Improving TCP Performance over heterogeneous networks

M. A. Alnuem

PhD

2009

Improving TCP Performance over heterogeneous networks

The investigation and design of End to End techniques for improving TCP performance for transmission errors over heterogeneous data networks

Mohammed A. Alnuem

Department of Computing
University of Bradford

Thesis submitted for the Degree of

Doctor of Philosophy

2009

Abstract

Transmission Control Protocol (TCP) is considered one of the most important protocols in the Internet. An important mechanism in TCP is the congestion control mechanism which controls TCP sending rate and makes TCP react to congestion signals. Nowadays in heterogeneous networks, TCP may work in networks with some links that have lossy nature (wireless networks for example). TCP treats all packet loss as if they were due to congestion. Consequently, when used in networks that have lossy links, TCP reduces sending rate aggressively when there are transmission (non-congestion) errors in an uncongested network.

One solution to the problem is to discriminate between errors; to deal with congestion errors by reducing TCP sending rate and use other actions for transmission errors. In this work we investigate the problem and propose a solution using an end-to-end error discriminator. The error discriminator will improve the current congestion window mechanism in TCP and decide when to cut and how much to cut the congestion window.

We have identified three areas where TCP interacts with drops: congestion window update mechanism, retransmission mechanism and timeout mechanism. All of these mechanisms are part of the TCP congestion control mechanism. We propose changes to each of these mechanisms in order to allow TCP to cope with transmission errors. We propose a new TCP congestion window action (CWA) for transmission

errors by delaying the window cut decision until TCP receives all duplicate acknowledgments for a given window of data (packets in flight). This will give TCP a clear image about the number of drops from this window. The congestion window size is then reduced only by number of dropped packets. Also, we propose a safety mechanism to prevent this algorithm from causing congestion to the network by using an extra congestion window threshold (*tthresh*) in order to save the safe area where there are no drops of any kind. The second algorithm is a new retransmission action to deal with multiple drops from the same window. This multiple drops action (MDA) will prevent TCP from falling into consecutive timeout events by resending all dropped packets from the same window. A third algorithm is used to calculate a new back-off policy for TCP retransmission timeout based on the network's available bandwidth. This new retransmission timeout action (RTA) helps relating the length of the timeout event with current network conditions, especially with heavy transmission error rates.

The three algorithms have been combined and incorporated into a delay based error discriminator. The improvement of the new algorithm is measured along with the impact on the network in terms of congestion drop rate, end-to-end delay, average queue size and fairness of sharing the bottleneck bandwidth. The results show that the proposed error discriminator along with the new actions toward transmission errors has increased the performance of TCP. At the same time it has reduced the load on the network compared to existing error discriminators. Also, the proposed error discriminator has managed to deliver excellent fairness values for sharing the bottleneck bandwidth.

Finally improvements to the basic error discriminator have been proposed by using the multiple drops action (MDA) for both transmission and congestion errors. The results showed improvements in the performance as well as decreases in the

congestion loss rates when compared to a similar error discriminator.

بسم الله الرحمن الرحيم

In the name of God, the most gracious, the most merciful

Acknowledgments

I wish to express my gratitude and sincere thanks to many people who have made this work possible and accessible for me.

At the top of the list I owe great thanks to my family, especially my parents Abdullah and Roqaya, my wife Amani and my children Joud, Leen and Abdullah and all my brothers. I believe that without their support and encouragement, I would never have been able to complete my studies away from home. My parents were always behind any achievement I ever done in my life. My lovely wife is the person who stood by my side no matter how hard thing were. Having my children with me made my life full of joy. My brothers were always trying to help me when I go back home to visit and they make every effort to make my life easier.

I also owe grateful thanks to my supervisors, Mr John Mellor and Dr.Rod Fretwell, for their valuable guidance, advice and support. They provided me with endless support and encouragement, which was extremely important for me. No matter how busy they have been, they have always found time for me. I do not think that I could have had a better supervisors during my studies.

Also I am pleased to express my sincere thanks to all my friends and colleagues back home and here in the UK for their support and encouragement. Our frequent meetings, SMS messages and phone calls made a long journey shorter and more enjoyable.

Last but not least, I would like to thank the ministry of higher education and King Saud University in Saudi Arabia for their full financial support during my studies in the UK.

Publications

1. M. Alnuem, J. Mellor and R. Fretwell "Performance evaluation of ECN-based TCP error discriminator over wireless networks", Proceedings of the Fourth International Conference on Performance Modelling and Evaluation Of Heterogeneous Networks (HET-NETs06), Ilkley, UK, September 2006.

2. M. Alnuem, J. Mellor and R. Fretwell, "Explicit Congestion Notification for Error Discrimination - A practical approach to Improve TCP performance over wireless networks", Proceedings of the 6th Wireless Telecommunications Symposium (WTS2007), Pomona, CA, USA, April 2007, pp 1-9, IEEE Communication Society, ISBN: 978-1-4244-0696-8.

3. M. Alnuem, J. Mellor and R. Fretwell, "TCP Performance over Lossy Links", Proceedings of the 8th Informatics Workshop for Research Students, University of Bradford, 2007.

4. M. Alnuem, J. Mellor and R. Fretwell, "TCP congestion window update for transmission errors", Proceedings of the 9th Informatics Workshop for Research Students, University of Bradford, 2008.

5. M. Alnuem, J. Mellor and R. Fretwell, "TCP Multiple drop action for transmission errors", The 9th Annual Postgraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting, PGNet2008, Liverpool John Moores University, 2008.

6. M. Alnuem, J. Mellor and R. Fretwell, "New algorithm to control TCP behaviour over lossy links", The International Conference on Advanced Computer Control, ICACC 2009, Singapore, January 2009, IEEE Computer Society, ISBN:978-1-4244-3330-8.

Abbreviations

ACK : New TCP packet acknowledgment.

AIMD : Additive increase multiplicative decrease.

AQM : Active queuing mechanism.

ARED : Adaptive random early detection.

ARPANET : Advanced Research Project Agency.

Cedge : Congestion edge.

Cwnd : TCP congestion window.

CWA : Congestion window action.

ECR : Effective cut rate.

EWA : Exponential weighted average.

DACK : Duplicate TCP packet acknowledgment.

FTP : File transfer protocol.

HTTP : Hypertext transfer protocol.

IP : Internet protocol.

LRD : Long range dependence.

MDA : Multiple drops action.

MMPP : Markovian Modulated Poisson Process.

OSI : Open System Interconnection.

PACK : Partial acknowledgment.

RED : Random Early Detection.

RTA : Retransmission timeout action.

RTT : Round trip time.

RTO : Retransmission timeout.

Ssthresh: Slow start threshold.

TCP : Transmission Control Protocol.

UDP : User Datagram Protocol.

