {

            n++;  # Total number of elements

            matrixSentPkts[n] = sentPackets;
         sentPacketsForAllErrorRates += sentPackets;

            matrixLostPkts[n] = lostPackets;
            lostPacketsForAllErrorRates += lostPackets;
      }
}

#================SEARCH PATTERN 2=================#
{
# Empty
}
#================SEARCH PATTERN 3===============#
{
# Empty
}


#=================FINAL ACTIONS=================#
 END {
#{
 if(n-1 ==  1) tStudent = 12.706;
 if(n-1 ==  2) tStudent =  4.303;
 if(n-1 ==  3) tStudent =  3.182;
 if(n-1 ==  4) tStudent =  2.776;
 if(n-1 ==  5) tStudent =  2.571;
 if(n-1 ==  6) tStudent =  2.447;
 if(n-1 ==  7) tStudent =  2.365;
 if(n-1 ==  8) tStudent =  2.306;
 if(n-1 ==  9) tStudent =  2.262;
 if(n-1 == 10) tStudent =  2.228;
 if(n-1 == 11) tStudent =  2.201;
```

```
 if(n-1 == 12) tStudent =  2.179;
 if(n-1 == 13) tStudent =  2.160;
 if(n-1 == 14) tStudent =  2.145;
 if(n-1 == 15) tStudent =  2.131;
 if(n-1 == 16) tStudent =  2.120;
 if(n-1 == 17) tStudent =  2.110;
 if(n-1 == 18) tStudent =  2.101;
 if(n-1 == 19) tStudent =  2.093;
 if(n-1 == 20) tStudent =  2.086;
 if(n-1 == 21) tStudent =  2.080;
 if(n-1 == 22) tStudent =  2.074;
 if(n-1 == 23) tStudent =  2.069;
 if(n-1 == 24) tStudent =  2.064;
 if(n-1 == 25) tStudent =  2.060;
 if(n-1 == 26) tStudent =  2.056;
 if(n-1 == 27) tStudent =  2.052;
 if(n-1 == 28) tStudent =  2.048;
 if(n-1 == 29) tStudent =  2.045;
 if(n-1 == 30) tStudent =  2.042;
 if(n-1 >  30) tStudent =  1.960;  # for n>30
#}
# END {

#+++++ Average or Arithmetic Mean of SENT Packets for error rate
#                 from 0% to  60%  ++++++++++++++++++++++++++++++

averageSentPacketsForAllErrorRates = sentPacketsForAllErrorRates / n;


#--------- Standard Deviation of SENT Packets---------------------

for(i  in  matrixSentPkts) {

  differenceSent    =  matrixSentPkts[i] - averageSentPacketsForAllErrorRates;
  varianceSent     +=  differenceSent * differenceSent ;
                           }

  standardDeviationSent = sqrt(varianceSent/(n-1));



#--------- Coefficient of Variation of SENT Packets  COV --------

COV_SentPackets =   standardDeviationSent / averageSentPacketsForAllErrorRates;

#---- Confidence intervals for SENT Packets  C1 < C2 u with 95% -----

 Confidence_Constant_Sent =  tStudent * ( standardDeviationSent / sqrt(n) );

 C1_SentPkts = averageSentPacketsForAllErrorRates + Confidence_Constant_Sent;

 C2_SentPkts = averageSentPacketsForAllErrorRates - Confidence_Constant_Sent;
```

```
#++++++++++ Average or Arithmetic Mean of LOST Packets for error rate
#                    from 0% to 60% ++++++++++++++++++#

 averageLostPacketsForAllErrorRates = lostPacketsForAllErrorRates / n;



#--------- Standard Deviation of LOST Packets---------------------

for(i  in  matrixLostPkts) {

   differenceLost    =  matrixLostPkts[i] -  averageLostPacketsForAllErrorRates;
   varianceLost     +=  differenceLost * differenceLost;
                    }

   standardDeviationLost = sqrt(varianceLost / (n-1));

#--------- Coefficient of Variation of LOST Packets  COV ----------

 COV_LostPackets =   standardDeviationLost / averageLostPacketsForAllErrorRates;

#----- Confidence intervals for LOST Packets  C1 < C2 u with 95%-----

Confidence_Constant_Lost =  tStudent * ( standardDeviationLost / sqrt(n) );

C1_LostPkts = averageLostPacketsForAllErrorRates + Confidence_Constant_Lost;

C2_LostPkts = averageLostPacketsForAllErrorRates - Confidence_Constant_Lost;



#++++++++++ Percent Ratio of  Send Packets to  Lost Packets +++++++++

percentageLostPacketsToSentPackets = ( averageLostPacketsForAllErrorRates /
averageSentPacketsForAllErrorRates ) * 100;


printf("\n");
printf("      STATISTICS\n");

printf("     Number of Samples:  %f\n", n );
printf("     t Student's distribution value used for Confidence Intervals:  n = %d --->
%.3f\n", n-1, tStudent );

printf("SENT Packet Statistics \n");

printf("     Average(or Mean) of Total Sent Packets:  %f\n",
averageSentPacketsForAllErrorRates );
printf("     Standard Deviation  of Total Sent Packets:  %f\n",
```

```
standardDeviationSent            );
printf("    Coefficient of variation of Total Sent Packets:   COV =  %f\n",
COV_SentPackets                  );
printf("    Minimum Bound Value of Total Average Sent Packets with 95\%%
confidence interval: C1 =  %f\n",   C1_SentPkts        );
printf("    Maximum Bound Value of Total Average Sent Packets with 95\%%
confidence interval: C2 =  %f\n",   C2_SentPkts        );
printf("    Average (+/-)Confidence Constant :  %f (+/-)%f\n",
averageSentPacketsForAllErrorRates, Confidence_Constant_Sent   );
printf("\n");
printf("LOST Packet Statistics \n");
printf("    Average(mean) of Total Lost Packets:  %f\n",
averageLostPacketsForAllErrorRates );
printf("    Standard Deviation  of Total Lost Packets:  %f\n",
standardDeviationLost            );
printf("    Coefficient of variation of Total Lost Packets:   COV =  %f\n",
COV_LostPackets                  );
printf("    Minimum Bound Value of Total Average Lost Packets with 95\%%
confidence interval: C1 =  %f\n",   C1_LostPkts        );
printf("    Maximum Bound Value of Total Average Lost Packets with 95\%%
confidence interval: C2 =  %f\n",   C2_LostPkts        );
printf("    Average (+/-)Confidence Constant :  %f (+/-)%f\n",
averageLostPacketsForAllErrorRates, Confidence_Constant_Lost   );
printf("\n");
printf("TOTAL Ratio of AverageLostPackets to AverageSentPackets:  %f\%%\n",
percentageLostPacketsToSentPackets);
printf("\n");
}
```

B.9 AWK script 4, to calculate the Average of an interval

```
# little statistical program to Calculate the Average or Arithmetic mean
# by intervals
#
# FORMAT
#
# awk -v col=? -v min=? -v max=? -f  avg-TESTED-OK.awk  ?
#
#
BEGIN {
minColumnValue = min;  # initial column Value
maxColumnValue = max;  # final  Column Value
printf("\n\n------ Average of an Interval ------\n\n  ");
}
{
column = $col; # Variables here
if(NR >= minColumnValue && NR <=maxColumnValue) {
        count++;
        avg += column;
printf("Values %d = %f\n  ",count,column);
         }
}
```

```
END {
printf("------------------------------\n  ");
printf(" Addition =  %f\n  ", avg);
printf(" Average  =  %f\n  ", avg/count);
printf("       n  =  %d\n  ", count);

    }
# Usage
# awk -v col=1 -v min=6 -v max=15 -f avg.awk xxxxx.dat
# awk -v col=1 -v min=1 -v max=NR -f avg.awk xxxxx.dat ;
# without intervals
```

B.10 Gnuplot script 1: for one Figure

```
#!/bin/sh
gnuplot << EOF

# Gnuplot script file for plotting data from a file
# made by Oscar Nunez Mori (PERU)
# Reference: The Original idea is from KAWANO (KAWANO, 2004)
#
set title "THROUGHPUT GAIN" font "Times-Roman,25"
set xlabel "(a)                          Error Rate = 0.1%                          Time(s)" font
"Times-Roman,20"
set ylabel "Throughput (bit/s)" font "Times-Roman,20"
set xrange [0:150]
set format y " %.1s%c"
set xtics 10

plot "../../../$1/FILENAME1.trg" \
        u 1:2 t 'TCP-vegas' w linespoints 3, \
    "../../../$1/FILENAME2.trg" \
        u 1:2 t 'TCP-westwood' w linespoints 2, \
    "../../../$1/FILENAME3.trg" \
        u 1:2 t 'TCP-holywood' w linespoints 1
# Outputing to a Postscript File

 set term epslatex color solid
 set output "Throughput-Gain--$1-END.tex"
replot
 set term postscript enhanced
 set output "Throughput-Gain--$1-END.ps"
replot
 set term gif
 set output "Throughput-Gain--$1-END.gif
replot
 set term png medium
 set output "Throughput-Gain--$1-END.png"
replot
 set term post eps enhanced color solid
 set output "Throughput-Gain-TIL_1-CN_3--$1-END.eps"
```

```
replot

EOF

# usage:
# > ./PROGRAM1.gpt 0.001
```

B.11 Gnuplot script 2: for four figures

```
 #!/bin/sh
gnuplot << EOF

# Gnuplot script file for plotting data from a file
# made by Oscar Nunez Mori (PERU)
# Reference: The Original Idea is from Kawano (KAWANO, 2004)
# Outputting to a Postscript File
# Note: put out the commentaries one by one but not more than one

 set term gif
 set output "Throughput-Gain--four-Figures-END.gif"

# set term postscript enhanced
# set output "Throughput-Gain--four-Figures-END.ps"
# set term post eps enhanced color solid
# set output "Throughput-Gain—four-Figures-END.eps"
# set term latex
# set output "Throughput-Gain--four-Figures-END.tex"

set title "THROUGHPUT GAIN" font "Times-Roman,16"
#set xlabel "Time[s] \n a" font "Times-Roman,15"
set ylabel "Throughput [bit/s]" font "Times-Roman,15"
set xrange [0:150]
set format y " %.1s%c"
set xtics 20
set samples 1000
 set lmargin 8
 set rmargin 8
set multiplot

#------------ Figure 1 ---------------#
set origin 0.05, 0.50
set size   0.5, 0.5
# set label 1 "Error=0.0%" at 1,100

set xlabel "(a)         Error=0.00%         Time[s]" font "Times-Roman,15"
set ylabel "Throughput [bit/s]" font "Times-Roman,15"
set nokey

plot "../../../$1/FILENAME-$1-1.trg"\
        u 1:2 t 'TCP-vegas' w linespoints 3,\
     "../../../$1/FILENAME-$1-2.trg"\
        u 1:2 t 'TCP-westwood' w linespoints 2,\
```

```
    "../../../$1/FILENAME-$1-3.trg" \
        u 1:2 t 'TCP-holywood' w linespoints 1

#------------- Figure 2 -------------#
set origin 0.5, 0.50
set size  0.5, 0.5

set xlabel "(b)        Error=0.01%        Time[s]" font "Times-Roman,15"
set ylabel ""
set nokey

plot "../../../$2/FILENAME-$2-1.trg"\
        u 1:2 t 'TCP-vegas' w linespoints 3,\
    "../../../$2/FILENAME-$2-2.trg"\
        u 1:2 t 'TCP-westwood' w linespoints 2,\
    "../../../$2/FILENAME-$2-3.trg"\
        u 1:2 t 'TCP-holywood' w linespoints 1

#------------- Figure 3 ------------#

set origin 0.05, 0.00
set size   0.5, 0.5

# set title ""
set xlabel "(c)        Error=1.00%        Time[s]" font "Times-Roman,15"
set ylabel "Throughput [bit/s]" font "Times-Roman,15"
set key right top

plot "../../../$3/FILENAME-$3-1.trg"\
        u 1:2 t 'TCP-vegas' w linespoints 3,\
    "../../../$3/FILENAME-$3-2.trg"\
        u 1:2 t 'TCP-westwood' w linespoints 2,\
    "../../../$3/FILENAME-$3-3.trg"\
        u 1:2 t 'TCP-holywood' w linespoints 1
#------------ Figure 4 -------------#
set origin 0.5, 0.00
set size  0.5, 0.5
# set title ""
set xlabel "(d)        Error=10.0%        Time[s]" font "Times-Roman,15"
set ylabel ""
set key right top

plot "../../../$4/FILENAME1-$4.trg"\
        u 1:2 t 'TCP-vegas' w linespoints 3,\
    "../../../$4/FILENAME2-$4.trg"\
        u 1:2 t 'TCP-westwood' w linespoints 2, \
    "../../../$4/FILENAME3-$4.trg"\
        u 1:2 t 'TCP-holywood' w linespoints 1
# ----------- END ------------------#
set nomultiplot
 pause -1
EOF
```