Contents

Acknowledgments **iv**

Publications **vi**

Abbreviations **viii**

Abstract **ix**

List of Figures **xvii**

List of Tables **xviii**

1 Introduction **1**
1.1 Introduction 1
1.2 Motivation 3
1.3 Aims and Objectives 4
1.4 Original Contributions 6
1.5 Thesis Outline 8

2 Transmission Control Protocol and TCP Congestion Control **10**
2.1 Introduction 10
2.2 Transmission Control Protocol 10
2.3 Sliding Window (Congestion Window) 13
2.4 TCP Congestion Control 15
2.4.1 Slow Start 16
2.4.2 Congestion Avoidance 17
2.4.3 Drop Detection 20
2.4.4 Timeout Mechanism 21
2.4.5 Retransmission Mechanism 23
2.5 TCP Reaction to Transmission Drops 26
2.5.1 The Problem 28
2.6 Summary 30

3 Enhancing TCP Performance: Related Work **31**
3.1 Introduction . . . 31
3.2 Solutions for TCP Performance Over Lossy Networks . . . 31
3.3 End-to-End Solutions . . . 33
3.4 Congestion Drops . . . 34
3.4.1 Retransmission Timeout (RTO) . . . 34
3.4.2 Fast Retransmission (TCP-Reno) . . . 35
3.4.3 Fast Retransmission Phase (TCP-NewReno) . . . 36
3.4.4 Selective Acknowledgment (TCP-Sack) . . . 38
3.5 Congestion and Transmission Drops . . . 39
3.5.1 Congestion Predictors . . . 39
3.5.1.1 TCP-Vegas . . . 40
3.5.1.2 TCP-Westwood . . . 43
3.5.2 Error Discriminators . . . 44
3.5.2.1 Network Dependent Error Discriminators . . . 44
3.5.2.2 TCP-Casablanca . . . 45
3.5.2.3 TCP-Ifrane . . . 46
3.5.2.4 Explicit Congestion Notifications . . . 47
3.5.2.5 TCP-Jersey . . . 48
3.5.2.6 Network Independent Error Discriminators . . . 49
3.5.2.7 Error Discriminator Based on Vegas Congestion Predictor . . . 51
3.5.2.8 Error Discriminator Based on CARD Delay-Based Congestion Predictor . . . 52
3.5.2.9 Error Discriminator Based on Tri-s Throughput-Based Congestion Predictor . . . 53
3.5.2.10 TCP-Veno . . . 54
3.5.2.11 Receiver Based Error Discriminators . . . 55
3.5.2.12 Fast Recovery Plus . . . 57
3.5.2.13 Spike Error Discriminator . . . 58
3.6 Summary: Action toward transmission errors . . . 59

4 Improving TCP Error Discriminators Reaction to Transmission Drops **62**
4.1 Introduction . . . 62
4.2 New TCP Reactions to Transmission Errors . . . 64
4.2.1 Congestion Window Action . . . 64
4.2.1.1 Motivation . . . 64
4.2.1.2 The Algorithm . . . 67
4.2.1.3 CWA Performance . . . 73
4.2.2 Recovering From Multiple Drops . . . 76
4.2.2.1 Motivation . . . 76
4.2.2.2 The Algorithm . . . 78

4.2.2.3 Nonconsecutive Drops . . . 79
4.2.2.4 MDA Performance . . . 83
4.2.3 Improving RTO back-off for Transmission Drops . . . 85
4.2.3.1 Motivation . . . 85
4.2.3.2 The Algorithm . . . 86
4.2.3.3 RTA Performance . . . 91
4.3 Transmission Window Action (TWA) . . . 94
4.4 Summary . . . 95

5 Simulation Model and Performance Evaluation of TWA 98
5.1 Introduction . . . 98
5.2 Simulation Model . . . 98
5.2.1 Rationale Behind Simulation: Creating a Controlled Environment . . . 99
5.2.2 Topology and Simulation Settings . . . 100
5.2.3 Round Trip Time - RTT . . . 105
5.2.4 Performance Metrics . . . 106
5.2.5 Simulation Validation . . . 108
5.2.6 Confidence Intervals and Relative Precision . . . 111
5.3 Performance Evaluation of TWA . . . 113
5.4 Summary . . . 119

6 End-to-end TCP Sender Error Discriminator with New Transmission Drops Action 120
6.1 Introduction . . . 120
6.2 TCP-RTT . . . 121
6.2.1 System Design . . . 123
6.3 Performance Results . . . 129
6.3.1 Experiment Settings and Assumptions . . . 129
6.3.1.1 Performance Metrics . . . 131
6.3.2 Results With No Congestion . . . 131
6.3.2.1 Goodput . . . 132
6.3.2.2 Congestion Window . . . 134
6.3.2.3 Number of RTO Events . . . 135
6.3.2.4 RTO Length . . . 137
6.3.2.5 The Effect of Packet Round Trip Time . . . 138
6.3.3 Results With Congestion . . . 138
6.3.3.1 Goodput . . . 139
6.3.3.2 Congestion Window and Retransmission Timeout . . . 140
6.3.3.3 The Effect of The Error Model . . . 143
6.4 Impact of TCP-RTT on the Network . . . 147
6.4.1 Impact on Network Congestion Loss Rate . . . 148
6.4.1.1 Single Flow Case: . . . 149

6.4.1.2 Multiple Flows Case: . . . 150
6.4.2 End-to-end Delay . . . 151
6.4.3 TCP-RTT Fairness . . . 153
6.5 Modeling TCP-RTT Behaviour . . . 157
6.5.1 A More General Model . . . 160
6.6 Summary . . . 162

7 Allowing Multiple Drops Action for Congestion Losses 166
7.1 Introduction . . . 166
7.2 Results . . . 167
7.3 Impact of TCP-RTTM on the Network . . . 169
7.3.1 Impact on Network Congestion Loss Rate . . . 169
7.3.1.1 Single Flow Case . . . 169
7.3.1.2 Multiple Flows Case . . . 170
7.3.2 End-to-End Delay . . . 172
7.3.3 TCP-RTTM Fairness . . . 172
7.3.4 Cedge Configuration - Varying *midalpha* . . . 174
7.4 Different RTT and Bandwidth Values . . . 177
7.4.1 Different RTT Values . . . 178
7.4.2 Different Bandwidth Values . . . 178
7.5 The Effect of Error Burst Size . . . 179
7.6 System Limitations . . . 183
7.7 Summary . . . 184

8 Conclusion 186
8.1 Conclusions . . . 186
8.2 Proposals for Extensions and Future Work . . . 189

References 192

Appendices

A Standard TCP Reaction to Drops 205

B MDA - The Case of Non Sequence Errors 207

C Traffic Generation 211

D TCP-RTT Source Code 215

List of Figures

2.1 TCP/IP and OSI Reference models 11

2.2 Slow start and Congestion avoidance in the presence of errors 18

2.3 Packet Drop 20

2.4 TCP with multiple drops 27

2.5 TCP behaviour in absence of errors 29

2.6 TCP with transmission errors 29

4.1 TCP duplicate acknowledgment action 67

4.2 CWA duplicate acknowledgment action 68

4.3 CWA Pseudo code 70

4.4 CWA+tthresh Pseudo code 72

4.5 Simple Network Topology 73

4.6 TCP vs. CWA. semi-log scale congestion window size (packets) . . . 75

4.7 TCP and burst of drops 80

4.8 Proposed multi drop retransmission 81

4.9 MDA Pseudo code 82

4.10 TCP vs. MDA. No.RTO 83

4.11 Semi-log scale TCP vs. MDA. No.RTO 84

4.12 Available bandwidth estimation 88

4.13 RTO back-off algorithm 89
4.14 RTO length 92
4.15 Available bandwidth and backoff level 93
4.16 Error discriminator general functionality 94
4.17 Error discriminator after adding TWA 95

5.1 Network topology 101
5.2 Min, Max and Average RTT values 106
5.3 Analytical and Experimental models comparison 109
5.4 Experimental results with 95% confidence intervals 112
5.5 TCP and TWA semi-log scale normalized goodput 115
5.6 TCP, TWA and Sack semi-log scale normalized goodput 115
5.7 TCP, TWA and NewReno semi-log scale normalized goodput 116
5.8 TCP and TWA average congestion window size 117
5.9 Number of retransmission timeout events 118
5.10 Total idle time for TCP and TWA 118

6.1 Spike states [1] 122
6.2 RTT states 125
6.3 TCP vs. TCP-RTT normalized goodput 132
6.4 TCP vs. TCP-RTT semi-log scale normalized goodput 133
6.5 TCP vs. TCP-RTT semi-log scale congestion window size 134
6.6 Congestion window evolution 135
6.7 TCP vs. TCP-RTT number of RTO events 136
6.8 TCP vs. TCP-RTT RTO length 137
6.9 TCP-RTT under different RTT values 139
6.10 TCP vs. TCP-RTT semi-log normalized fair share goodput 140

6.11 semi-log congestion window size 141
6.12 Number of RTO events 142
6.13 Timeout length 142
6.14 TCP vs. TCP-RTT semi-log normalized fair share goodput - Uniform transmission errors 144
6.15 TCP performance under uniform and bursty transmission errors . . . 145
6.16 TCP-RTT performance under uniform and bursty transmission errors 145
6.17 TCP congestion window size growth under uniform and bursty errors 146
6.18 Congestion window variability 148
6.19 Network congestion loss rate (TCP, TCP-RTT and Spike) 149
6.20 Network congestion loss rate for multiple flows 150
6.21 end-to-end delay 152
6.22 Average queue size 152
6.23 Fairness index - 180Mb bottleneck 154
6.24 Fairness index - 36Mb bottleneck 155
6.25 Congestion window increase/decrease behaviour 158
6.26 Goodput - increasing RTT - 1% error rate - TCP-RTT and Analytical model 163
6.27 Goodput - increasing RTT - 5% error rate - TCP-RTT and Analytical model 163
6.28 Goodput - increasing RTT - 10% error rate - TCP-RTT and Analytical model 163

7.1 TCP vs. TCP-RTTM normalized fair share goodput 167
7.2 TCP vs. TCP-RTT vs. TCP-RTTM normalized fair share goodput . 168
7.3 Network congestion loss rate (TCP, TCP-RTT and SpikeNR) 170

7.4 Network congestion loss rate for multiple flows 171
7.5 Average queue size 172
7.6 end-to-end delay 173
7.7 Fairness index - TCP-RTTM and SpikeNR 173
7.8 Improvement in goodput and impact on the network-$midalpha = 0.15$ 176
7.9 Improvement in goodput and impact on the network-$midalpha = 0.25$ 176
7.10 Improvement in goodput and impact on the network-$midalpha = 0.50$ 176
7.11 Congestion drop rate 177
7.12 Performance with different RTT values - semi-log scale normalized fair share goodput 178
7.13 Performance with different bandwidths 179
7.14 Performance with different transmission error burst size 180
7.15 Performance with different transmission error burst size 2 182

A.1 Standard TCP reaction to drops 206

B.1 Multiple drops action + error rate estimation flow chart 209

C.1 Simple Topology 212

List of Tables

5.1 Confidence intervals and relative precision for 95% confidence level . . 113

6.1 Spike Throughput improvement 156

C.1 MMPP traffic generator Hurst parameter for different intervals 212

Chapter 1

Introduction

1.1 Introduction

The Internet today is a large set of interconnected heterogeneous networks [2]. Since it was first introduced in the late sixties as ARPANET [2] (Advanced Research Project Agency) many new network technologies and communication environments have been integrated into the Internet infrastructure. One of the main improvements is the introduction of wireless and mobile networks which have been connected to existing wired networks adding to the heterogeneity of the Internet. The introduction of wireless links has brought with it many new challenges among them the high rate of bit errors in wireless links compared to wired links [3].

TCP (Transmission Control Protocol [4]) is one of the most used transport protocols in the Internet [5]. Many widely used applications use TCP for sending data like File Transfer Protocol [6] (FTP), Telnet [7] and Web-HTTP [8] connections.

TCP is a connection oriented end-to-end transmission protocol. It lies in the transport layer of the OSI [9] reference model (Open System Interconnection reference model) and it is usually used to provide a reliable way of delivering data by

using acknowledgments for sent packets.

An important mechanism in TCP is the congestion control mechanism [10, 11]. It controls the TCP sending rate and provides end-to-end congestion avoidance and control. However, TCP congestion control was designed under the assumption that congestion is the main cause of packet drops and that drops due to link errors happen rarely [10]. This assumption was acceptable when the Internet was first introduced in the sixties because of the small scale of computer networks and when most networks used wired links [2] which usually have small error rates. In fact Jacobson [10] in 1988 indicated that, in most networks, drops for reasons other than congestion are far less than 1%.

However, today's networks, specially the Internet, are of big scale and use many new unreliable channels which may suffer from errors unrelated to congestion. For example wireless, mobile and satellite networks may drop packet for reasons other than congestion, like bad weather conditions and natural or artificial obstructions [3].

TCP was found to perform poorly over heterogeneous networks when transmission (non-congestion) errors exist [5, 12–19]. This is due to the fact that TCP is unable to distinguish between congestion errors and transmission errors caused by link failure and hence TCP reduces its sending rate for all errors, implicitly assuming congestion exists in the connection path.

In our work we identify mainly three areas where TCP interacts with errors: the congestion window mechanism, retransmission mechanism and retransmission timeout mechanism. We study the effect of transmission errors on these mechanisms and propose new algorithms (called transmission window actions) which can make TCP cope with transmission errors.

In practice we cannot use the transmission window actions directly in TCP since congestion and transmission drops may coexist. One solution is to discriminate

between errors and deal with each error type differently. An error discriminator will be added to TCP to replace the congestion control mechanism and to decide on when and how to cut the sending rate. For that we use an end-to-end error discriminator based on the packet round trip delay. The new transmission window actions are added to the error discriminator and the impact on the network and on the other flows are measured.

The resulted techniques are fully end-to-end techniques that require no changes to the network or to the receivers (clients) and only the sender (server) TCP implementation needs to be changed. This will reduce significantly the scale of changes required to adopt the new techniques in today's Internet .

In this work we will refer to packet drops caused by congestion as *congestion drops* and drops caused by link errors as *transmission/non-congestion drops* or *wireless drops* since wireless links are important source for transmission drops in today's networks.

Also when we mention TCP during this thesis we mean Reno [11, 20] version unless stated otherwise. The Reno version has been chosen to be base of our work because it is considered the commonly used TCP version in the Internet [21,22]. Also TCP-Reno implements the standard TCP requirements as presented in [4,20,23,24].

1.2 Motivation

The aim of this work is to participate in the efforts in progress to improve the performance of TCP protocol over heterogeneous networks where transmission drops may occur frequently .

Many studies like [5, 12–19] have shown that the performance of TCP degrades noticeably when transmission errors occur because TCP cannot distinguish between

congestion drops and transmission drops. This confusion has caused TCP to think that all errors are caused by congestion and hence continuously slowdown and reduces its performance in order to reduce the assumed congestion in the connection path. During this work we investigate the behaviour of TCP over unreliable links and we advise solutions that can resolve the problem.

The motivation for this work is that it deals with a protocol that is considered as the standard transport protocol in the Internet [5] since it is used to deliver huge amount of the Internet content. For example, popular applications like FTP and Internet browsers (HTTP) use TCP as the main transport carrier.

Moreover, with the increase of using mobile networks and the need to access Internet content, which usually is delivered over TCP using mobile devices, the issues related to TCP became of interest to the mobile and wireless networks research community specially TCP performance over wireless networks.

Although many TCP performance issues have been researched and solutions have been proposed, the introduction of new network channels like satellite and wireless links and the increased use of TCP to carry data over such channels with higher error rates compared with wired links, all of this has increased the need to study and solve the potential problem when TCP works in such conditions. Because of that TCP became under heavy revision; see for example excellent surveys in [22,25].

1.3 Aims and Objectives

Based on the motivations mentioned earlier the main aim of this work is to improve TCP performance over heterogeneous networks. This main aim can be divided into following smaller aims:

- To develop a new action toward transmission drops which can improve TCP

performance and at the same time prevent causing more congestion because of the error discriminator mismatch between error types.

- To propose an end-to-end solution which requires minimum changes to TCP and no changes to the network by using a delay based error discrimination which will allow TCP to implement different actions for errors occurring during network congestion and for transmission errors.

In order to achieve these aims we have to achieve the following objectives:

- To conduct a comprehensive literature review in order to understand how TCP works, which mechanisms are affected by packet drops and what different solutions have been proposed to overcome the problem.

- To investigate the end-to-end solutions specially error discriminators and to understand how they work and how they react to transmission errors.

- To develop an end-to-end reaction toward transmission errors which increases TCP performance and at the same time apply gentle action on the network.