```
# usage:
#                $1   $2    $3  $4
# > ./PROGRAM.gpt   0   0.0001    0.01   0.1
```

APPENDIX C COMPLETE SET OF RESULTS FIGURES

C.1 TCP HolyWood versus TCP Reno (Standard)

We used the topology 1 in all the tests.

C.1.1 Impact of Error Rate on Throughput

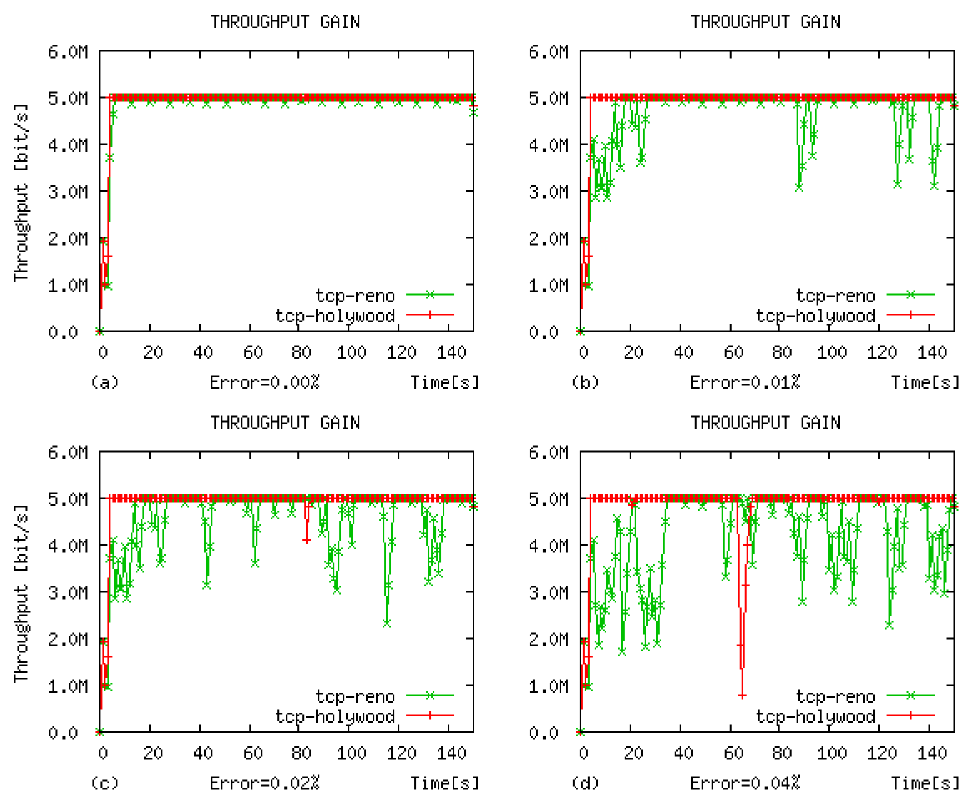

Figure C.1.1.1: Impact of Error Rate on Throughput (1 of 3)

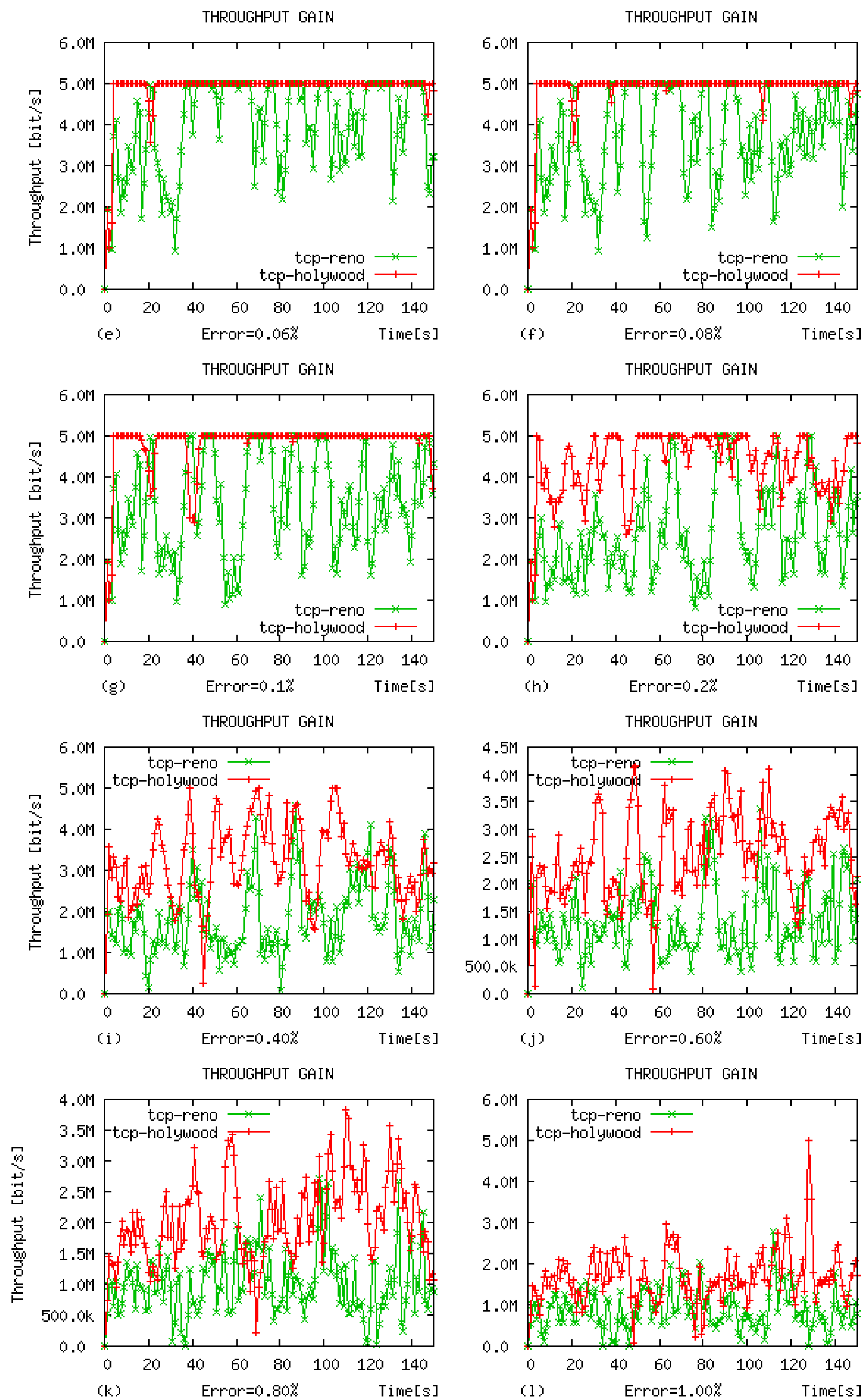

Figure C.1.1.2: Impact of Error Rate on Throughput (2 of 3)

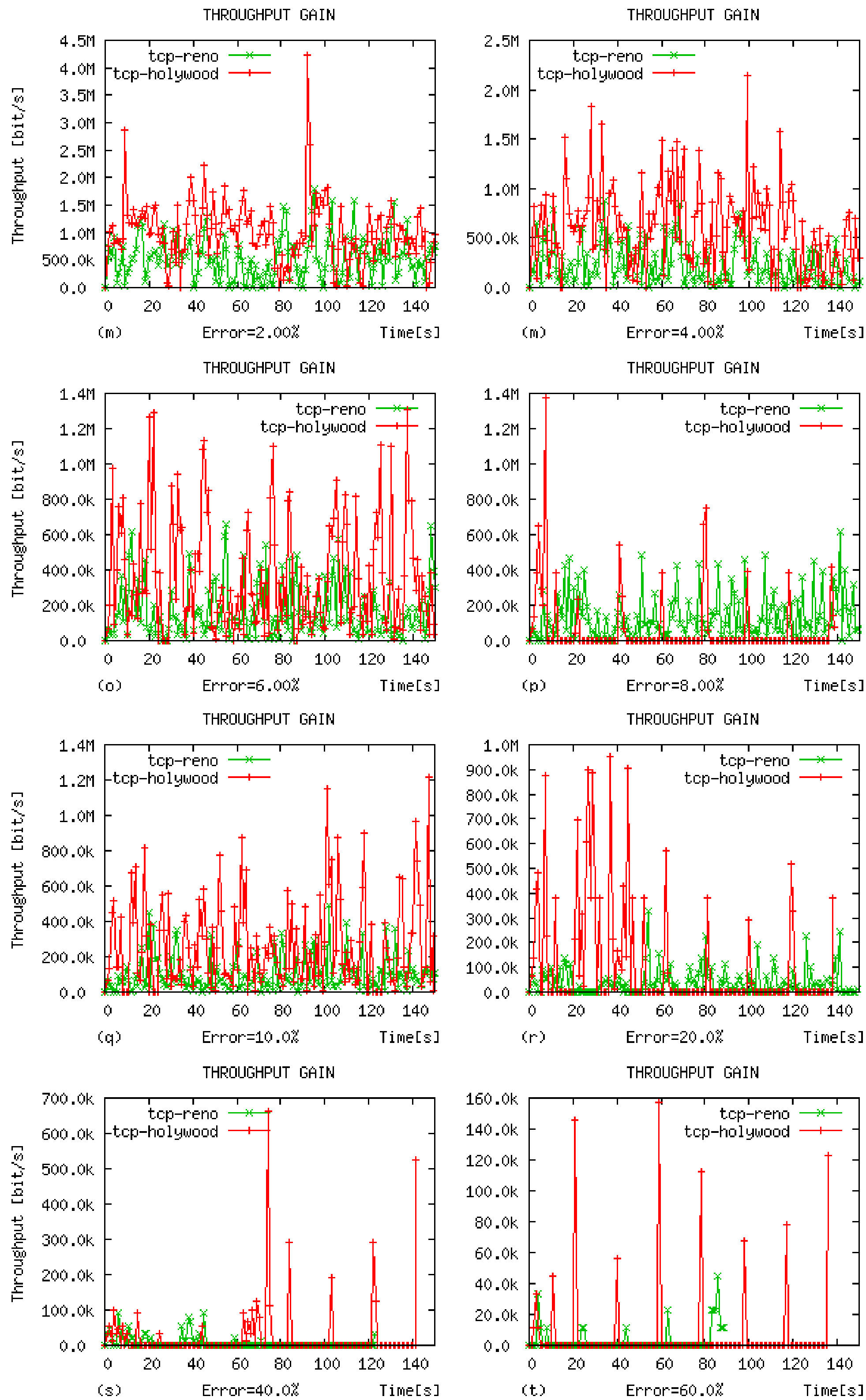

Figure C.1.1.3: Impact of Error Rate on Throughput (3 of 3)

C.1.2 Impact of Propagation Time Throughput

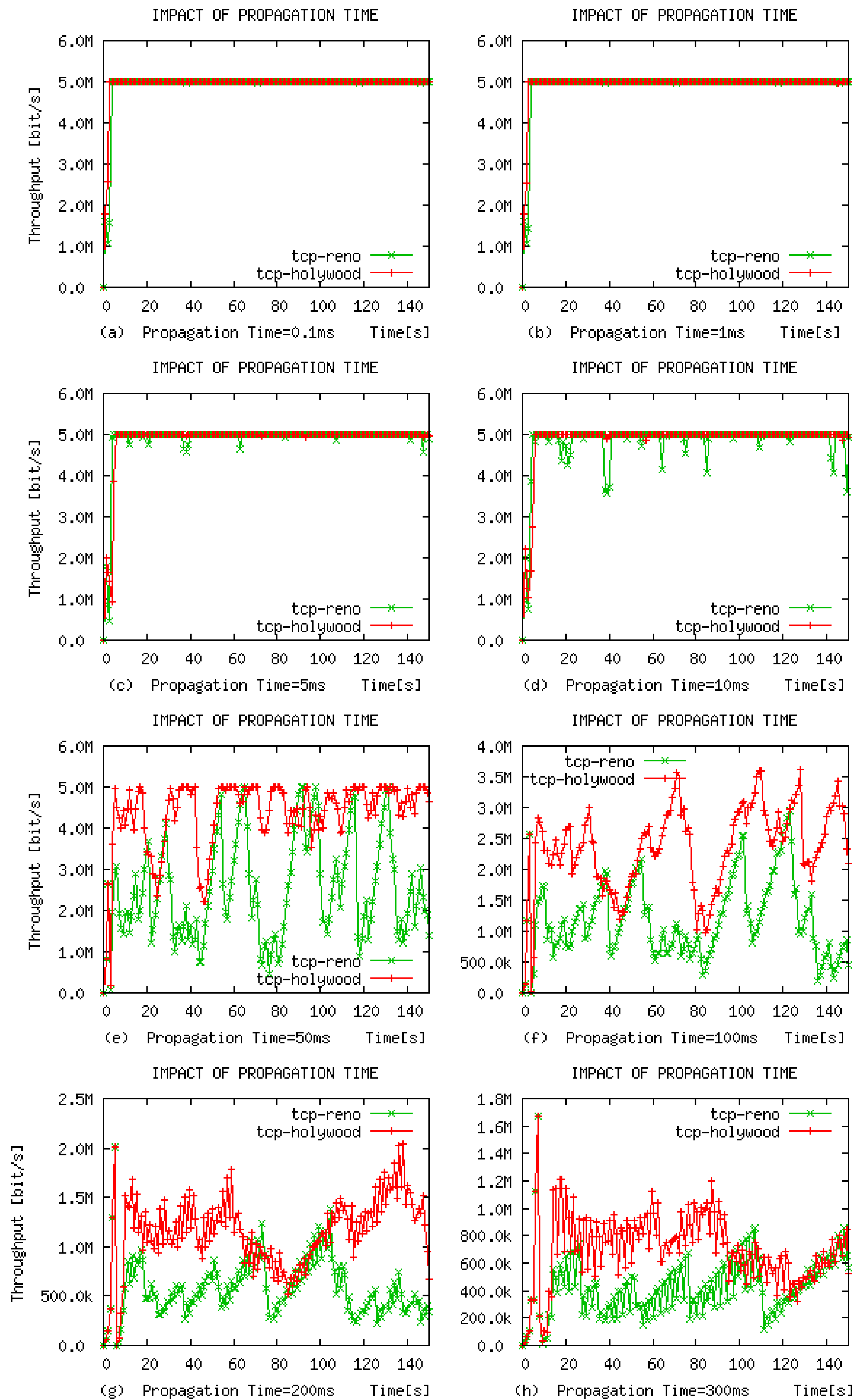

Figure C.1.2.1: Impact of Propagation Time Throughput (1 of 2)

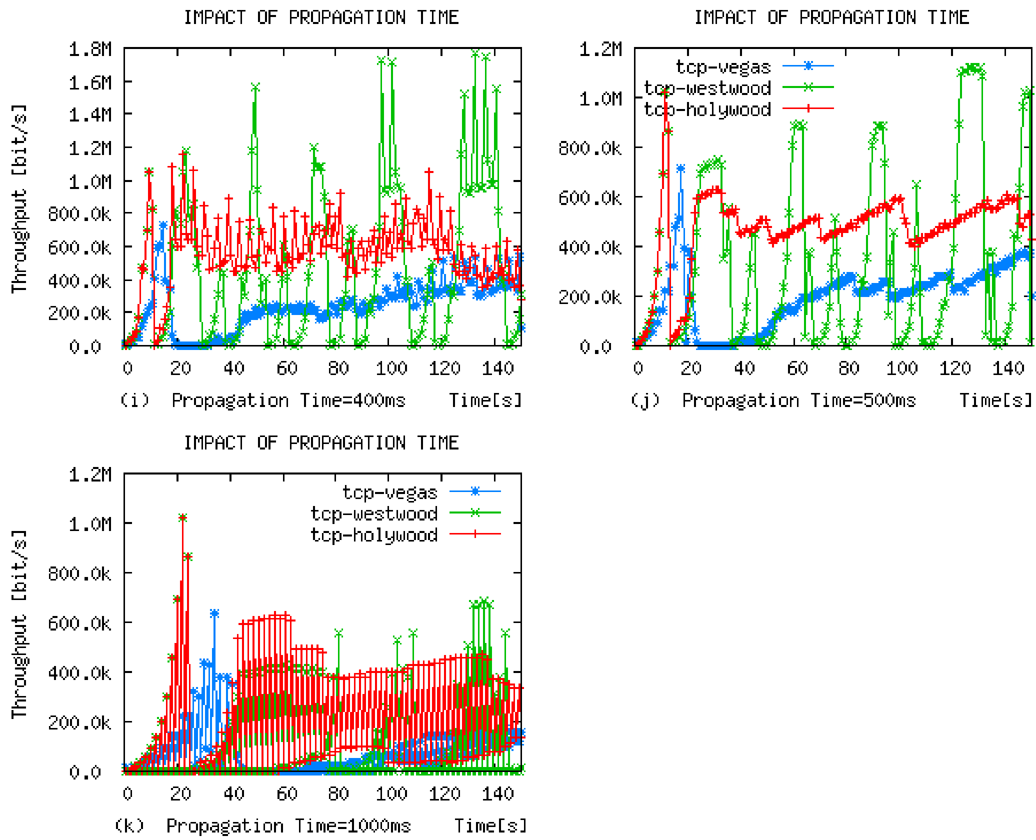

Figure C.1.2.2: Impact of Propagation Time Throughput (2 of 2)

C.1.3 Impact of Bottleneck Bandwidth on Throughput

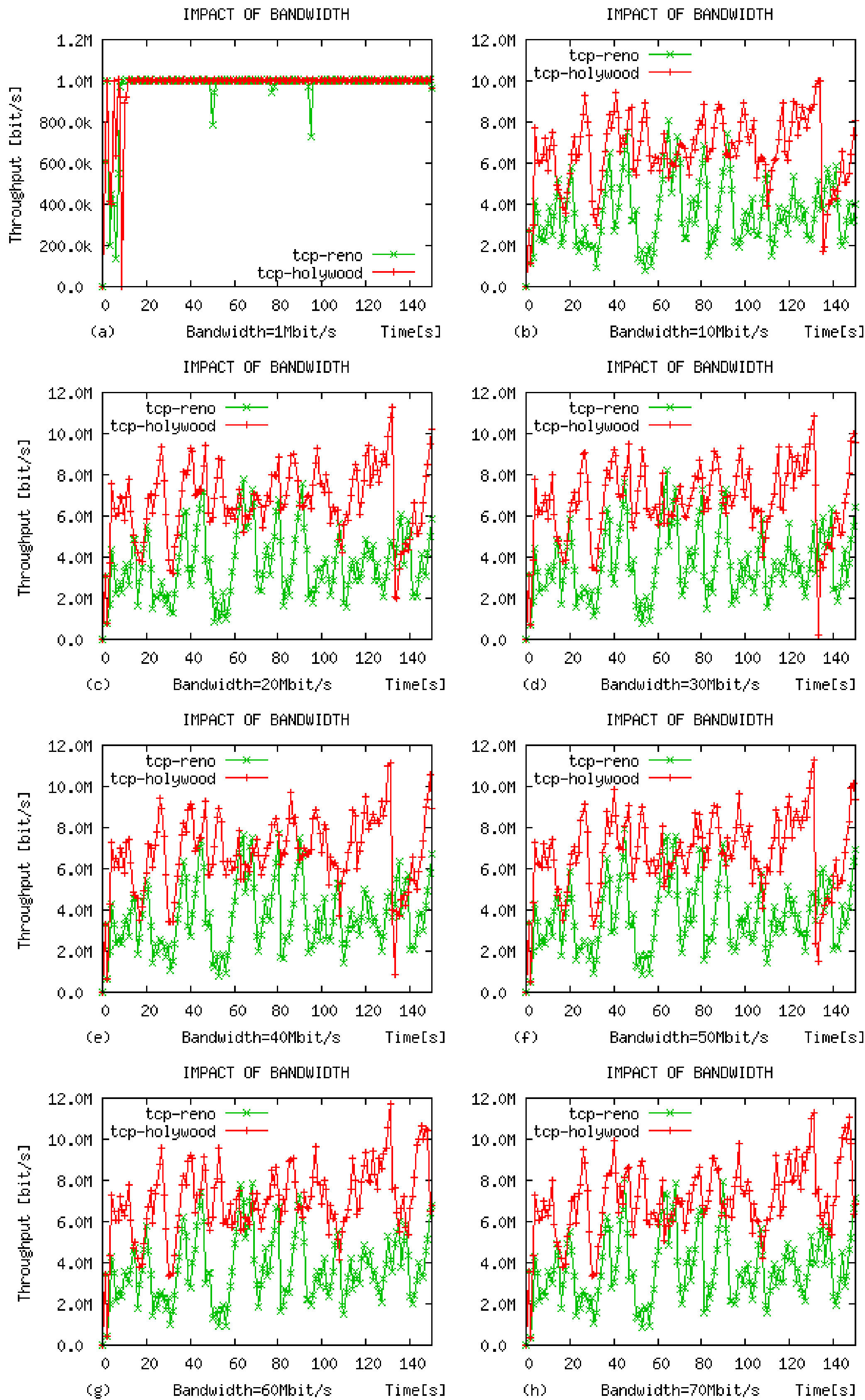

Figure C.1.3.1: Impact of Bottleneck Bandwidth on Throughput (1 of 2)

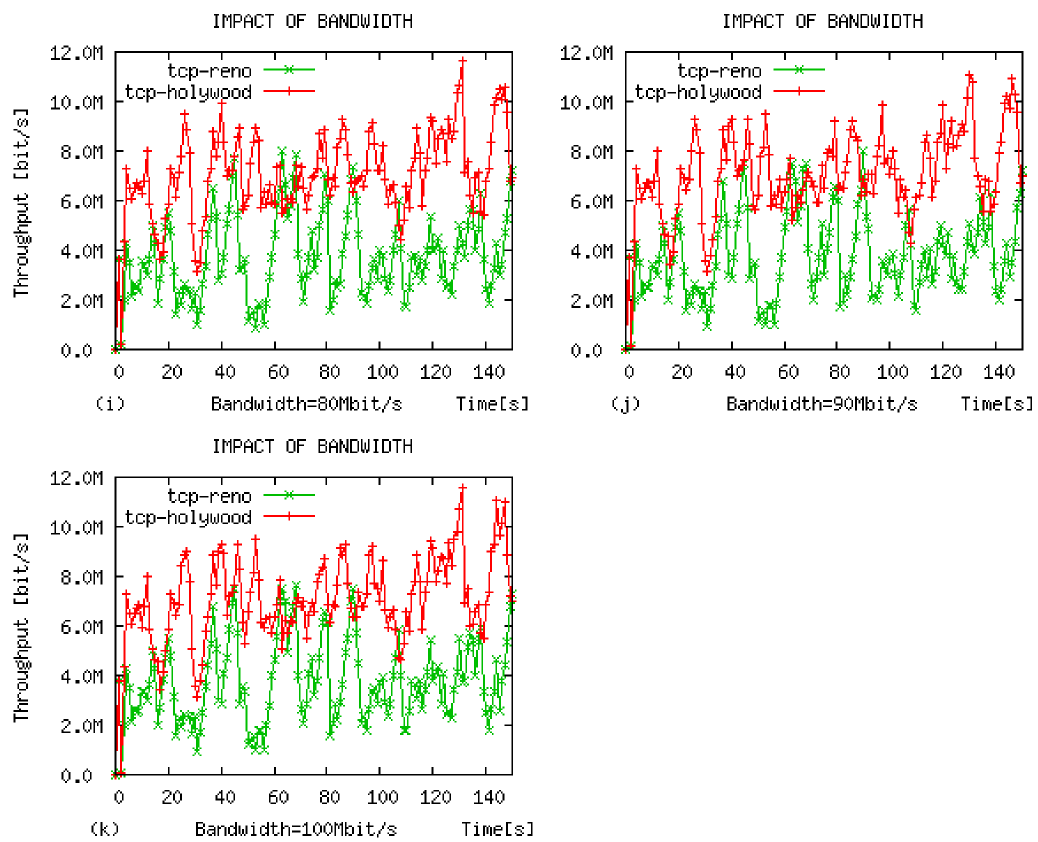

Figure C.1.3.2: Impact of Bottleneck Bandwidth on Throughput (2 of 2)

C.1.4 Impact of Error rate on Jitter

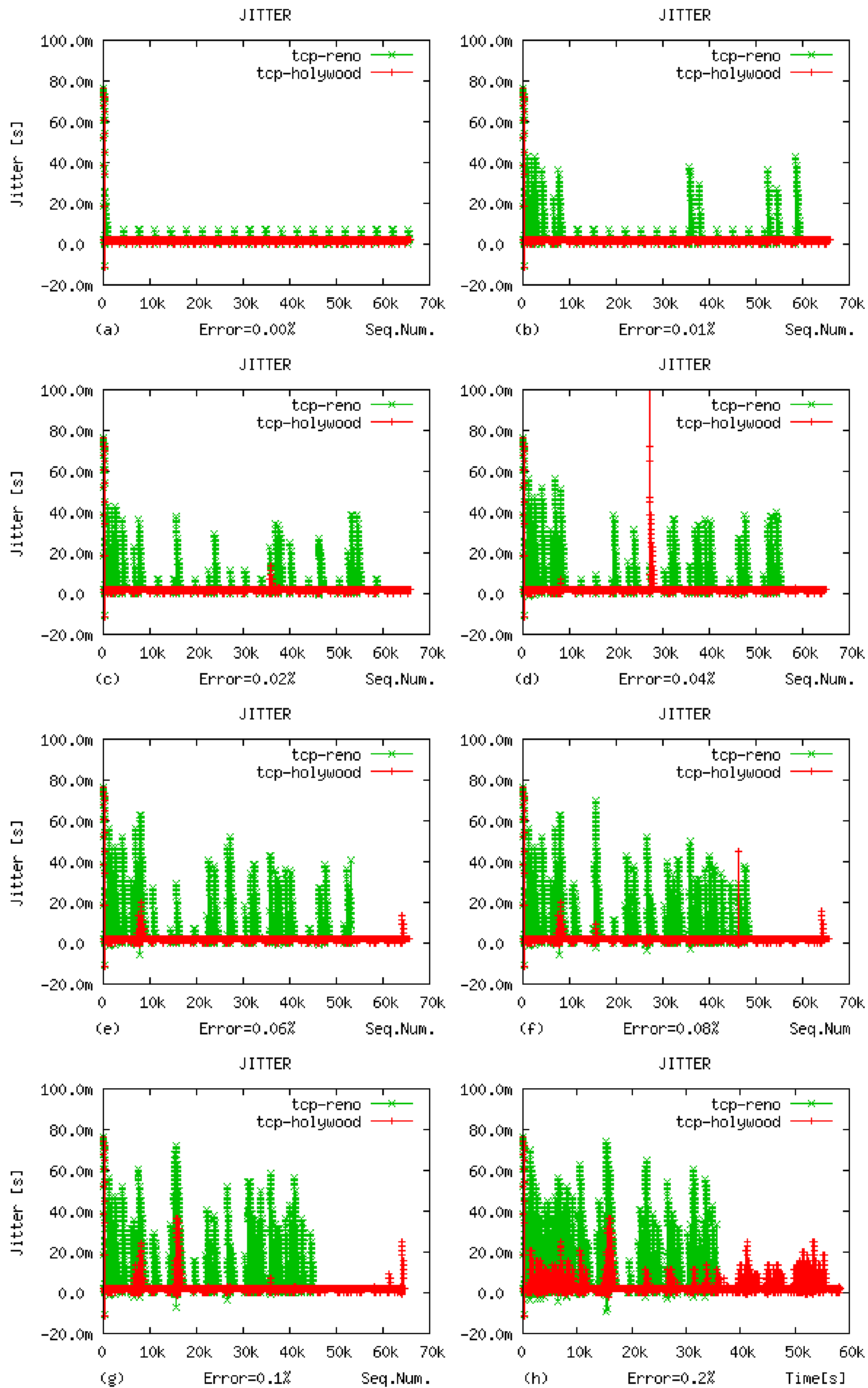

Figure C.1.4.1: Impact of Error Rate on Jitter (1 of 3)

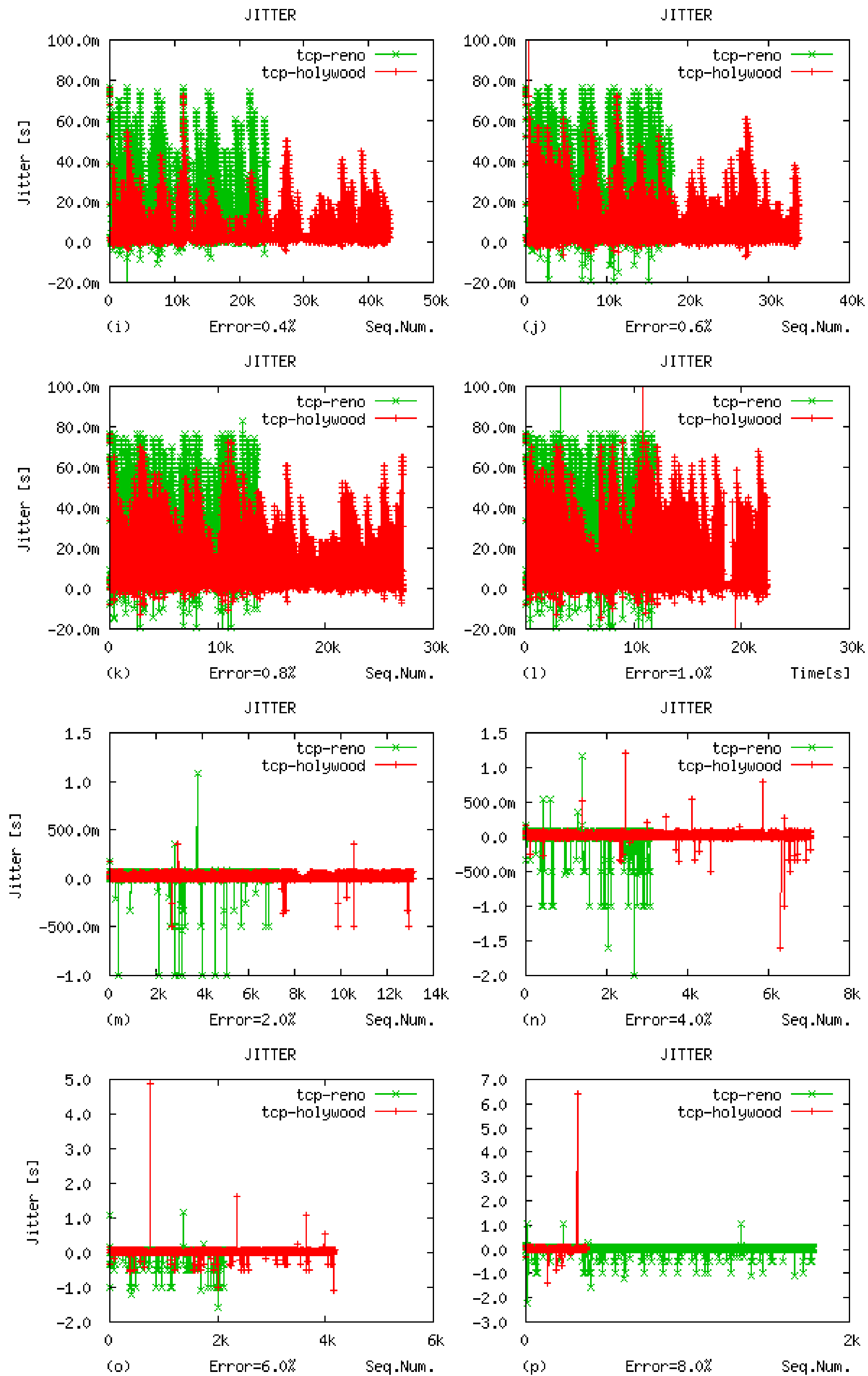

Figure C.1.4.2: Impact of Error Rate on Jitter (2 of 3)

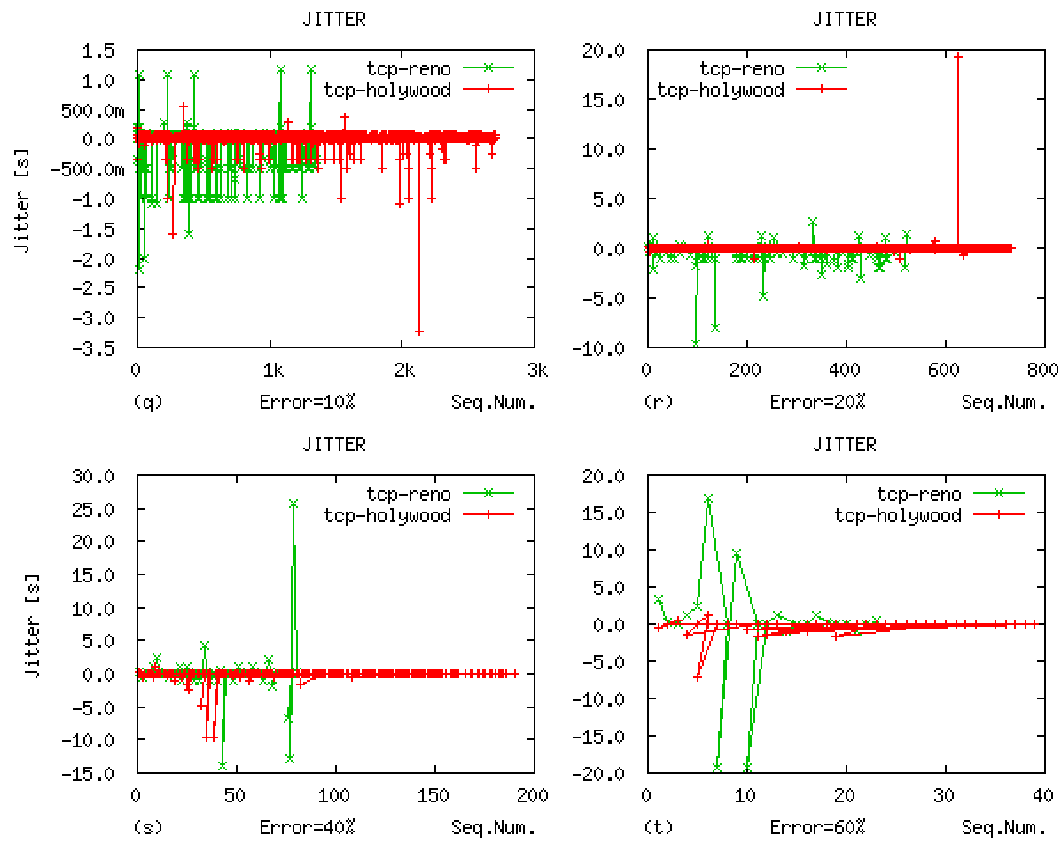

Figure C.1.4.3: Impact of Error Rate on Jitter (3 of 3)

C.1.5 Impact of Propagation Time on jitter

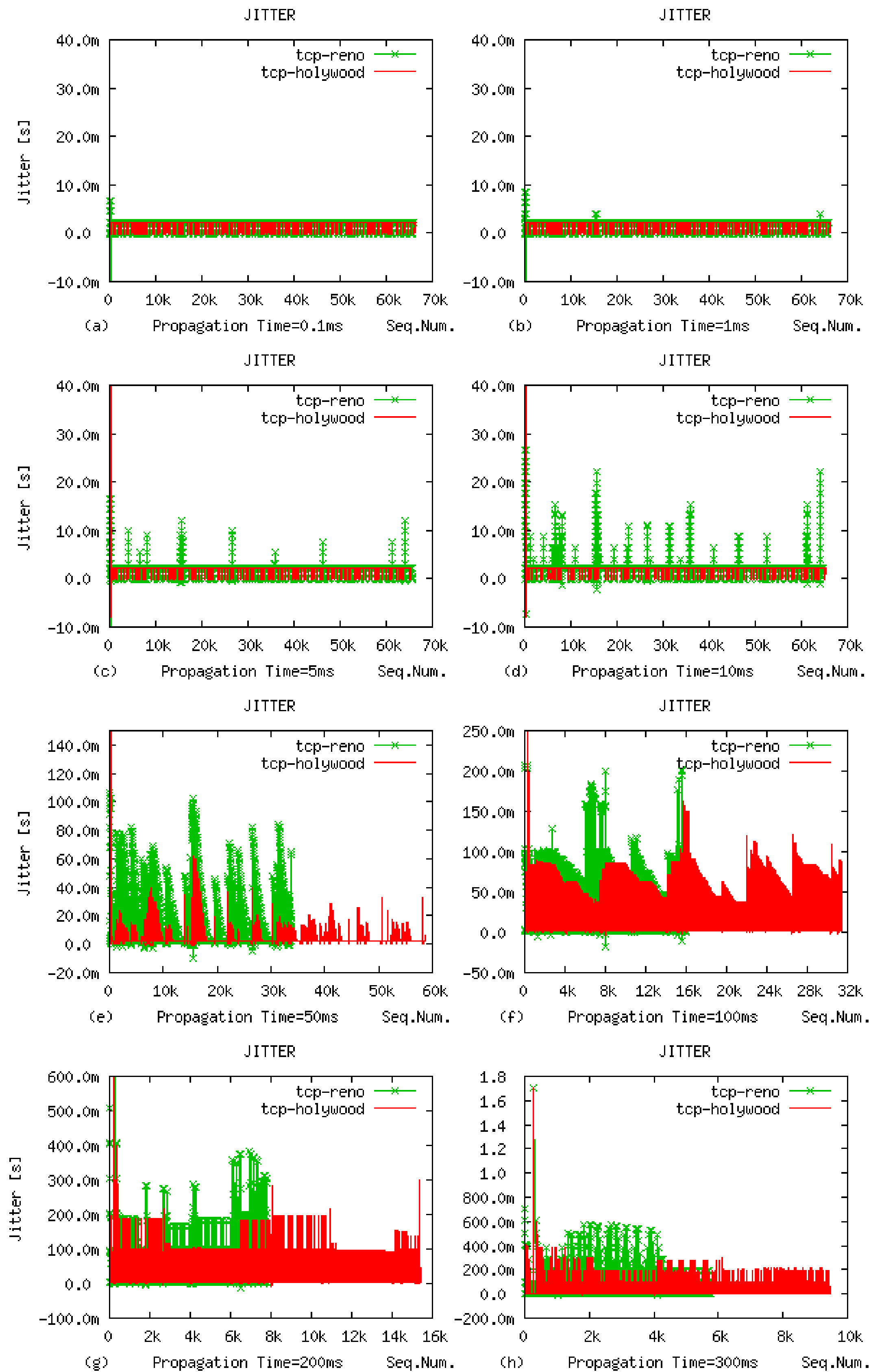