- To develop a simulation environment that will be used to simulate TCP behaviour when transmission errors exist and to use this simulator to evaluate the impact of the proposed transmission drops algorithms on TCP performance.

- To develop an end-to-end error discriminator based on packet delay information to discriminate between congestion and transmission errors.

- To add the proposed transmission drops algorithms to the error discriminator and to evaluate the performance of the new technique and its impact on the network and other TCP connections.

1.4 Original Contributions

Primary contributions:

- Proposing a three stage TCP action for transmission drops over lossy networks (congestion window action, multiple drops retransmission action and retransmission timeout action):
  - Congestion Window Action (CWA): We present a new TCP congestion window cut algorithm for transmission errors. TCP cuts the congestion window to half of the original size after receiving three duplicate acknowledgments which makes TCP performance decrease unnecessarily if the drops were not caused by congestion. Instead of cutting the congestion window after receiving three duplicate acknowledgments we delay the cut decision until TCP receives all the duplicate acknowledgments for a given window of data (packets in flight). This will give TCP a clear image of the number of drops from this window. Then congestion window size is reduced only by the number of dropped packets. The CWA can be used by TCP error discriminators to create more gentle reaction toward transmission errors. Also we propose a safety mechanism to prevent this algorithm from causing congestion to the network by using extra congestion window threshold *tthresh* to save the safe area where there are no drops of any kind.
  - Multiple Drops retransmission Action (MDA): TCP cannot deal with multiple drops from the same window of data and when a burst of drops occurs, either because of congestion or transmission errors, TCP reduces its sending rate significantly and waits for retransmission timeout to recover the lost packets. However, due to the bursty nature of transmission

drops from sources like wireless networks [3,26] which can lead to multiple packet drops from the same window, in our work we present a multiple drops action which can retransmit multiple drops from the same window of data.

- Retransmission Timeout Action (RTA): TCP uses an exponential back-off policy in response to multiple drops of same packet in case of heavy error rates which does not consider the actual network conditions (if the network is congested or not). In our work we have developed a back-off policy that makes use of the available bandwidth to compute a new back-off level based on the link available capacity.

All these actions are combined to form a complete reaction to transmission errors we call them transmission window actions (TWA).

- Developing and testing an end-to-end error discriminator based on packet round trip time measurement and proper transmission window actions. This error discriminator is sender based so it requires changes only to one side (server side) of the connection and it is entirely end-to-end so it does not require any change to the network which make it easily deployed in real networks.

- Improving the performance of the proposed error discriminator by using MDA for both transmission and congestion drops.

- A simulation environment has been implemented to assess the impact of the new proposal on the performance of TCP by comparing TCP-Reno with a TCP version modified to include the proposed error discriminator along with the new transmission window actions. Also the impact of the proposed scheme

on the network and other TCP flows has bean analyzed and compared with the impact of exiting error discriminators.

- We proposed and tested an analytical model to approximate TCP performance when the congestion window cut factor is based on the error rate instead of a fixed factor.

1.5 Thesis Outline

The remaining parts of this thesis are organized as follows:

In Chapter 2 we start by explaining TCP and how it works. We give a description of TCP congestion control mechanism and its main components, namely slow start, congestion avoidance, retransmission mechanism and timeout mechanism. Then TCP reaction to transmission drops is investigated.

Chapter 3 describes different solutions to the problem presented in Chapter 2. The solutions are categorized into three categories: end-to-end, split connection and link layer solutions. An extended explanation of the end-to-end solutions is given.

In Chapter 4 we present three proposals to change TCP congestion control mechanism reaction to transmission drops. First a proposal to change the way TCP cuts the congestion window after a transmission error has occurred (called the congestion window action - CWA). Second a new mechanism to retransmit multiple drops from the same window of data (called multiple drops action - MDA). Finally we describe a new mechanism to calculate TCP retransmission timeout back-off level based on the available bandwidth (called retransmission timeout action - RTA). Each mechanism is evaluated alone to show how it can improve TCP performance in presence of transmission errors. The evaluation of these mechanisms combined is left to the next chapter.

Chapter 5 explains in detail the simulation model we used, the topology, the experiment settings and parameters, how we generate the errors, what kind of traffic we used in our experiments, how we validate our simulation and the use of confidence intervals and relative precision to decide how many runs of each experiment we need. After that we add the three algorithms presented in Chapter 4 CWA, MDA and the RTA to TCP and call them the transmission window actions (TWA) and we use the simulation model to evaluate them.

In Chapter 6 we present the design and evaluation of an end-to-end error discriminator that uses the packet delay to discriminate between error types and uses the proposed transmission window actions (TWA) in case of transmission errors. The impact of the new actions on the error discriminator performance and on the network is measured and evaluated. Also in this chapter we present an analytical model to approximate the performance of the error discriminator with the new transmission window actions.

Chapter 7 describes improvements to the basic algorithm which lead to new results. The main improvement is the use of the proposed multiple drops action (MDA) in case of congestion as well as transmission errors. The resulting error discriminator is called TCP-RTTM and is tested under different bandwidth, delay values and error burst sizes.

In Chapter 8 we conclude the thesis by explaining the main work we did and the possible areas for future work.

Chapter 2

Transmission Control Protocol and TCP Congestion Control

2.1 Introduction

In this chapter we will discuss the problem TCP performance faces when operating over heterogeneous networks where unreliable links can exist. We start by an overview of TCP [4] and how it works. Then we talk about TCP congestion control mechanism [10] and its relation to the TCP performance degradation over lossy links.

2.2 Transmission Control Protocol

TCP lies at the heart of two of the most widely used network architectures, the OSI [9] and the TCP/IP [27]reference models. It is located in the Transport layer in both models see figure 2.1 . The main aim of the transport layer is to give an end-to-end data transport service to the upper layers and also to act in a way so

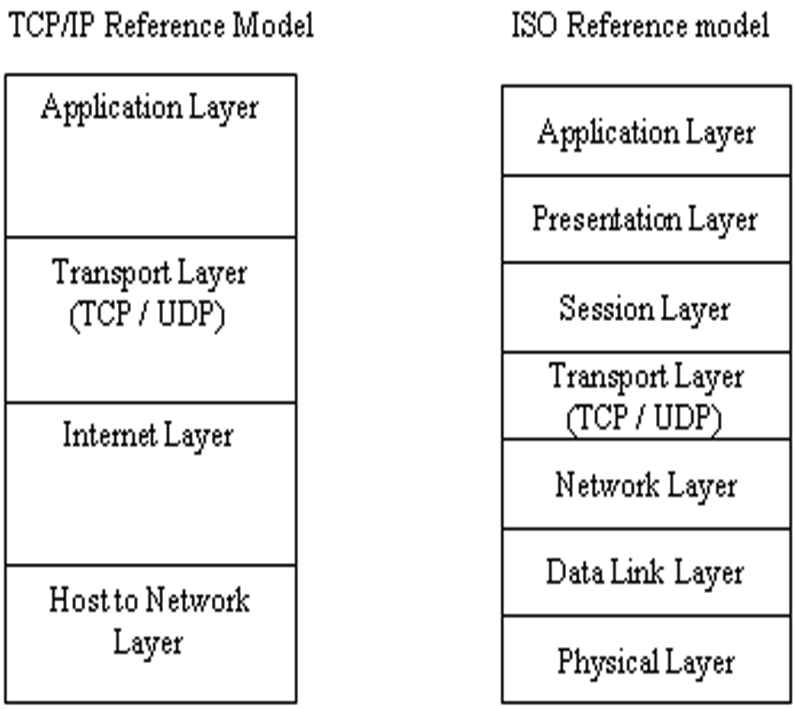

Figure 2.1: TCP/IP and OSI Reference models. Source [2]

that the upper layers can function normally even if there are changes in lower layers like hardware upgrade for example [2].

Two types of services are provided by the transport layer. One is based on the simple idea of sending messages without any guarantee of whether it will arrive at the destination or not and without any ordering of the sent messages [2]. This type of service is useful if loosing some messages will not affect the validity of the transmission like the case of video and audio streaming where some packets will not affect the service, provided that the number of lost packets is in an acceptable range. Also in this type of service (i.e. video and audio), retransmission of missing data is not acceptable since it is a real time service where having most of the packets arrive on time is much more important than retransmitting some packets in the middle.

The transport layer implements these types of services using the User Datagram

Protocol, widely known as UDP [28]. UDP does not provide guarantees in terms of message delivery and correctness and it does not provide ordering. Moreover, UDP is suitable for applications that provide their own sequencing and flow control mechanisms [2].

The second type of service provided by the transport layer is a guaranteed service. Here the correctness and the ordering of delivered data are guaranteed. The implementation of this service is done by TCP [2].

Transmission Control Protocol [4], widely known as TCP, is a transport layer protocol to transport data through the network in a reliable and error free mode. It delivers messages from one end to another and makes sure that all packets have been delivered uncorrupted and in order. TCP is a connection oriented protocol, so, before TCP starts transmitting data between two ends, it establishes an agreement of how the connection should be operated. This is done by exchanging control packets before the real data is transmitted [29] [2]. An analogy to this is when we use the analogue phone and we need to dial the number before we can start talking.

Because of the nature of the IP routing used in the Internet, not all packets will follow the same path from source to destination. This fact may cause packets to be reordered so some packets may be received out of order. TCP should be prepared to accept out of order packets and to expect delays in some packets. Normally TCP will assume a time limit for each packet to reach a destination and the packet will be considered lost after this time limit [29].

Other important functions of TCP are flow control and congestion control. In flow control, TCP makes sure that there is a coordination between the sender and the receiver so the sender will not send more than the capacity of the receiver buffer. TCP should be able to coordinate the source and destination, so the source will not overflow the destination buffer. To achieve this the receiver tells the sender its

maximum buffer size during the connection establishment phase [29].

In congestion control, TCP detects congestion by using packet drops as a sign of congestion in the network and then it tries to resolve congestion by reducing the sender's transmission rate to prevent creating permanent congestion. TCP should have some awareness of the link status in order to avoid injecting the link with more data than its capacity. Congestion control algorithms [10] [11] are used for this purpose [29].

Moreover, in order for TCP to operate in an environment like the Internet, TCP should be able to handle connections with variable round trip times (Round trip time, RTT, is the time from a packet is sent until an acknowledgment is received) [29]. Since, TCP is supposed to be able to connect any two hosts in the Internet, no matter how far apart they are, it should be able to accept different round trip times for different connections and even different round trip times for the same connection. So it should be able to adjust its timeout mechanism to adopt with the variation in round trip times [29]. Later in this chapter we will show how TCP is able to handle variations in RTT.

These are some issues about TCP functionality. Next we explore in more detail some important mechanisms in TCP.

2.3 Sliding Window (Congestion Window)

A sliding window algorithm is at the heart of reliability and congestion avoidance services provided by TCP. An understanding of the sliding window algorithm used in TCP will help to understand the congestion avoidance algorithm and how TCP provides reliability and ordering.

In the TCP sliding window algorithm the sender maintains a window called

Congestion Window or in short *cwnd* and gives a sequence number for each packet in this window. *cwnd* determines how many packets can be sent before receiving any acknowledgment. This way the *cwnd* controls TCP sending rate (the bigger the *cwnd* the higher the sending rate and vice versa)

The sender keeps track of the last received acknowledgment and the last sent packet. When the sender receives an acknowledgment it sends a new packet and updates the last sent and last received variables . Also, the sender attaches a timer with each sent packet. If the timer expires before receiving acknowledgment, the packet can be resent [29].

On the other side, the receiver handles a window (a buffer) and three variables: the window size, the sequence number of last packet received and the largest sequence number that can be accepted which is calculated by adding the buffer size to the sequence number of last packet received [29]. When the receiver receives a packet the packet sequence number should be bigger than the last received packet sequence number and less than or equal to the largest sequence number that can be accepted. If not then the packet is discarded because it is outside the receiver window [29].

On the other hand, if the new packet lies inside the receiver window, the receiver accepts the new packet and if the new packet is the next expected packet, then it updates the last packet received variable and sends an acknowledgment for this packet. However, sometimes the packet may arrive out of order; in this case the receiver does not acknowledge the new packet. Instead it sends an acknowledgment for the last in order received packet. This is called a duplicate acknowledgment.

For example, if the last packet received in order is 3 and we have received 5 and 6 before 4, then the receiver will send a duplicate acknowledgment carrying the sequence number 3 to tell the sender that it is still waiting for packet 4. When

packet 4 finally reaches the receiver, the receiver will issue an acknowledgment for packet number 6 which will acknowledge all previous packets. On the other hand, if packet 4 never reaches the receiver (i.e. lost in the path), the sender timer will expire and it will send the packet again [29].

The main feature of this algorithm is that it controls the number of packets in the network at any time by making the maximum number of packets the sender can put in the network at one time equal to the window size and that the sender can not deposit new packets in the network until it receives acknowledgments that the old ones have been received (have left the network).

2.4 TCP Congestion Control

TCP applies five congestion control algorithms namely slow start, congestion avoidance [10], retransmission mechanism (fast retransmission [11] and fast recovery [11]) and timeout mechanism [4]. The slow start and congestion avoidance algorithm, were added to TCP in [23] and then a full description of the algorithms were first documented for the Internet in [20]. After that, authors in [24, 30] specified all the algorithms with more detail and they discussed more issues and concerns about situations where actions that should be taken by TCP after restarting ideal connections and the requirements that the TCP receiver should guarantee in acknowledgments (ACKs). Also they raised some security issues like the ability to attack a system that runs TCP by forging duplicate acknowledgments or causing packets to be lost [24].

Throughout the thesis we will talk about each algorithm as required while following I will give a brief explanation of each algorithm.

2.4.1 Slow Start

When TCP sends data it limits its sending rate to the minimum of its congestion window size *cwnd* and the receiver buffer. However, it is found that if TCP starts the connection by sending the whole window at once this may cause unnecessary transient congestion and hence packet drops which will affect TCP performance badly [10]. So when TCP starts sending for the first time or when it restarts a broken connection it should send packets gradually.

The idea of the slow start mechanism is that instead of sending the whole window at once the TCP starts by sending one packet and then increases the window size exponentially until it reaches the maximum available window size (minimum of sender window and receiver buffer). Slow start increases the congestion window exponentially by doubling the congestion window size with each round trip time. This is done by increasing the congestion window size by one packet for each new acknowledgment [10].

Since slow start defines the initial behaviour of a TCP connections, one of its aims is to discover the link capacity gradually by continually increasing the sending rate until the link capacity is reached when drops occur. At this point TCP moves to the next mechanism, congestion avoidance, which will handle the rest of the connection life time. For this reason TCP defines a threshold called slow start threshold (*ssthresh*) which defines the border between slow start and congestion avoidance mode.

So TCP increases the *cwnd* exponentially until it reaches the *ssthresh* where it switches to congestion avoidance mode as we will describe next.

2.4.2 Congestion Avoidance

The congestion avoidance phase is the most important phase in the TCP life time since it represents the equilibrium state of the connection life time. The equilibrium state is defined as the state when a TCP sender puts a packet into the network as another packet from the same connection leaves the network at the receiver [31]. Moreover, most of the effect of transmission errors will take place in this phase since it is the longest phase.

When TCP starts sending data the congestion window starts to grow exponentially during the slow start phase until a packet is dropped which indicates two things: first that the link capacity has been reached, second that the congestion avoidance phase has started. At this stage the TCP congestion avoidance mechanism takes control. The TCP congestion avoidance mechanism has mainly two different and important jobs:

- One is to decide on the increase/decrease of the congestion window size (the direction of the change).
- And another is to decide the value of the increase/decrease in the congestion window (the amount of the change).