Figure C.1.5.1: Impact of Propagation Time on Jitter (1 of 2)

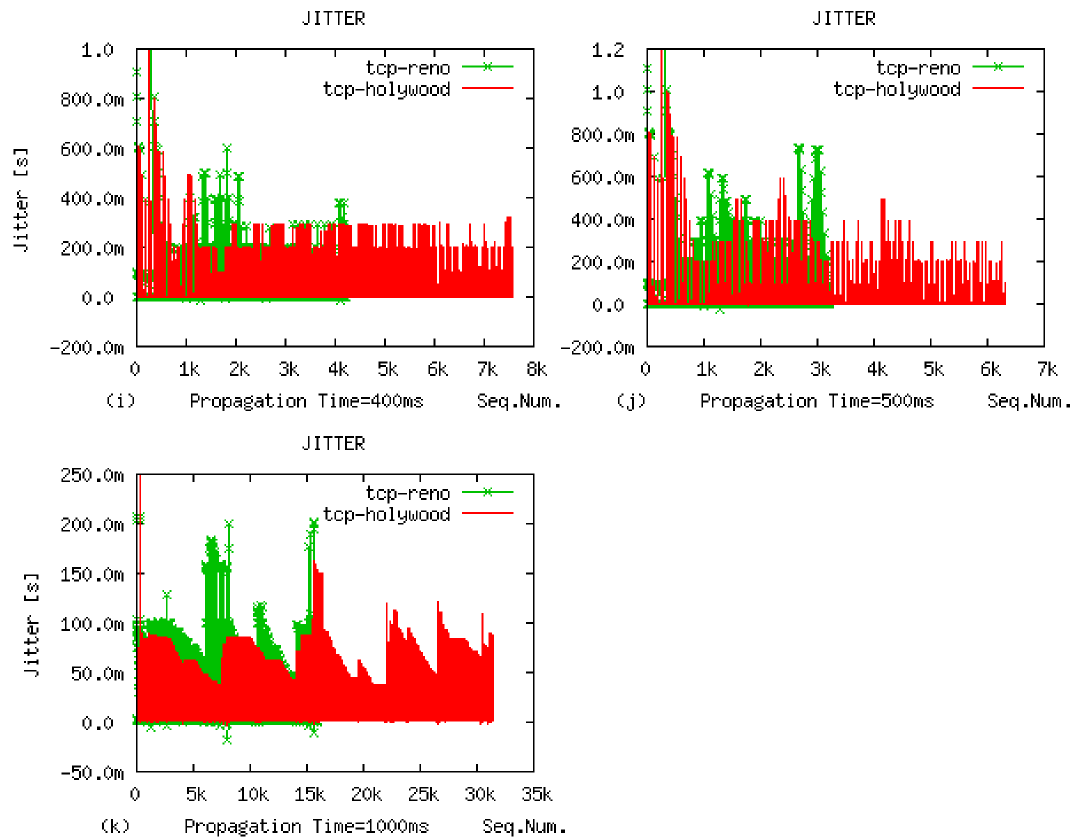

Figure C.1.5.2: Impact of Propagation Time on Jitter (2 of 2)

C.1.6 Impact of Bottleneck Bandwidth on Jitter

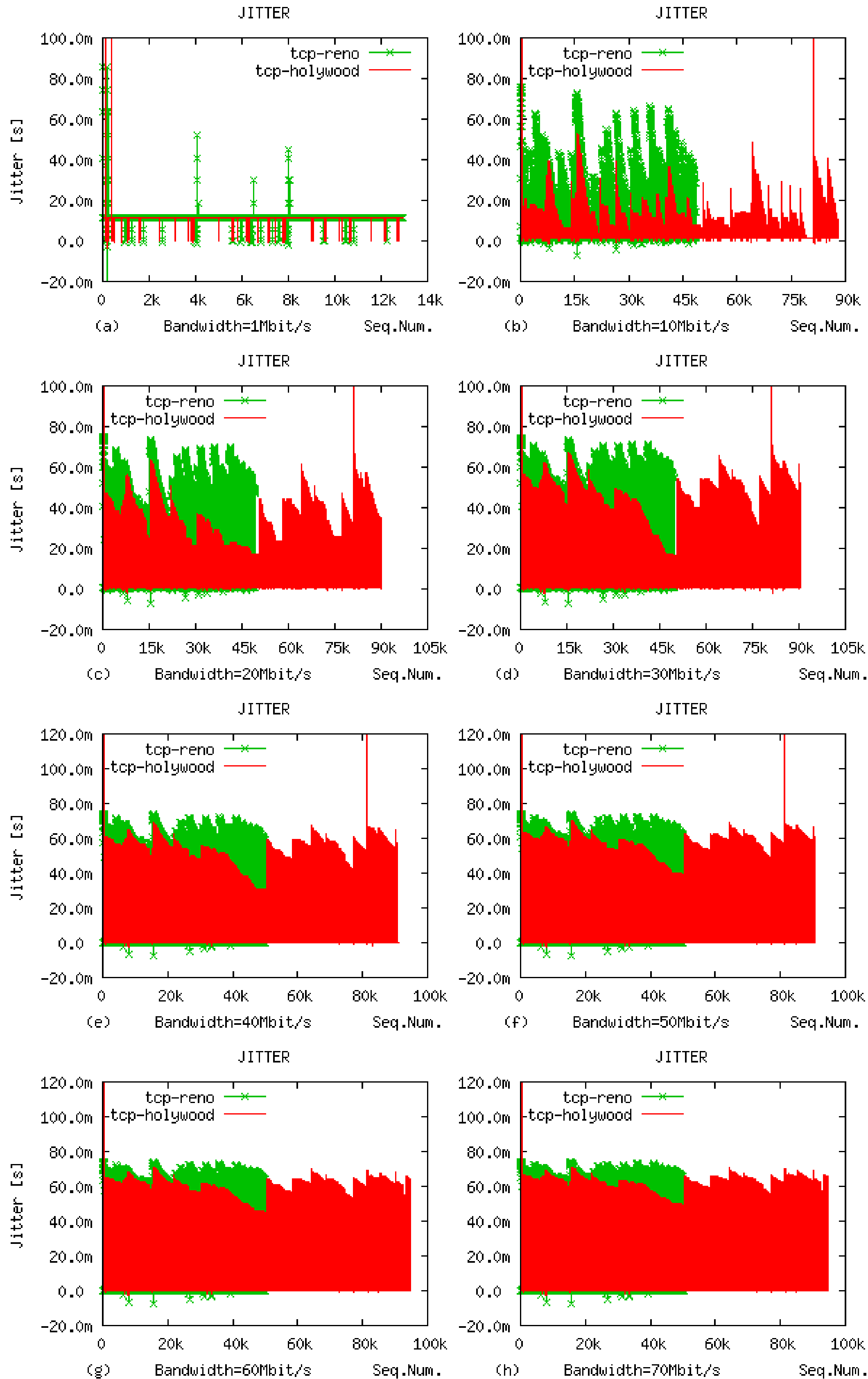

Figure C.1.6.1: Impact of Bottleneck Bandwidth on Jitter (1 of 2)

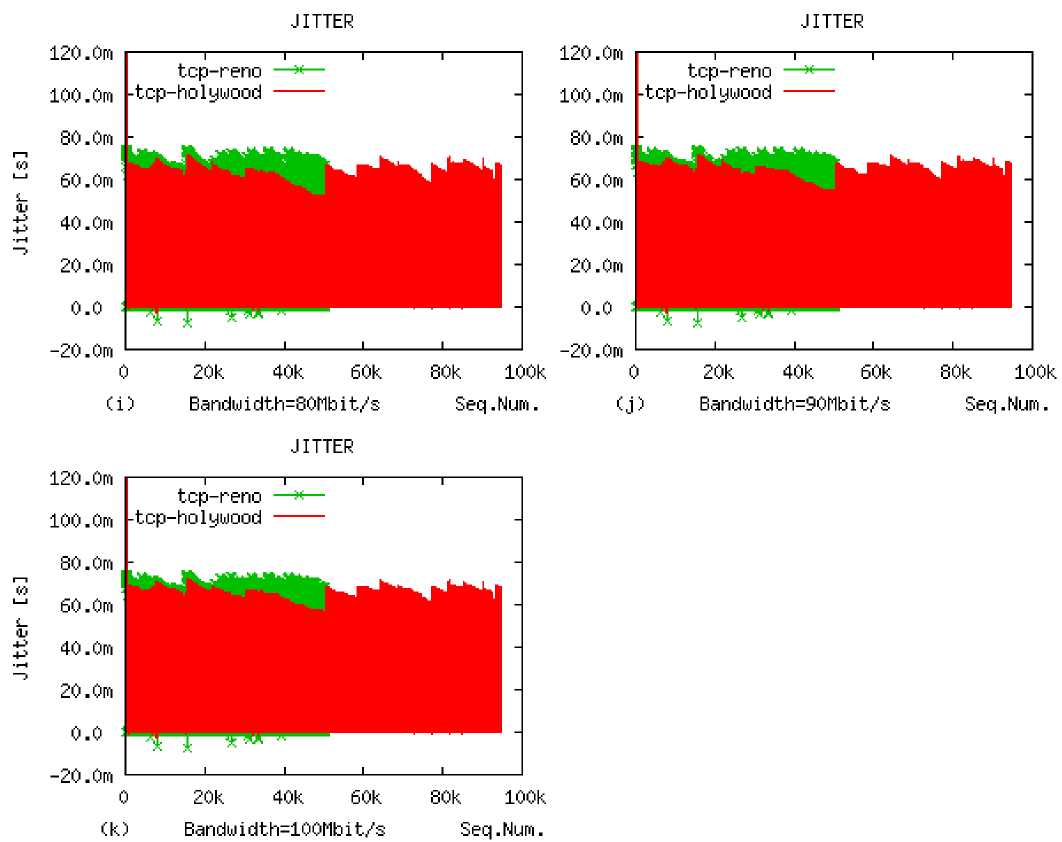

Figure C.1.6.2: Impact of Bottleneck Bandwidth on Jitter (2 of 2)

C.2 TCP HolyWood versus Other Protocols

C.2.1 Impact of Error Rate on throughput of TCP HolyWood, TCP Westwood and TCP Vegas

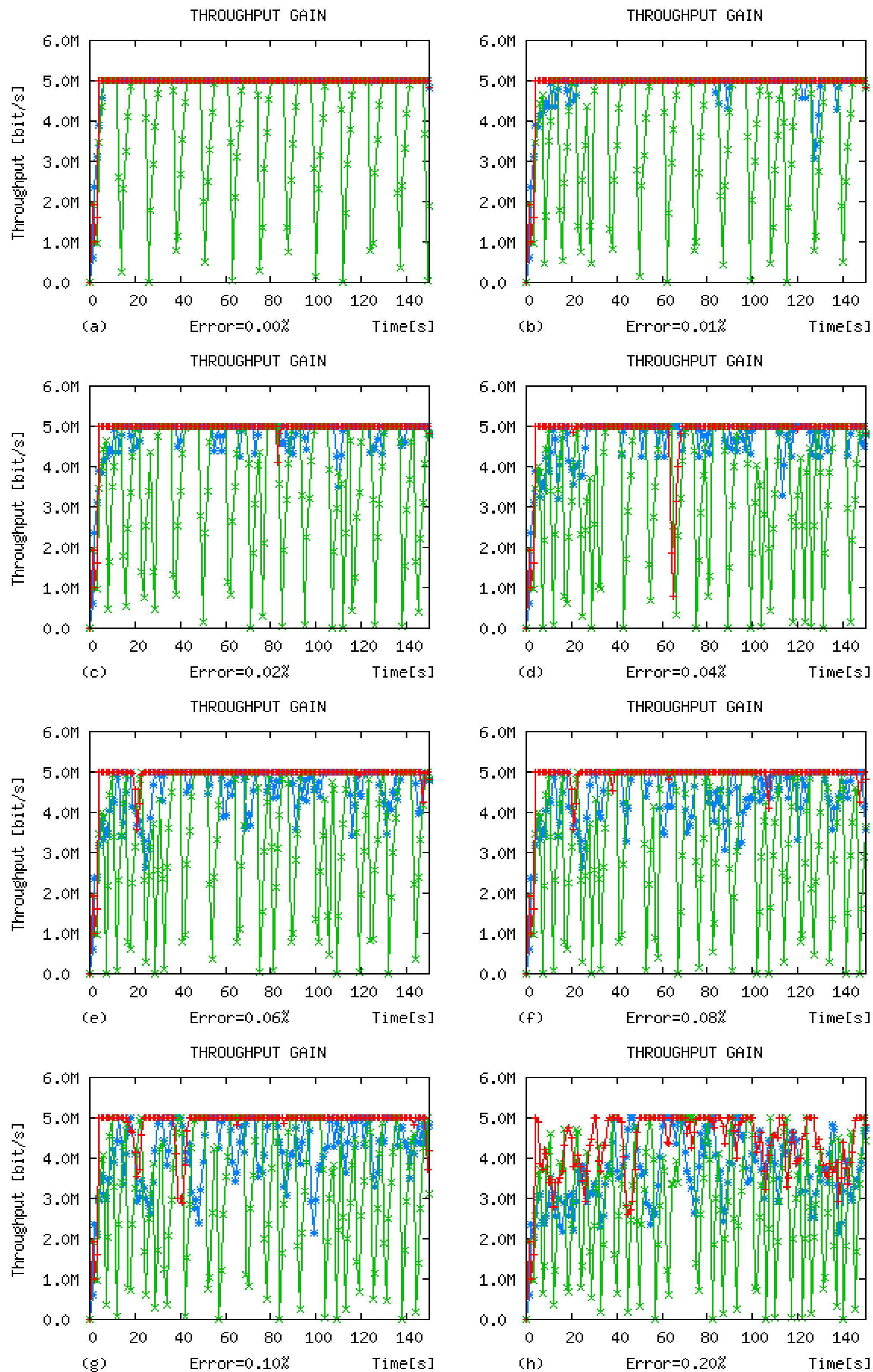

Figure C.2.1.1: Impact of Error Rate on throughput of TCPs (1 of 3)

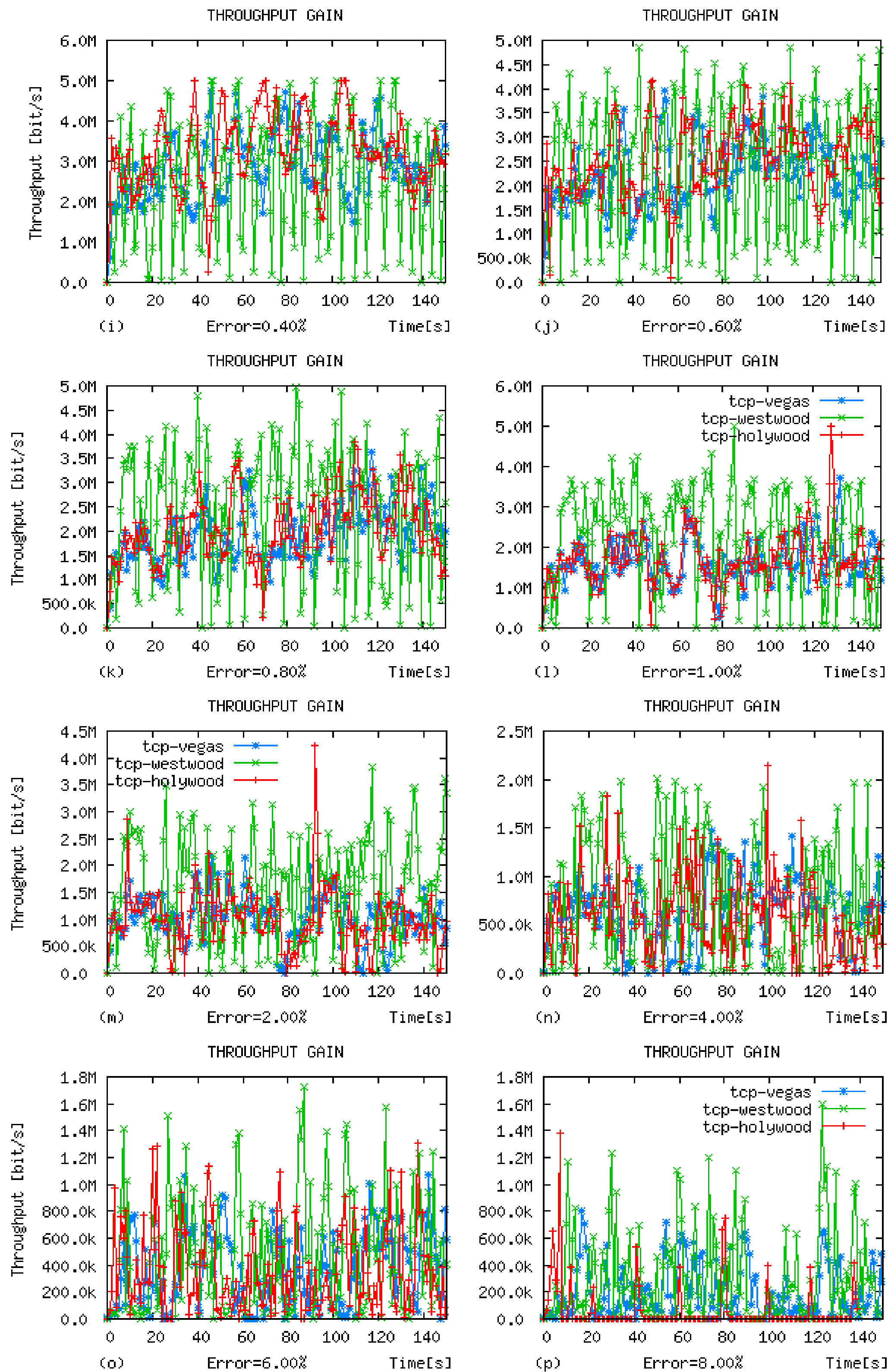

Figure C.2.1.2: Impact of Error Rate on throughput of TCPs (2 of 3)

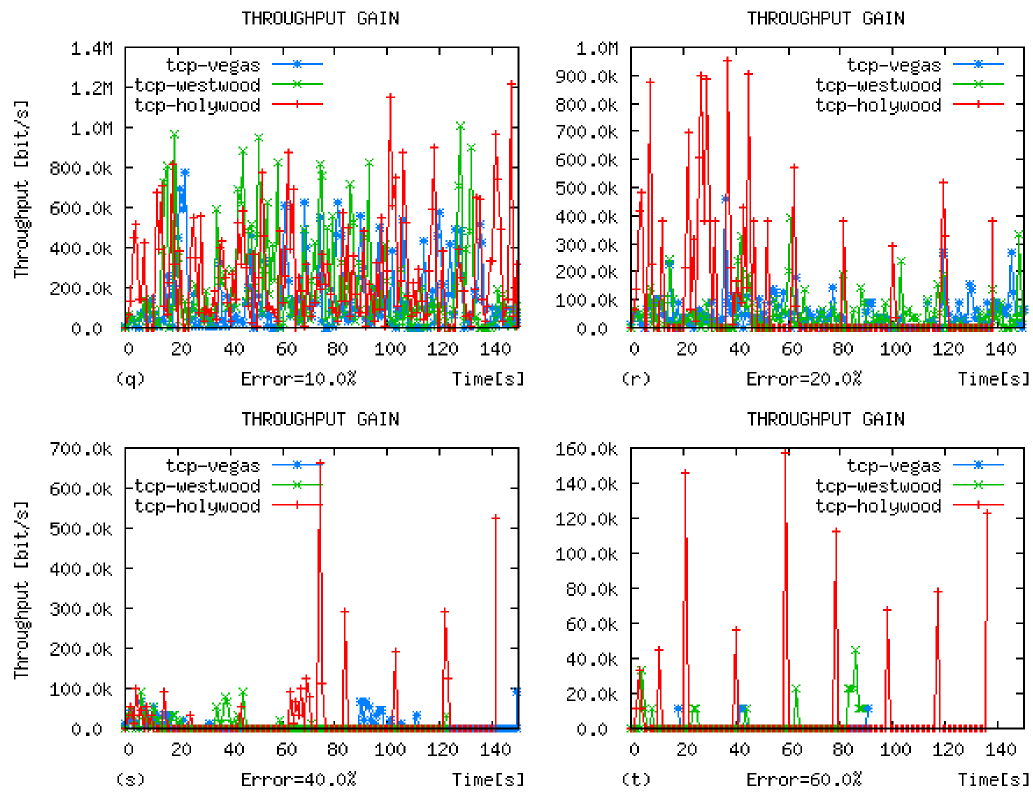

Figure C.2.1.3: Impact of Error Rate on throughput of TCPs (3 of 3)

C.2.2 Impact of Propagation time on throughput of TCP HolyWood, TCP Westwood and TCP Vegas

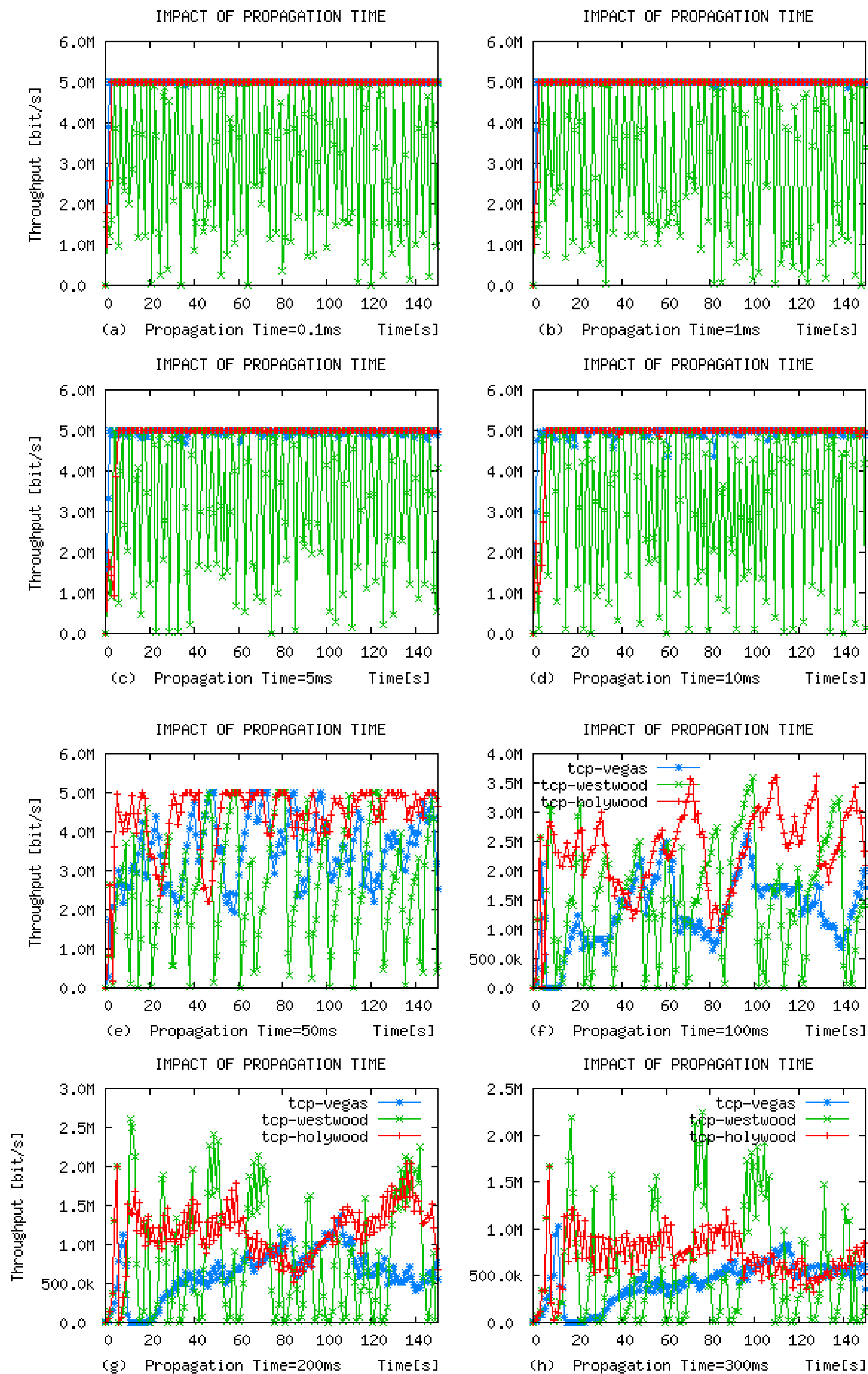

Figure C.2.2.1: Impact of Propagation time on throughput of TCPs (1 of 2)

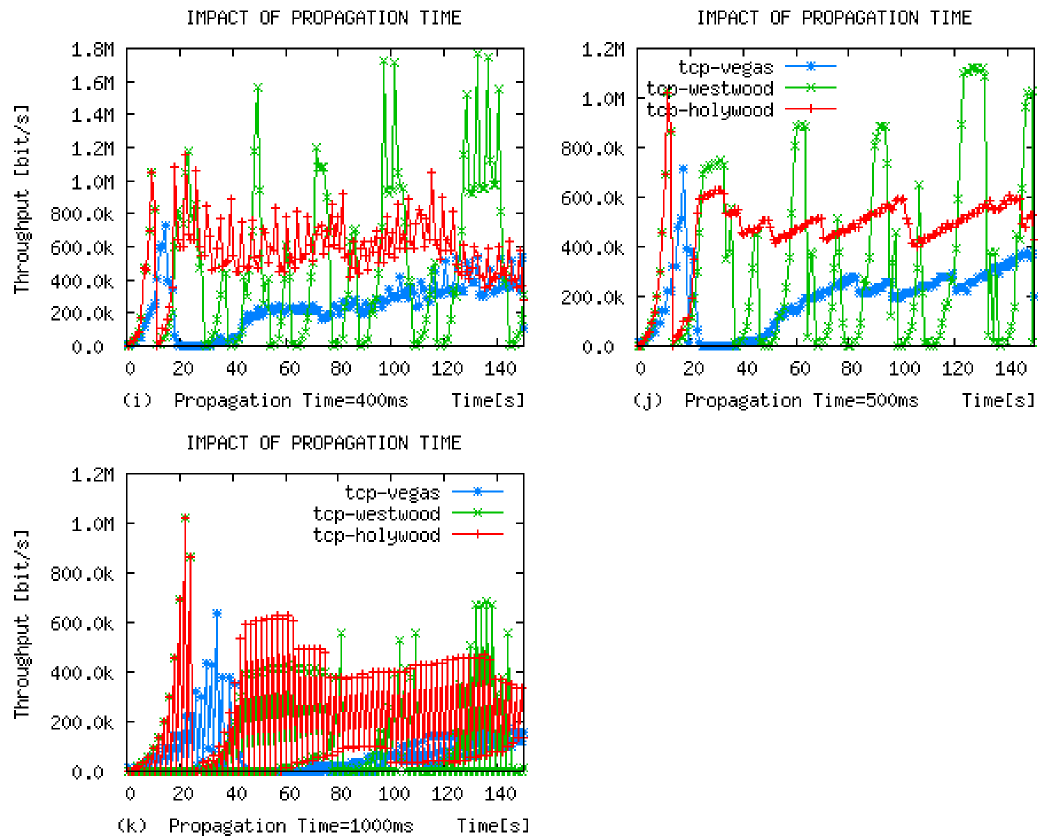

Figure C.2.2.2: Impact of Propagation time on throughput of TCPs (2 of 2)

C.2.3 Impact of Bottleneck Bandwidth on throughput of TCP HolyWood, TCP Westwood and TCP Vegas

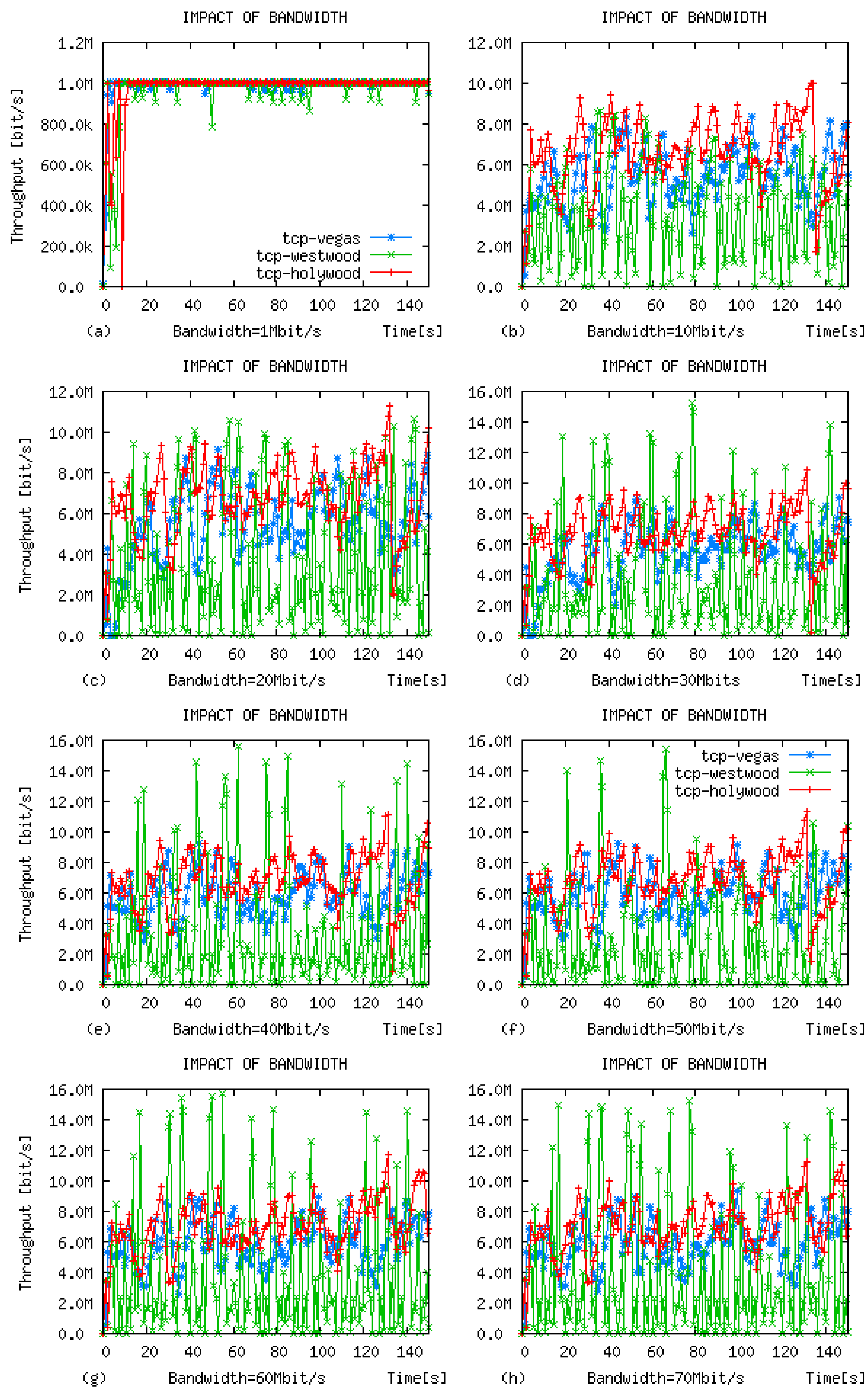

Figure C.2.3.1: Impact of Bottleneck Bandwidth on throughput of TCPs (1 of 2)

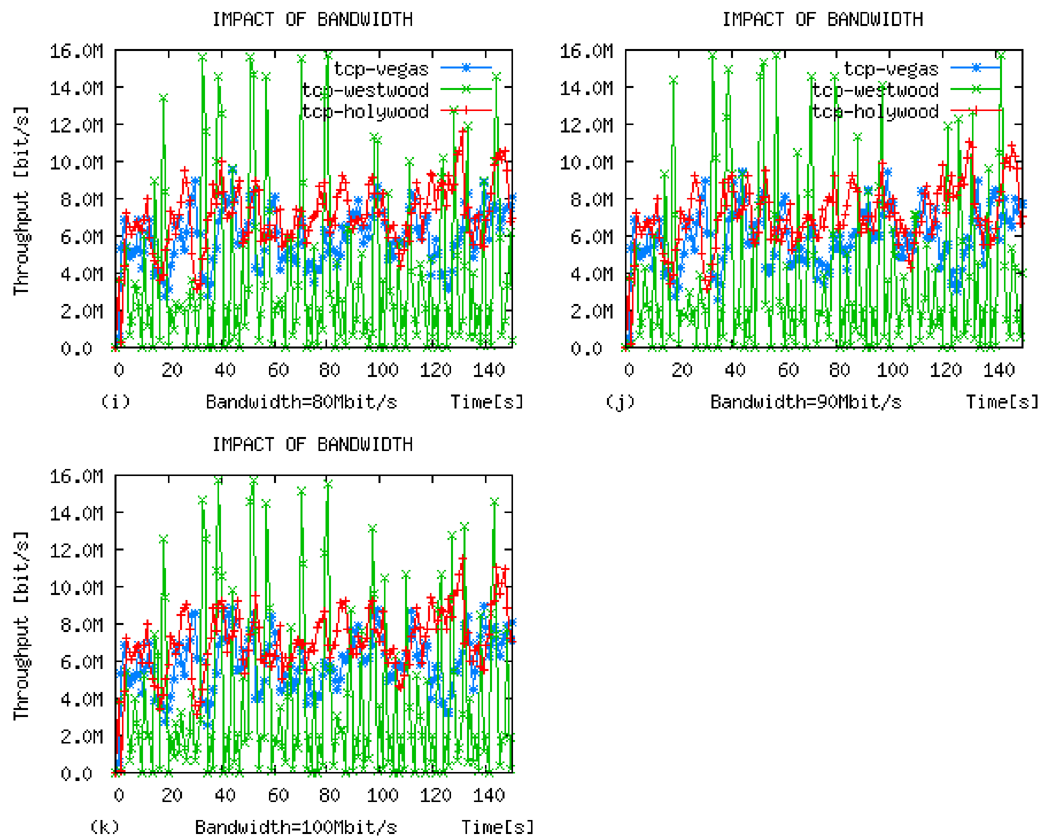

Figure C.2.3.2: Impact of Bottleneck Bandwidth on Throughput of TCPs (2 of 2)