In order to decide the direction of the change TCP uses packet drops as a signal that the congestion window should be decreased (downward direction) and the absence of the drops, and hence receiving of new acknowledgment, as a signal that the congestion window needs to be increased (upward direction).

TCP decides the amount of change (decrease/increase) based on the direction of that change by using AIMD mechanism (Additive increase Multiplicative decrease) [10]. TCP increases the congestion window linearly (Additive increase) by $\frac{1}{window\_size}$ with each new acknowledgment received [24]. This is equivalent to increasing the

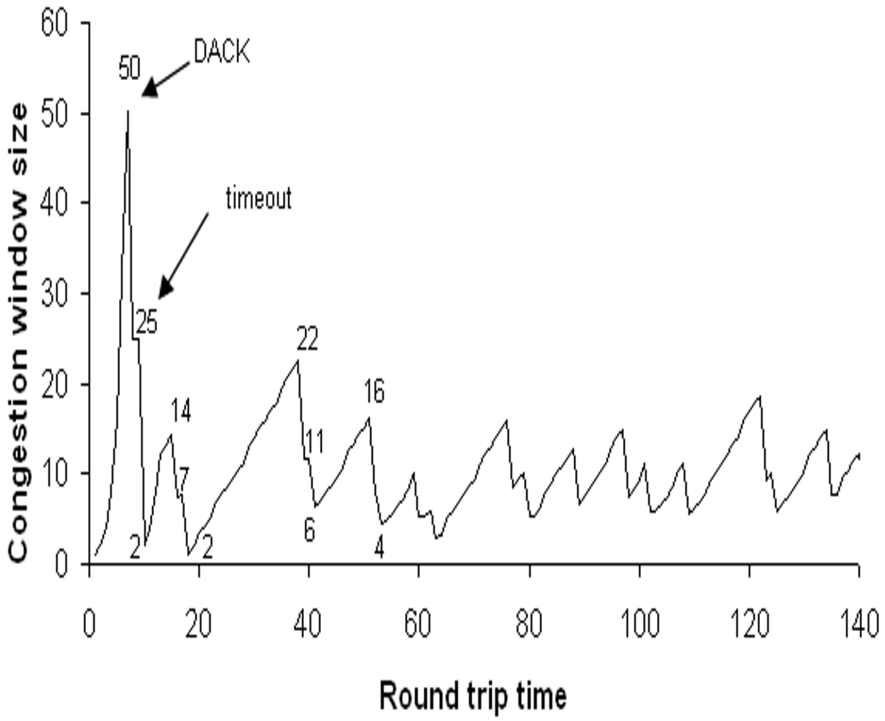

Figure 2.2: Slow start and Congestion avoidance in presence of errors

congestion window by one packet each round trip time. In other words, TCP will wait until the whole window is sent safely (i.e. without any drop) and then it increases the window size by one packet.

However, if a drop occurred TCP takes this as an indication that the new window exceeded the link capacity or that a new load has been introduced to the network (for example new users start downloading FTP files). In this case TCP will reduce congestion window size to half of its size before the drop occurred (multiplicative decrease by factor if 0.5).

Figure 2.2 shows a typical TCP congestion widow behaviour (slow start and congestion avoidance) in the presence of drops which cause timeouts and duplicate acknowledgments. The y-axis represents the congestion window size in packets at each round trip time. The numbers on the top of each curve are the values of the

congestion window size before the errors occur and numbers on the bottom are the size after TCP cuts the window.

The reason for choosing 0.5 as the cut factor in the case of duplicate acknowledgments (DACKs) is presented in [10, P328]. The author argues that usually when congestion happens that means a new connection has started and it usually will use 50% of the network bandwidth and hence the available bandwidth is reduced by 50% so TCP needs to reduce the window size by 50% to allow fair sharing of the connection. This will reduce TCP performance in a multiplicative manner since with each drop TCP multiplies the current window size by 0.5.

Other authors like [32] suggest to reduce the window size by other factors like 87.5% instead of 50%. However, TCP [23] has adopted the Jacobson [10] approach by using 0.50 as the decrease factor.

The authors in [32] indicated that the aim of the AIMD mechanism is to achieve two main objectives: first to achieve fairness among competing TCP flows. Second to reach effective utilization of the bottleneck bandwidth. So using a multiplicative decrease after drops will make the connections with bigger windows (i.e. bigger share in the congestion) cut more data (for example a connection with window size of 100 packets will cut 50 packets while a connection with 10 packets window will cut 5 packets only) which will help to resolve the congestion faster and will increase fairness among competing flows. Also multiplicative decrease will make sources to slow down quickly when a congestion occurs in order to give congested routers enough space to clear the congestion [33].

On the other hand, additive increase helps TCP to explore the link capacity in a gentle way in order to avoid oscillation which can occur if aggressive multiplicative increase is used [33]. However, the additive increase will ensure linear increase in sending rate for all connections that have same round trip time.

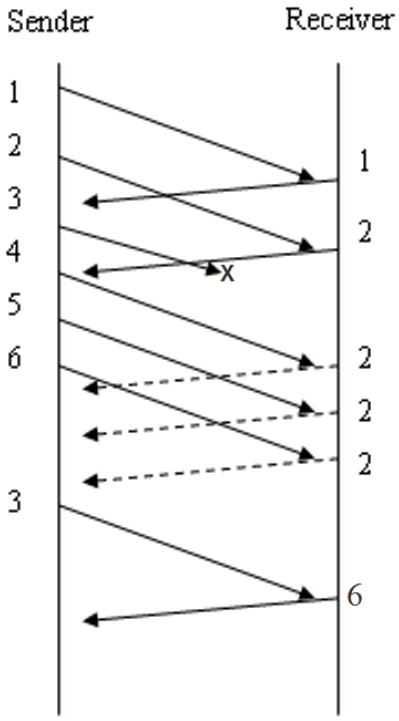

Figure 2.3: Packet Drop

It has been shown in works like [34] that the use of additive increase/multiplicative decrease behaviour in TCP will result in fair share of the network resources and less oscillation in TCP throughput [34].

2.4.3 Drop Detection

TCP uses two ways to detect drops, duplicate acknowledgment and retransmission timeout. A duplicate acknowledgment is used to identify that a packet is missing. For example in figure 2.3 a TCP sender sends six packets to a TCP receiver and packet number three has been dropped by a congested router in the connection path. When the receiver receives the first two packets it sends acknowledgments for them (packets one and two). However packet three is missing and packet four is received instead. When the receiver receives an out of order packet (i.e. packet

four) it will not acknowledge it but instead it will resend the acknowledgment of the last in order packet which is packet two. This acknowledgment is called duplicate acknowledgment (DACK).

The receiver will continue to send duplicate acknowledgments with every out of order packet (packets five and six in the figure) until it receives the required packet (packet three). The sender resends the lost packet eventually (after the third DACK as we see in figure 2.3). When the receiver receives packet three it will acknowledge all received packets by sending an acknowledgment for the last packet received in order (packet six) which indicates that it has received all packets up to packet number six correctly.

As we said the TCP sender waits until it receives three consecutive duplicate acknowledgments before deciding that there is a drop and resending the lost packet. The number three has been chosen by TCP congestion control designers [11] and has been accepted as a standard in most TCP versions. However, choosing the number of duplicate acknowledgments for deciding drops will depend heavily on the network topology and routing techniques used in the network and it is out of the scope of our thesis.

Another way TCP uses to detect drops is the timeout mechanism which will be explained next.

2.4.4 Timeout Mechanism

Retransmission timeout (RTO) for TCP was first described in RFC 793 by Postel [4, P41]. RTO is one of the first methods defined to recover from losses in TCP due to packet corruption or network congestion. It works as follows: when TCP sends a packet it sets a local timer for this packet and if the timer expires before an acknowledgment is received for this packet then TCP assumes the packet is lost.

TCP then resends the packet and sets the congestion window to the minimum allowed size (one segment).

However, calculating an accurate timeout is important because too long timeouts will reduce TCP performance because TCP will wait long periods before restarting to send again and, too short timeouts may increase the congestion in the network because the network will not have enough time to resolve the congestion.

Moreover, in order for TCP to be able to work in environments with delay variations, TCP uses RTT to calculate the retransmission timeout. Postel [4] explained how to calculate retransmission timeout based on the weighted average of the RTT readings as follows:

$$\text{AvgRTT} = \alpha \times \text{AvgRTT} + (1 - \alpha) \times \text{RTT} \tag{2.1}$$

The weighted average is used to filter sudden fluctuations in the RTT and to get the long term average. TCP specification recommends α to be between 0.8 and 0.9 [35] [29].

RTO takes the value of the average RTT providing that it is between an upper limit of 1 minute and a lower limit of 1 second as presented in equation 2.2. β is a constant value used as an estimation of RTT variation (fixed to 2) [4].

$$\text{RTO} = \min\left(1mnt, \max\left[\beta\text{AvgRTT}, 1sec\right]\right) \tag{2.2}$$

However, using constants like β and one minute/second limits was found not suitable for high speed and large networks which may suffer from delays either higher or lower than one minute/second limit [23]. So, Jacobson in [10] proposed to use a dynamic calculation of RTT variations as follows:

$$\text{RTT\_Var} = \theta \times \text{RTT\_Var} + (1 - \theta) \mid \text{RTT} - \text{AvgRTT} \mid \tag{2.3}$$

Where $\theta = 0.75$ as suggested in [10]. The one minute/second limit has been removed from RTO calculation as following:

$$\text{RTO} = \text{AvgRTT} + 4 \times \text{RTT\_Var} \tag{2.4}$$

So the RTO is now the average round trip time plus four times the average round trip time variation. The choice of 4 in equation 2.4 is based on results experienced in real networks [29].

One problem during RTT sampling is the ambiguity that occurs when there is retransmission (i.e does the acknowledgment belong to the original packet or the retransmitted one). This problem is solved by Karn's algorithm in [36] by simply discarding any RTT reading during retransmission. The changes of Jacobson [10] and Karn [36] have been added by Braden [23] as a must to be implemented in TCP.

Finally, Braden [23] discussed the implementation of RTO and he suggested two variations. One is to use a retransmission queue to store all packets that have been sent but not acknowledged yet and when a retransmission is required the packet will be ready in the queue. Another option suggests not to use a retransmission queue and instead to recreate each packet upon the resend request [23]. Clearly the first option will ease the retransmission process but it will need more buffering and processing power.

2.4.5 Retransmission Mechanism

As we said before, when TCP sends a packet it sets a timer for this packet. If no acknowledgment is received before the timer is expired (i.e. retransmission timeout

RTO occurred) the packet is sent again. However, the authors in [11] proposed a faster way to recover a lost packet. Instead of waiting for the timer to expire, if TCP receives enough duplicate acknowledgments (usually three DACKs) then we can safely assume the packet is dropped and we can resend the packet without the need to wait for the packet's timer to expire. The sender then reduces its sending rate in order to avoid increasing congestion.

So when the sender receives three duplicate acknowledgments it does the following actions [20, 24]:

- Slowdown the sending rate by reducing the congestion window size to half of its size before the drop.
- Resends the lost packet.
- Reset retransmission timer in order to allow more time for the retransmitted packet to reach the receiver before a timeout occurs.
- TCP waits for a new acknowledgment that acknowledges the resent packet and all packets in flight at the same time.
- Also TCP increases the congestion window size by one segment for every duplicate acknowledgment received until a new acknowledgment is received. The logic behind this is that since a duplicate acknowledgment tells us that one packet (even if it is out of order) has been received by TCP-receiver and hence it left the network then it is safe to put another packet in the network in its place and hence TCP increases its congestion window with every duplicate acknowledgment [20].

The first three actions are called fast retransmission. Third and forth actions are called fast-recovery [11, 20, 24] and they are all part of the retransmission mechanism

in TCP.

However, TCP can recover only one dropped packet per window of data [37]. If more than one packet is dropped TCP will timeout and its performance will decrease dramatically.

TCP cannot recover multiple drops because it exits retransmission mode (fast retransmission/fast recovery) after any new acknowledgment. So if two or more packets are dropped from the same window boundaries (i.e. multiple drops occur from the same set of packets in flight) TCP resends the first one then directly exits the retransmission mode after any new acknowledgment and this will prevent resending the rest of the dropped packets.

However, TCP can deal with this problem using the retransmission timeout. So since some packets are dropped but not yet acknowledged TCP will wait for their acknowledgment which will not occur and so a retransmission timeout occurs and then TCP will resend them again.

From that we can see that in the case of multiple packet drops the meaning of the new acknowledgment has changed. TCP fast retransmission [11] assumes one packet will be dropped per window and hence a new acknowledgment after packet retransmission usually indicates that all packets have been received and hence it is safe for TCP to exit retransmission mode. However, some times even if TCP receives a new acknowledgment it may actually be an acknowledgment for part but not all of the sent window. This acknowledgment is called a partial acknowledgment [37] and TCP is not prepared to deal with it. So, if a burst of packets were dropped then only the first packet will be treated by fast retransmission and the rest will trigger timeouts.

However, although the timeout mechanism will guarantee resending of all dropped packets it will also reduce the TCP congestion window (sending rate) to

the minimum. Because of that it is desirable to avoid falling into timeout events by resending all dropped packets as we will see later.

Figure 2.4 shows an example of this scenario. In this example packets 3 and 5 were dropped. The sender resends packet 3 after receiving 3 DACKs and then a new acknowledgment (partial acknowledgment) is received. However TCP exits fast retransmission even though packet 5 is dropped and not retransmitted and it waits for a new acknowledgment for packet 5 which will not occur and eventually will timeout. The timeout will trigger the retransmission of the dropped packet 5 and also unnecessary transmission of packets 6 and 7.

In chapter 4 we will propose a modification to TCP retransmission mechanism in a way that allows TCP to resend all lost packets at the same time and reducing unnecessary retransmissions as much as possible . This will help to improve TCP performance specially with transmission errors where multiple packet drops can occur for several reasons like mobile base station hand off or temporary signal fading on wireless networks.

2.5 TCP Reaction to Transmission Drops

When talking about TCP performance over networks with lossy nature, most of the literature is directed to wired-wireless, ad-hoc and mobile networks [25]. This is due to the increase importance and usage of these type of networks and the fact that they become an important part of today's networks. Also this covers any network with links of low quality and reliability (although the main application in practice may still the wireless and mobile networks). Following we will explain the problem.