C.2.4 Impact of Eror Rae on Jitter of TCP HolyWood, Westwood and Vegas

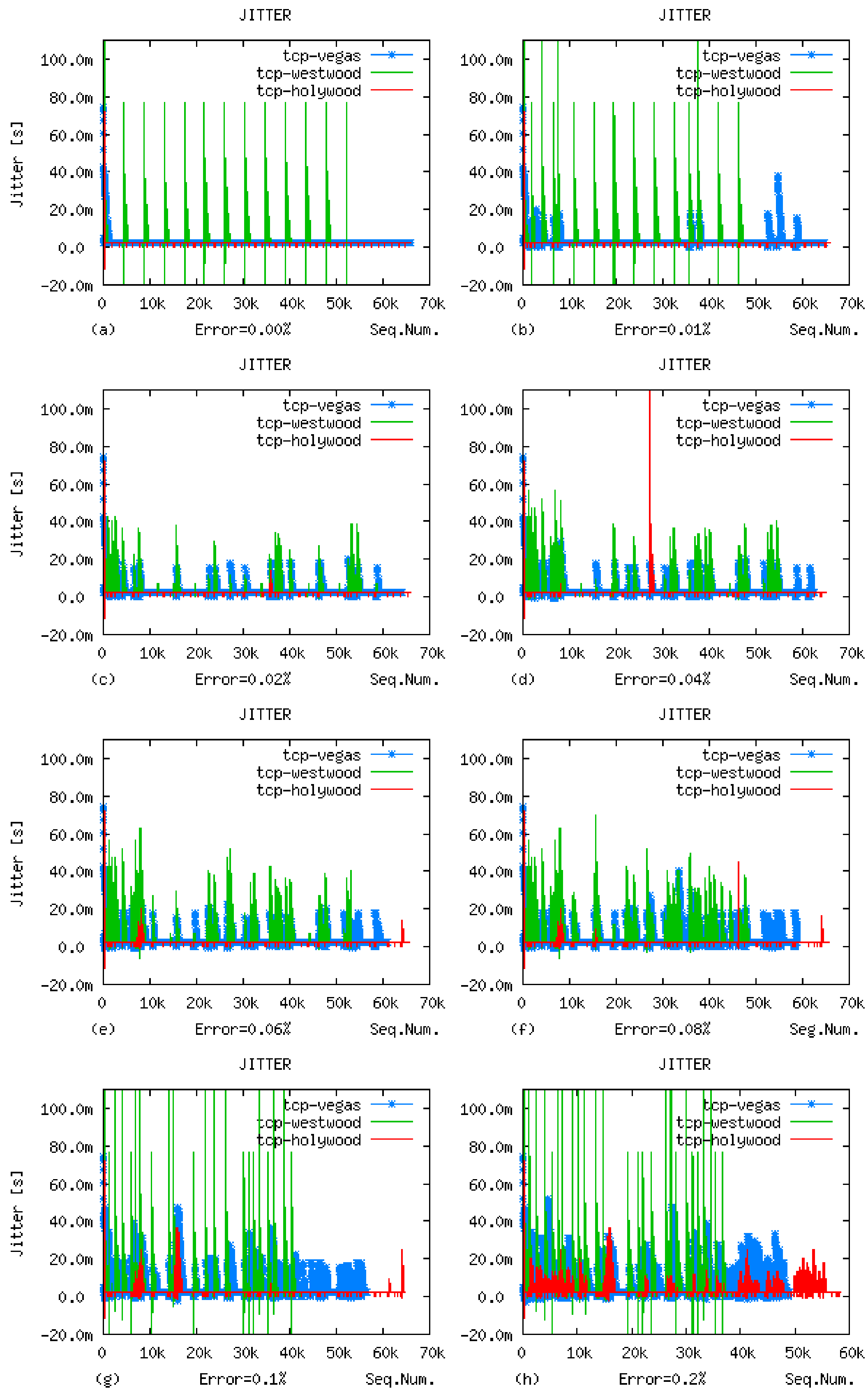

Figure C.2.4.1: Impact of Error rate on Jitter of TCPs (1 of 3)

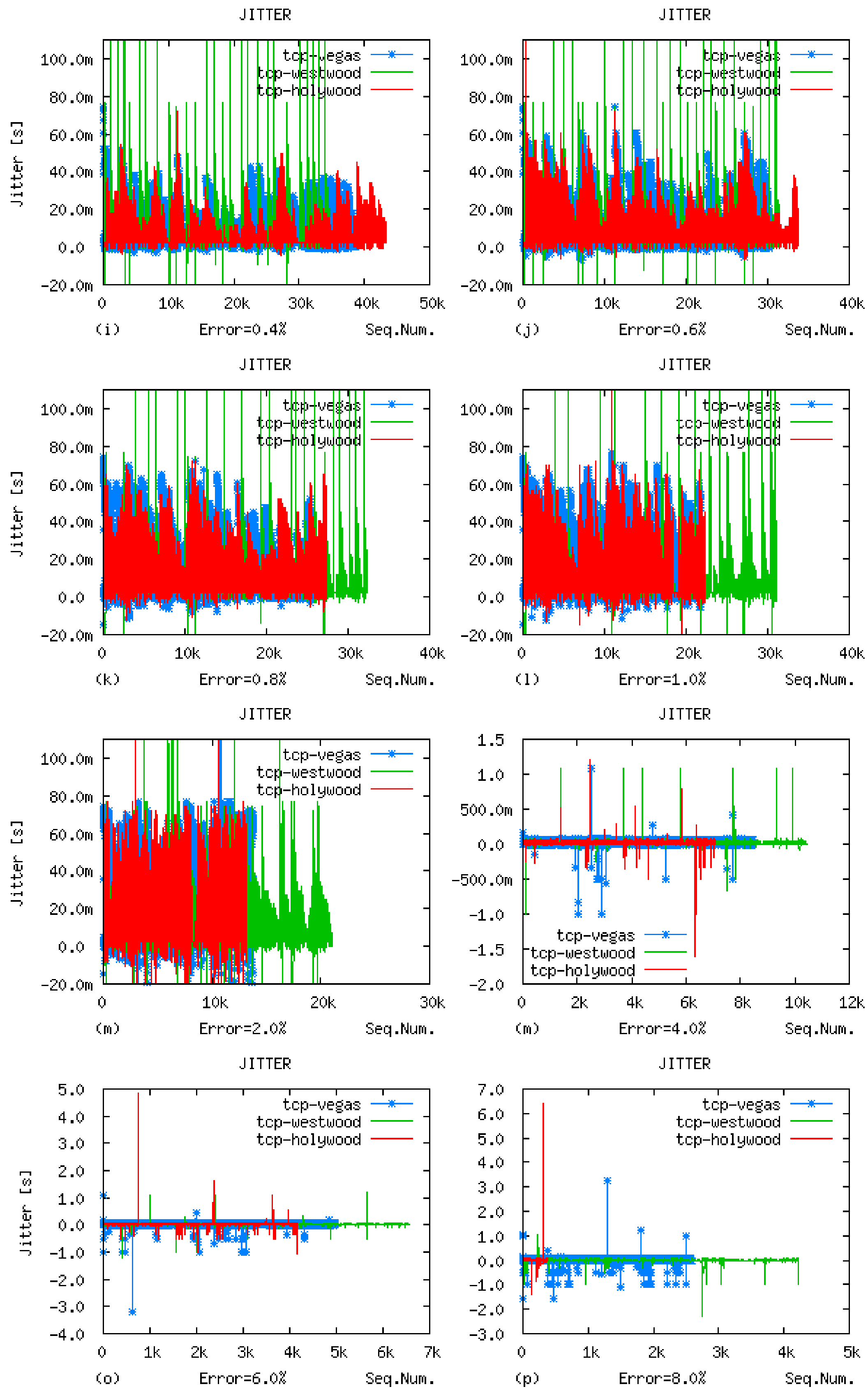

Figure C.2.4.2: Impact of Error rate on Jitter of TCPs (2 of 3)

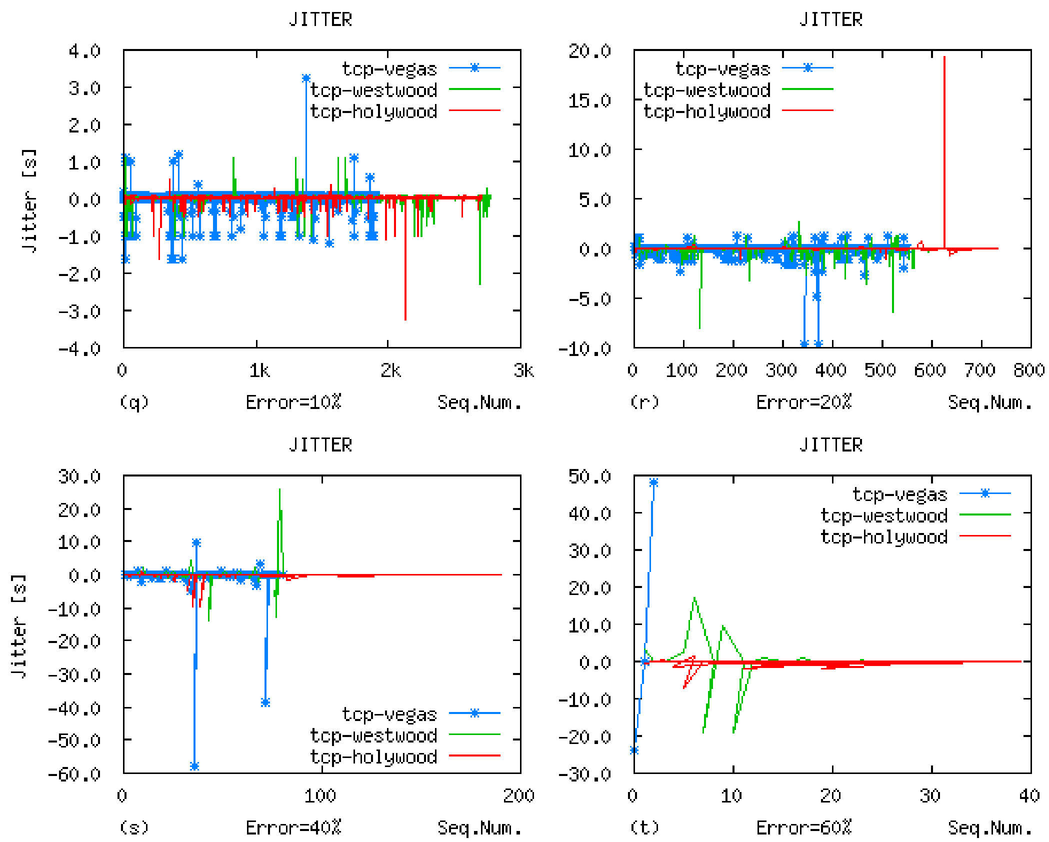

Figure C.2.4.3: Impact of Error rate on Jitter of TCPs (3 of 3)

C.2.5 Impact of Prop. Time on Jitter of TCP HolyWood, Westwood and Vegas

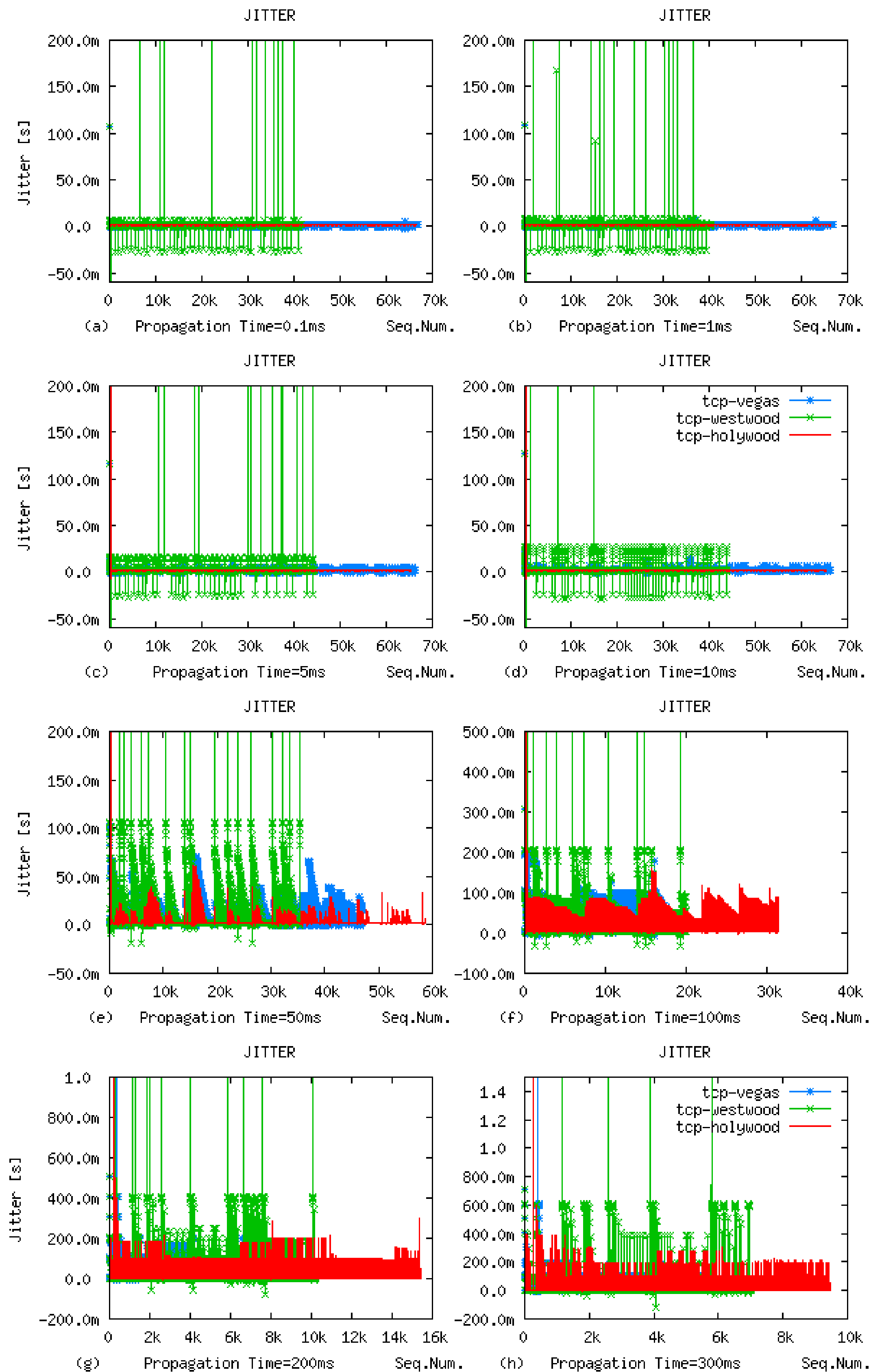

Figure C.2.5.1: Impact of Propagation Time on Jitter of TCPs (1 of 2)

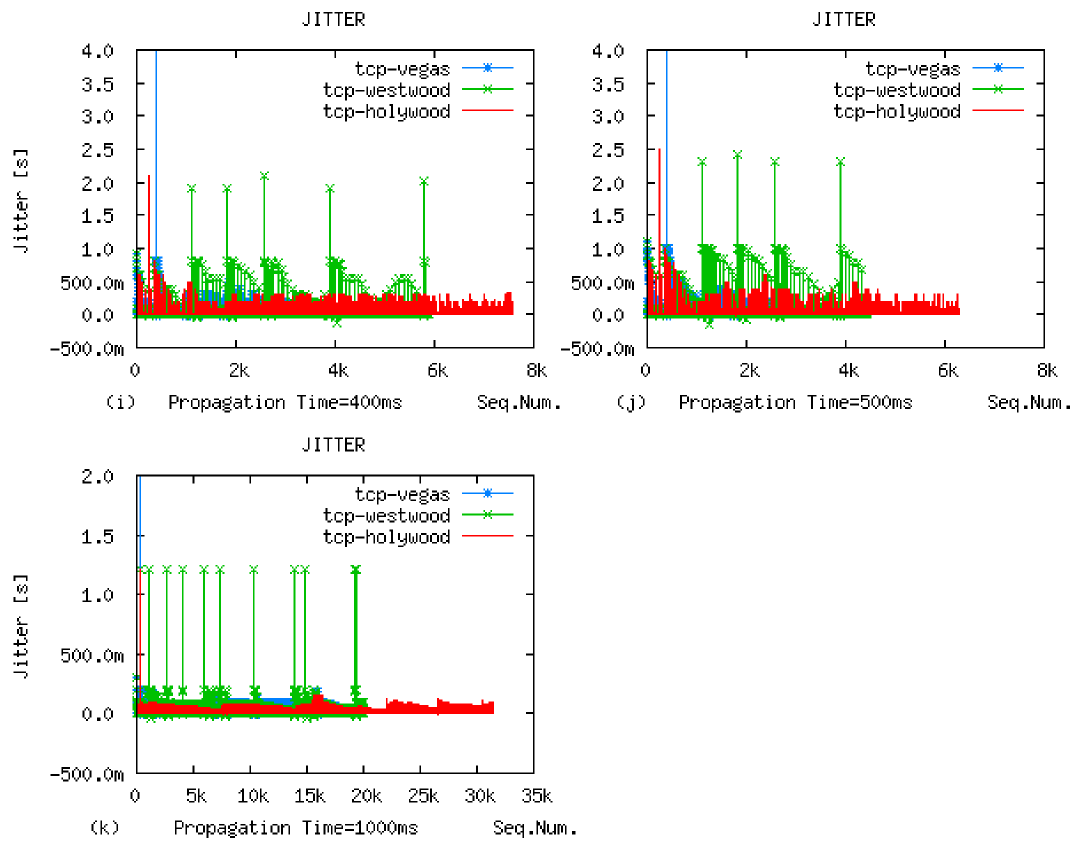

Figure C.2.5.2: Impact of Propagation Time on Jitter of TCPs (2 of 2)

C.2.6 Impact of Bottleneck Bandwidth on Jitter of TCP HolyWood, TCP Westwood and TCP Vegas

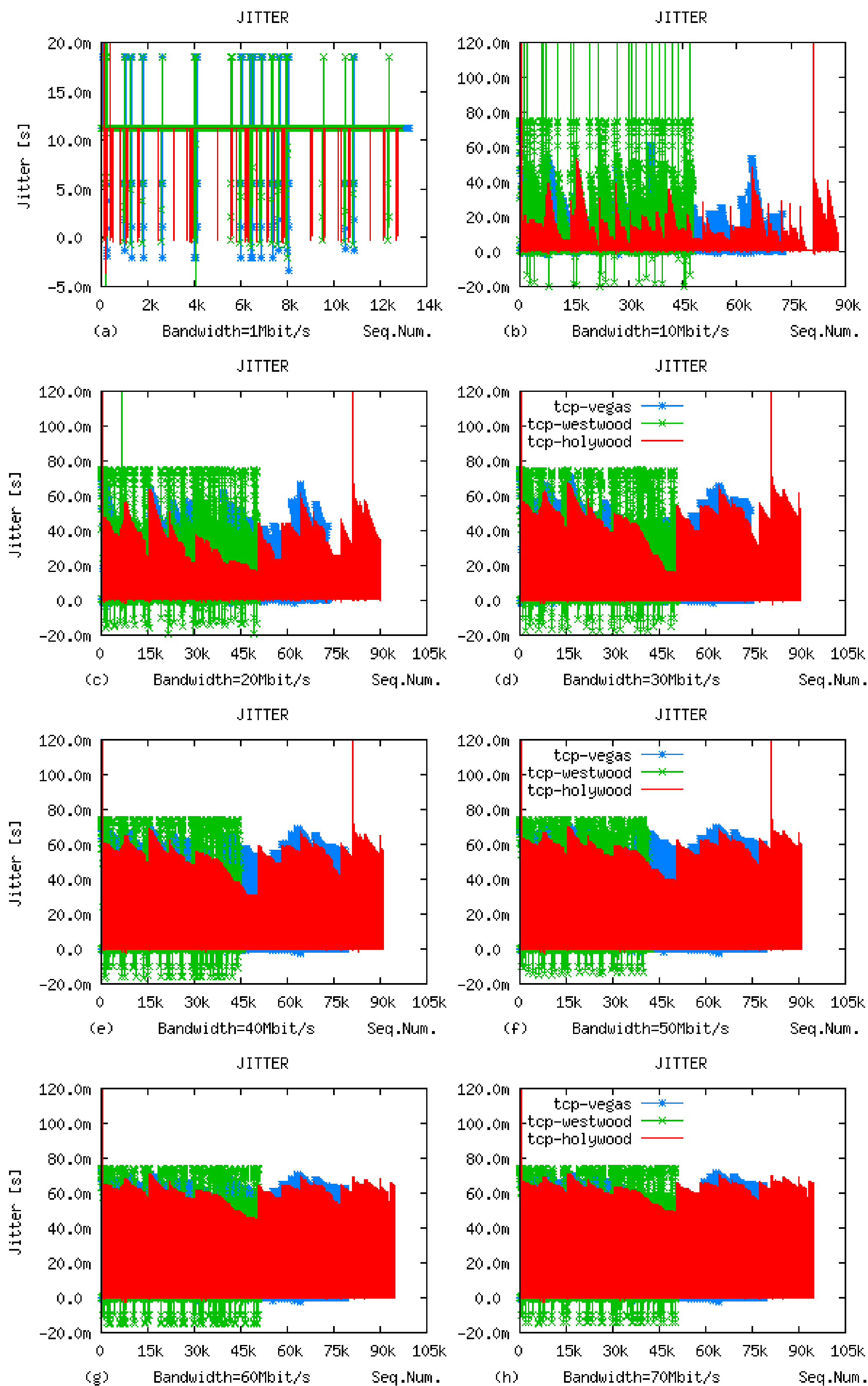

Figure C.2.6.1: Impact of Bottleneck Bandwidth on Jitter of TCPs (1 of 2)

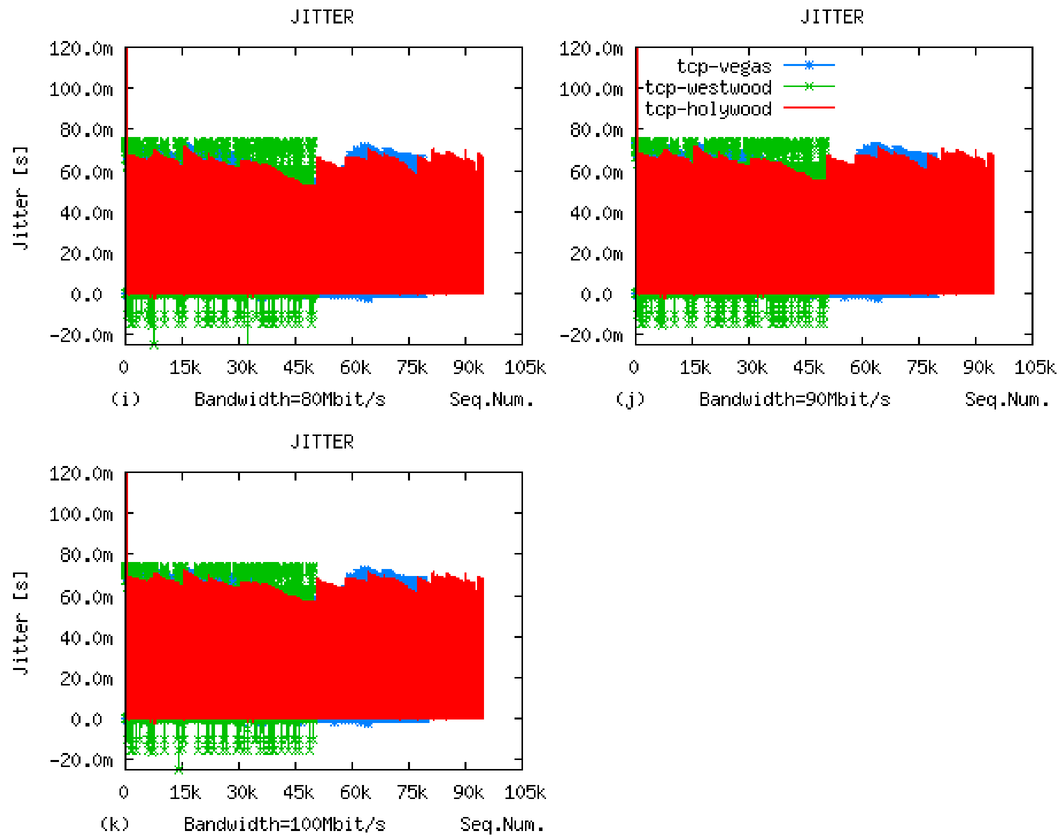

Figure C.2.6.2: Impact of Bottleneck Bandwidth on Jitter of TCPs (2 of 2)

APPENDIX D SOFTWARE INSTALLATIONS

D.1 Installation of NS-2.1b8a with TCP HolyWood in FreeBSD 4.10

We install the Network Simulator ns-2 in the FreeBSD 4.10 Operative System as follows:

1. Download the file "ns-allinone-2.1b8a.tar.gz" from the Source, section Getting Older Versions of NS in:

   < http://www.isi.edu/nsnam/dist/>

   [ns-allinone-2.1b8a.ta+ 26-Jun-01 09:33 49M]

   and with root attribute do the follow:

   > cd */usr/local*
   > tar xvzf ns-allinone-2.1b8a.tar.gz (ns-allinone-2.1b8a directory will be created)

2. */usr/local/ns-allinone-2.1b8a/tcl8.3.2/generic*
   > touch tclStuInit.c
3. In /usr/localns-allinone-2.1b8a/
   > ./ install
4. Add in .cshrc file of your personal account, the default values that are shown after install ns-2.
5. In */usr/local/ns-allinone-2.1b8a/ns-2.1b8a*
   > make clean
   > ./conFigure - -enable-debug
   > (it creates a new make file)
6. Modify here “Makefile” already created adding
7. OBJ_CC = ...
   1. ....
   2. TCP-holywood.o \
   3. TCP-westwood.o \ (optional)
   4. ....
   5. ....
8. Copy TCP-holywood.{cc,h} and TCPwestwood.{cc,h} (optional)

   > in */usr/local/ns-allinone-2.1b8a/ns-2.1b8a/*
9. Modify ns-default.tcl in */usr/local/ns-allinone-2.1b8a/ns-2.1b8a/tcl/lib* as a requeriment of TCP westwood (Optional)

10. Using an editor as pico, vi, or emacs
    > In the configuration file ".cshrc" of the User, for example Oscar or */usr/local* directory or any other name, (because we are using TCSH shell) add the following 3 lines:

    set path =(... /usr/local/ns-allinone-2.1b8a/bin /usr/local/ns-allinone-2.1b8a/tcl8.3.2/unix … /usr/local/ns-allinone-2.1b8a/tk8.3.2/unix ...)

setenv LD_LIBRARY_PATH /usr/local/ns-allinone-2.1b8a/otcl-1.0a7:/usr/local/ns-allinone-2.1b8a/lib

setenv TCL_LIBRARY /usr/local/ns-allinone-2.1b8a/tcl8.3.2/library

After that save it, it is advisable to make a backup copy of ".cshrc"

11. After that do:

    > source .cshrc
    > env (to see your changes)

12. In */usr/local/ns-allinone-2.1b8a/ns-21b8a/* do:

    > make
    >./ validate

D.2 Installation of Trace Graph 2.02

We installed the Trace graph tool in the operative system FreeBSD 4.10 and then do the following steps:

1. First we downloaded two files from

   http://www.geocities.com/tracegraph/
   In order to do that, we have to fill in a form before downloading Trace graph. The files are:

   - Trace graph 2.02 Linux version 966 894 bytes (tracegraph202linux.tar.gz)
   - and the Matlab 6.1 Run Time Libraries 8 247 599 bytes (mglinstaller.gz)

   If there are problems with permissions as a Root user do:
   > chmod 771 tracegraph202linux.tar.gz
   > chmod 771 mglinstaller.gz

2. Now as a simple user decompress the first file with tar command as follows:

   > tar xvzf tracegraph202linux.tar.gz

   In addition, the directory ‘‘tracegraph202’’ should be created together with the Binaries ready to use. However, they require the MATLAB Math and Graphics Run-Time Libraries, (MGRTL)

3. You may copy "mglinstaller" where you want with the proper path, but for simplicity we copy it into tracegraph202 directory:

   > cp mglinstaller.gz tracegraph202/

   Then we decompressed it with gunzip command

   gunzip mglinstaller.gz

   Moreover, the mglinstaller files have to appear.

4. Because this software was intended for linux, we used a linux emulator in FreeBSD to unpack the MGRTL libraries:

   > /compat/linux/bin/sh mglinstaller

that you will have a message:

Enter the directory to which to install the run-time libraries
[default: /home/oscar/tracegraph202]:

you may type MGRTL
MGRTL (MATLAB Math and Graphics Run-Time Libraries) and the directory MGRTL is created.

5. Now, we have to configure the ".cshrc" file because we use shell TCSH

   Adding:
   setenv LD_LIBRARY_PATH YOUR_PATH_USER/tracegraph202/MGRTL/bin/glnx86
   for example:
   setenv LD_LIBRARY_PATH /usr/home/oscar/tracegraph202/MGRTL/bin/glnx86
   or
   setenv LD_LIBRARY_PATH /usr/local/ns-allinone-2.1b8a/otcl1.0a7:/usr/local/ns-allinone2.1b8a/lib:/usr/home/oscar/MGRTL/bin/glnx86
   In the last example MGRTL is together with ns-2 configuration settings.

7. After that do:

   > source .cshrc
   > env (to see your changes)

   If in LD_LIBRARY_PATH exist other paths separete them with ":"

8. Before we forget it, a last setting is important, in file *~/tracegraph202/trgraph.cfg* change

   MAXIMAL NUMBER OF LINES=1000 to
   MAXIMAL NUMBER OF LINES=10000000 (or more)

9. For more information, in the directory “tracegraph202/doc” read install.txt and help.txt.

10. Finally, for easy use for simplicity, it is good to create a directory WORK inside the Trace graph tool, as follows:

    > /usr/home/oscar/tracegraph202
    > mkdir WORK

    in this directory you may organize your FILES.tr and FILES.mat and any time You want to use Tracegraph do as follows:
    Inside the directory */usr/home/oscar/tracegraph202* do:

    > ./trgraph WORK/FILE-NS-2.tr or
    > ./trgraph WORK/FILE-NS-2.mat
    or
    > ./trgraph ANYWHERE-FILE.tr or
    > ./trgraph ANYWHERE-FILE.mat

    Because if you start Trace graph outside the next message will appear:

    ERROR: Configuration file does not exist!
    Default configuration has been set.

For example:

```
Sainte_Marie
/usr/home/oscar/tracegraph202
> ./trgraph  ~/temp/test-1-simple-150s-Err-0.001-MAT.mat
```

In this example, we had our processed Trace graph ".mat" files in a temp file outside of the "tracegraph202" directory, but we fire the Trace Graph tool with the comand *trgraph* from */usr/home/oscar/tracegraph202* directory.

APPENDIX E TCP HolyWood

O *Transmission Control Protocol* (TCP) é baseado nos conceitos descritos primeiramente por Cerf e Kahn (1974). A finalidade principal do TCP é fornecer um serviço de conexão fim-a-fim com confiabilidade. Tendo em vista que os protocolos da camada de rede IP não oferecem confiabilidade, a camada de transporte TCP deve garantí-la.

As primeiras versões do TCP possuíam somente um controle de fluxo através de um mecanismo de janela deslizante simples, sem nenhum controle de congestionamento. Não obstante, depois de uma série de colapsos na rede Internet, nos anos 80, procurou-se solucionar este problema. Em 1988, Van Jacobson apresentou o TCP Tahoe e, dois anos mais tarde, introduziu o TCP Reno. O TCP Reno é composto dos seguintes algoritmos: estimação da variação do RTT (*Round Trip Time*), RTO (*Round-trip Time-Out*) exponencial Backoff, algoritmo de Karn, *Slow-Start*, *Congestion-Avoidance*, *Fast Retransmit* e *Fast Recovery*, os quais são explicados em detalhe no capítulo 2.

Após o trabalho de Van Jacabson nos anos 90, e até nossos dias, diversas propostas para melhorias do TCP apareceram. O TCP transformou-se num foco contínuo de pesquisa por parte da comunidade de informática. Isto é evidenciado pelas diversas propostas que apresentamos no capítulo 3, como por exemplo, TCP Veno, TCP Westwood, TCP Santa Cruz, entre outros. Em todas estas propostas, o que se tenta melhorar é a vazão em diferentes ambientes de rede, não obstante o padrão *de-facto* continuar sendo o TCP Reno.

Com o objetivo de melhorar a vazão do TCP, apresentamos no capítulo 4 a nossa proposta, denominada "TCP HolyWood".

Como indicado por Stallings (2004, p.239), para se obter um bom desempenho em sistemas fim-a-fim em um ambiente de rede, o projeto e a execução do protocolo de transporte são ingredientes vitais. Seguindo esta orientação, começamos a fazer experimentos variando diferentes parâmetros do TCP Reno, no intuito de obter um maior ganho na vazão do TCP e um menor *jitter* fim-a-fim. Um aspecto importante que foi mantido em nossa proposta é manter todas as características dinâmicas do TCP Reno. As modificações introduzidas no protocolo TCP só afetam o lado do emissor. Assim, não há nenhuma necessidade de reconfigurar roteadores intermediários ou o receptor. Ainda que obtidos através de um modelo simulado do nosso protocolo, os resultados vistos no capítulo 6 são promissores; assim, podemos considerar nossa proposta como um ajuste fino do TCP Reno.

O modelo proposto, portanto, é derivado do TCP Reno, fazendo-se algumas calibrações a fim conseguir um desempenho mais elevado da vazão, sem, no entanto mudar a essência do TCP padrão. Nossas modificações foram as seguintes:

1. Foi modificado o algoritmo *Slow Start* do TCP Reno, alterando-se o *CWND = CWND + 1* para *CWND = CWND + 9/5*, para termos um *Slow Start* mais rápido. Usamos, portanto, 9/5 (ou 1.8) em vez de um. Este parâmetro não é um número mágico, mas fruto de diversos experimentos e medidas, até que se chegasse a ele. Supondo um ponto *SSTHRESH* de 65536 bytes, nossa proposta é mais rápida do que TCP Reno. Enquanto o TCP Reno requer 16 passos para chegar até o *SSTHRESH*, a nossa proposta requer somente 11 passos.
2. Foi introduzida uma alteração que torna mais rápido o *Slow Start* do TCP Reno, compensando-o na fase de *Congestion Avoidance*. Diminuímos o valor original do *CWND = CWND + 1/CWND* para *CWND = CWND + 1 (4.CWND).* A idéia por trás disto é ter aproximadamente uma função constante. Em outras palavras, procuramos aproximar a nossa curva de vazão de uma linha constante, tendendo ao valor máximo da largura de banda.
3. Depois de três *ACKs* duplicados o TCP Reno reduz o *SSTHRESH* à metade (50%) e a janela de congestionamento (*CWND*) à metade mais três (+3). O TCP HolyWood reduz o *SSTHRESH* a cinco sextos (83%) e a janela de congestionamento a cinco sextos mais três (+3), diminuindo a janela por um fator de 1/6.
4. Quando ocorre *timeout* devido ao congestionamento ou a uma taxa de erro mais elevada, o TCP Reno reduz o *SSTHRESH* à metade (50%) e ajusta o CWND a 1; o TCP HolyWood reduz o *SSTHRESH* a treze vigésimos (65%) e ajusta o CWND em 3. Decidimos usar um aumento de 15% mais do que o TCP Reno por não saber se a congestão continua depois que ocorre um *timeout*. Podemos inferir que há uma probabilidade igual. Apesar de tudo, se o congestionamento persistir, o outro *timeout* (intervalo de parada) ocorrerá e o tempo deste *timeout* dobrará. Decidimos também, ajustar o *CWND* em 3, no lugar de 1, depois que o *timeout* termina e quando um *Slow Start* começa, baseado no RFC 2414 (ALLMAN; FLOYD; PERDIZ, 1998). A figura E.1 mostra de forma comparativa o desempenho do TCP HolyWood com o TCP Reno.

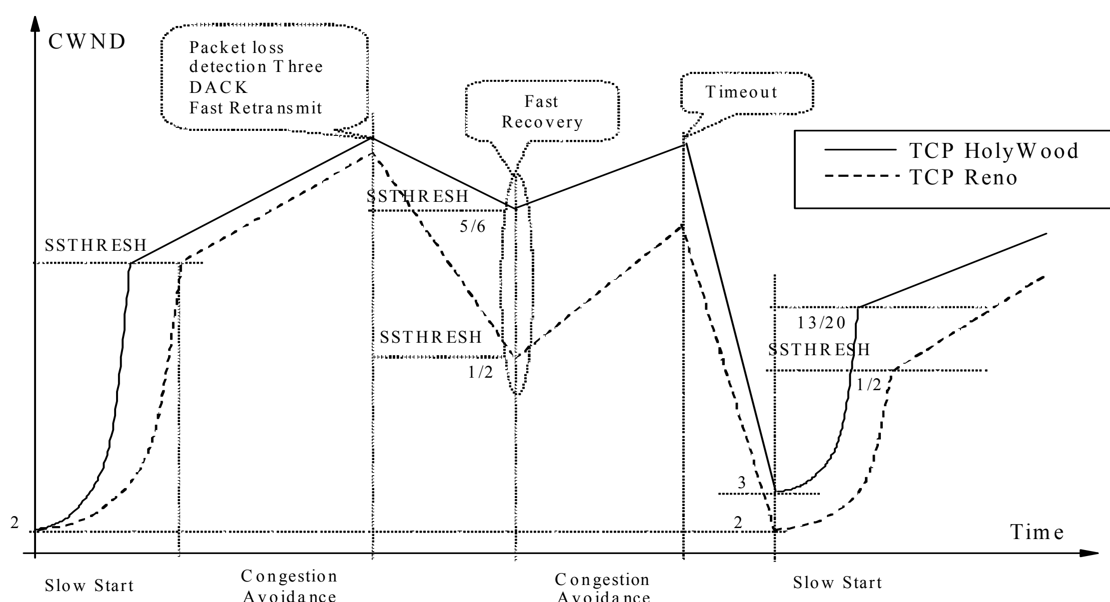

Figura E.1: Comparativo do TCP HolyWood com o TCP Reno.

No capítulo 5, apresentam-se as técnicas de avaliação de desempenho que foram aplicadas à nossa proposta de protocolo, denominada TCP HolyWood. Devido a fatores como tempo limitado e recursos escassos, escolhemos a simulação como técnica de avaliação de desempenho. Tendo em vista sua larga aceitação na comunidade acadêmica e sua facilidade de uso, foi utilizada como ferramenta de simulação nesta dissertação o simulador de rede ns-2, versão ns-2.1b8a.

Usamos em nossas simulações dois cenários de rede, chamados de topologia n.1 e a topologia n.2, representadas nas figuras E.2 e E.3 respectivamente. A topologia n.1, da figura E.2, apresenta quatro pontos sendo que n_0 e n_3 são o emissor e o receptor de uma conexão TCP respectivamente, e n_1 e n_2 são roteadores de nó intermediários. Os enlaces de acesso dos terminais tem uma capacidade de 100 Mbit/s, enquanto o enlace entre os roteadores forma um gargalo, pois possui uma largura de banda de apenas 5 Mbit/s.

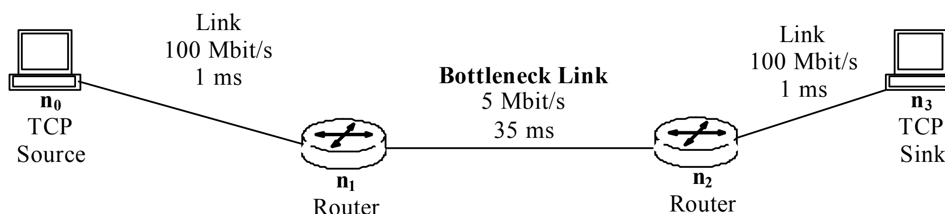

Figura E.2: Topologia simulada n.1

Na figura E.3 apresenta-se a topologia n.2, que na verdade é uma extensão da topologia n.1, adicionando-se dez (10) transmissores, os nós n_{101}, n_{102}..., n_{110}, e dez (10) receptores, n_{201}, n_{202}..., n_{210}. Cada um dos processos de fonte (*source*) TCP está ligado a seu respectivo agente emissor TCP através de um enlace de 15 Mbit/s via nó n_0. No lado remoto encontramos os dez processos de recepção (*sink*) TCP, também ligados ao agente TCP através de enlaces de 15 Mbit/s via nó n_3. Os nós n_0, n_1, n_2, n_3, ao longo do caminho, são roteadores. Novamente temos um ponto de estrangulamento (*bottleneck*) formado pelo enlace de 5 Mbit/s entre os nós n_1 e n_2 respectivamente,

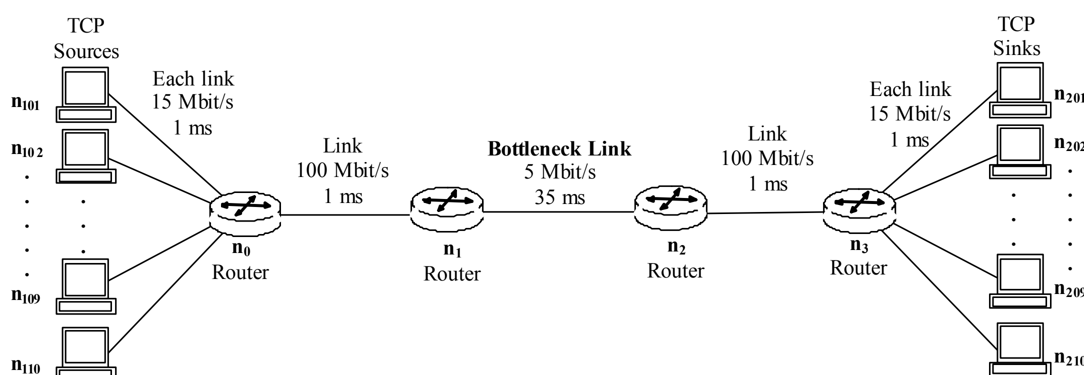

Figura E.3: Topologia simulada n.2

Foram utilizadas as seguintes condições de contorno nos cenários experimentais:

1. As simulações utilizaram pacotes de tamanho fixo.
2. Para fins de comparação, cada cenário experimental era aplicado ao TCP Westwood, TCP Reno, TCP Vega e TCP HolyWood.
3. Alterou-se um parâmetro por vez (taxa de erro, tempo de propagação ou largura de banda), para que se pudesse monitorar as mudanças entre os diferentes protocolos TCPs sob teste.

4. Foram usados os valores-padrão do simulador ns-2, versão ns-2.1b8a.
5. O tempo de simulação foi de 150 segundos para cada teste.

No capítulo 6 são apresentadas as curvas experimentais obtidas e as respectivas análises. Os resultados experimentais foram obtidos em relação aos padrões TCP Reno, TCP Vegas, TCP Westwood. As métricas principais foram vazão e *jitter*, e as condições de contorno experimentais foram taxa de erro, tempo de propagação e largura de banda. Foram usadas também as métricas definidas em Jain (1991), como justiça (*fairness*) e grau de amigabilidade (*friendliness*), no caso da comparação com o TCP Reno.

Finalmente, no capítulo 7 apresentam-se as conclusões e sugestões de trabalhos futuros, que podem ser sumarizadas como a seguir:

Quanto à Vazão

No intervalo da taxa de erro entre 0% a 60%, o TCP HolyWood apresentou um ganho na vazão média de 73.46% em relação ao TCP Reno, 30.65% em relação TCP Westwood e 67.46% em relação ao TCP Vegas.

No intervalo do tempo da propagação de 0.1 ms a 1000 ms, o TCP HolyWood obteve um ganho na vazão média de 53.59% sobre o TCP Reno, 47.76% sobre o TCP Westwood e de 66.42% sobre o TCP Vegas.

No intervalo da largura de banda de 1 Mbit/s a 100 Mbit/s, o TCP Holywood obteve um ganho na vazão média de 77.49% sobre o TCP Reno, de 76.7% sobre o TCP Westwood e de 17.71% sobre o TCP Vegas.

Conclui-se daí que, variando a taxa de erro, o tempo da propagação e a largura de banda em um ambiente de rede cabeada, e usando-se o simulador ns-2, o TCP HolyWood apresenta melhor desempenho quanto à vazão do que o TCP Reno, TCP Westwood e TCP Vegas.

***Jitter* (Variação de atraso)**

No intervalo da taxa de erro de 0% a 60%, quanto ao *jitter* em relação ao TCP HollyWood, o TCP Reno apresentou 38.28% a mais, o TCP Westwood apresentou 0.11% a menos, e o TCP Vegas apresentou a 265.32% a mais.

No intevalo de tempo de propagação de 0.1 ms a 1000 ms, o TCP Reno mostrou 52.49% mais *jitter* do que TCP HolyWood, o TCP Westwood mostrou 13.6% a mais, e o TCP Vegas mostrou 64.17% a mais.

No intervalo de largura de banda de 0 Mbit/s ate 100 Mbit/s, o TCP Reno mostrou 76.81% mais *jitter* do que TCP HolyWood, o TCP Westwood mostrou 8.95% a menos, e o TCP Vegas mostrou 17.55% a mais.

Conclui-se que, variando a taxa de erro, o tempo da propagação e a largura de banda faixa num ambiente de rede cabeado, e usando o simulador ns-2, a proposta de modelo apresenta melhor *jitter* médio em relação ao TCP Reno e ao TCP Vegas, tendo valor ligeiramente pior em relação ao TCP Westwood.

***Fairness* (Imparcialidade)**

Conclui-se que o TCP HolyWood é tão imparcial quanto TCP Reno.

***Friendliness* (Amizade)**

Conclui-se que, quando o número de TCPs Reno que competem com um único TCP HolyWood aumenta, este se torna mais amigável com os outros TCPs Reno.

Latência ou atraso

Em todos os testes comparativos feitos entre o TCP HolyWood e os demais, ele apresentou latência média mais elevada, porém, a diferença diminui quando a taxa de erro aumenta.

Pacotes Perdidos

No intervalo da taxa de erro de 0% a 60%, tendo o TCP HolyWood como base em todos os casos, o TCP Reno e o TCP Vegas apresentaram, respectivamente, 8.61% e 25.47% menos pacotes perdidos, enquanto que o TCP Westwood apresentou 57.21% mais pacotes perdidos.

Ou seja, o TCP HolyWood mostrou menor percentual de pacotes perdidos em comparação ao TCP Westwood, e maior percentual comparativamente ao TCP Vegas e TCP Reno, ao longo de todo o eixo da taxa de erro. É importante observar que os TCPs emitiram quantidades diferentes de pacotes para a mesma simulação.

Considerando todas as simulações realizadas, pode-se concluir que o TCP HolyWood poderia trabalhar eficientemente em um ambiente cabeado de rede, com menos de 8% de taxa de erro.

Para terminar este apêndice, nossa proposta será desejável segundo alguns pontos de vista. Em usabilidade: devido às modificações, pequenas, simples e fáceis de ajustar. Em interação: como observado na seção de resultados, a proposta é tão justa quanto TCP Reno, e pode trabalhar perfeita e harmoniosamente junto com o padrão. Em competência: a proposta mostrou um desempenho bom no que diz respeito à vazão e ao *jitter*. Certamente, isto convida a implementar a proposta em um sistema operacional real e a fazer novos testes em outros ambientes, como redes sem fio e redes heterogêneas.

The End