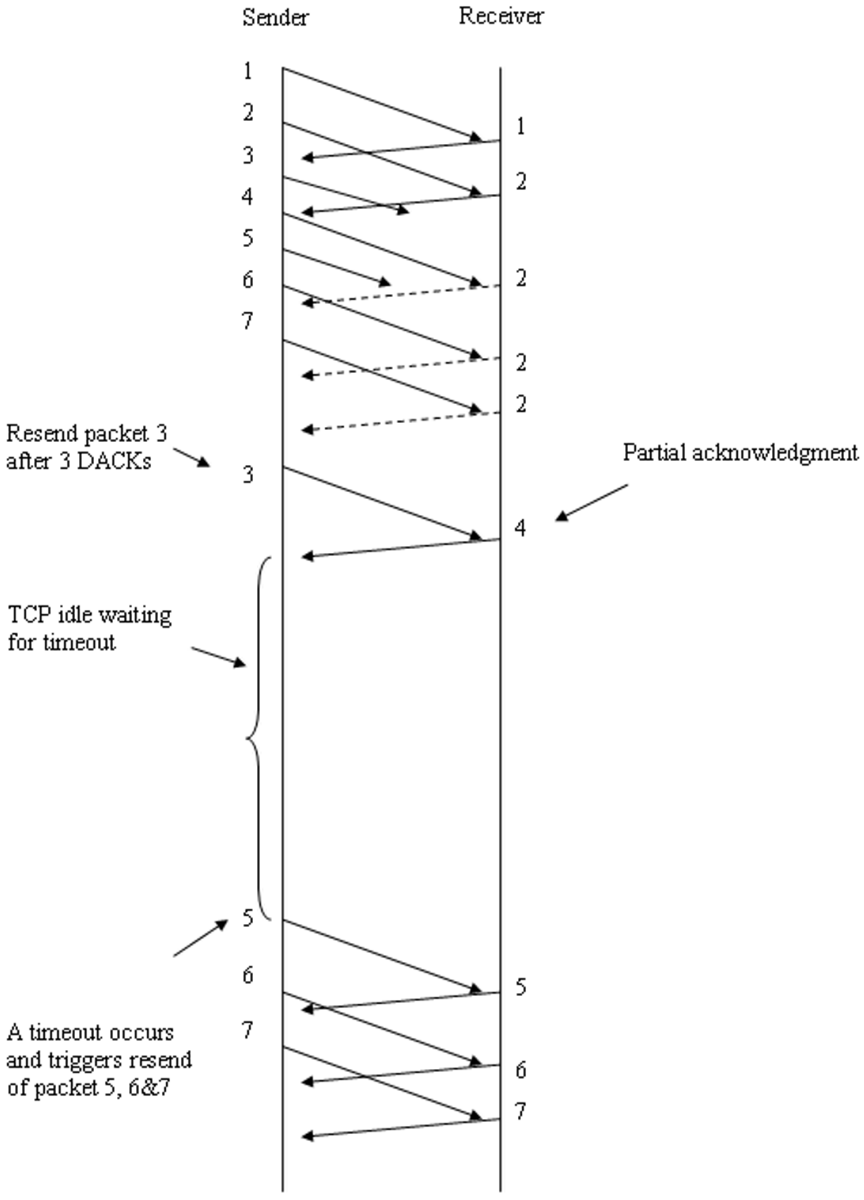

Figure 2.4: TCP with multiple drops

2.5.1 The Problem

The degradation of TCP performance when it operates over networks of a lossy nature is a well known problem and it has been discussed in many publications, see for example [5, 12–17, 19]. However, the first attempt to understand what causes TCP to perform badly over wireless networks was done by Caceres et al. [13]. They indicated that this problem has been mentioned in earlier publication, like [38–40] but without explaining what causes TCP to reduce its throughput.

TCP uses errors as an indicator of congestion and, based on that, it reduces its transmission rate. However, it confuses the congestion errors with errors caused by unreliable links and assumes that congestion is occurring whenever a transmission error is detected. Hence, TCP reduces its transmission rate even if there is no congestion. So TCP deals with both kind of errors (i.e. congestion errors and transmission errors) as congestion. In the case of transmission errors, like the ones caused by wireless links, it is not required to reduce TCP throughput aggressively as in the case of congestion. Many solutions, as we will see later in chapter 3, resend the lost packet and will not reduce TCP performance (congestion window) at all.

The problem is not because of TCP itself as originally defined in [4]. The congestion control mechanism which was introduced to TCP first in [10] is responsible for causing this problem.

We identified three areas where congestion control mechanism affects TCP performance negatively in the case of transmission errors: the congestion window cut mechanism, the retransmission mechanism and the retransmission timeout mechanism. In chapter 4 we will discuss this issue in more detail and we will propose solutions to the problem.

Figure 2.5 shows the normal behaviour of TCP when there are no errors. The first phase is the exponential increasing slow start then the congestion avoidance (lin-

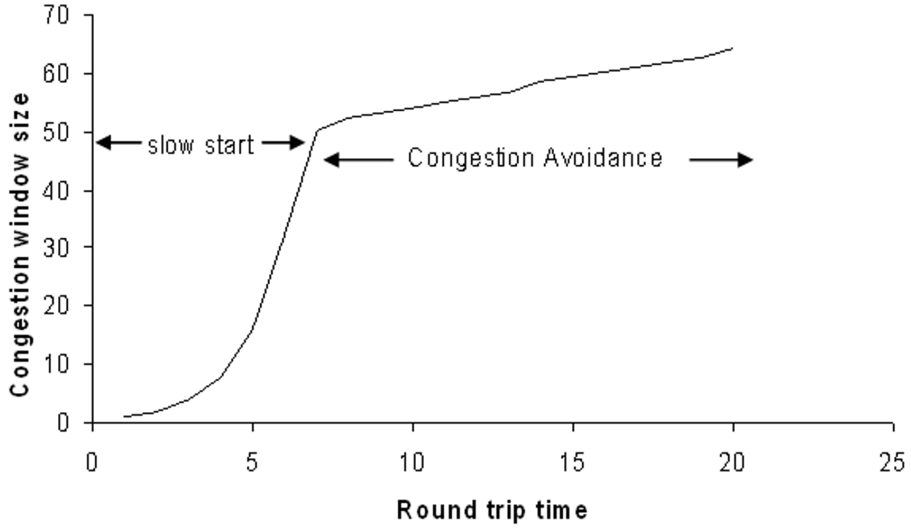

Figure 2.5: TCP behaviour in absence of errors

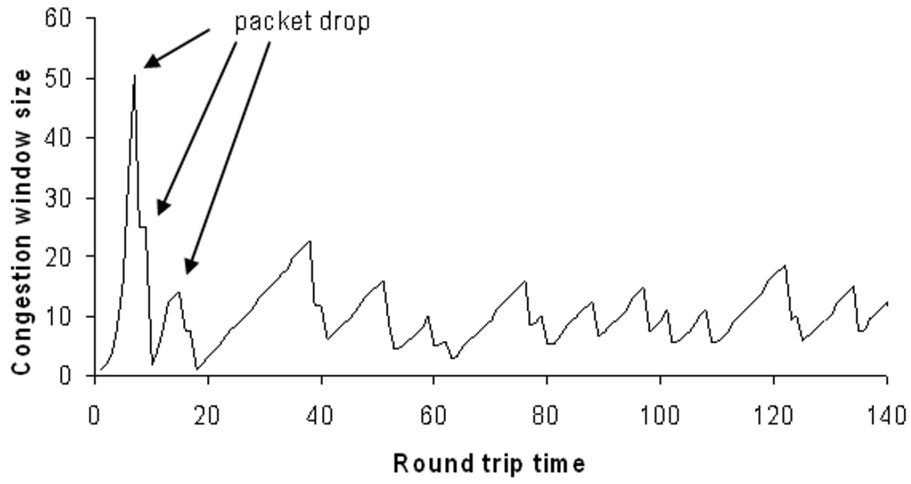

Figure 2.6: TCP with transmission errors

ear increase). However, when TCP operates over links suffering from non-congestion errors, like wireless errors, it cuts its congestion window size assuming that a congestion is occurring as we can see in Figure 2.6.

The transmission error rate in figure 2.6 is 1% and we pointed out the first three drops in the figure. In both figures the y-axis shows the congestion window size in packets and the x-axis shows the connection round trip time.

2.6 Summary

TCP performance decreases when operating over heterogeneous networks with links of lossy nature. In order to understand the problem, in this chapter, we started by giving an explanation of TCP and its basic functionality.

We then explained some of the important components in TCP namely sliding window algorithm, slow start, congestion avoidance, retransmission mechanism and timeout mechanism.

Finally we explained how TCP reacts to transmission errors and what causes TCP to perform badly over networks with lossy links. Research shows that TCP confuses congestion and transmission errors, and thereby deals with transmission errors as if they are signs of congestion in the network. This confusion causes TCP to reduce its throughput aggressively over lossy links.

In the next chapter we will survey some of the end-to-end solutions to the problem presented here.

Chapter 3

Enhancing TCP Performance: Related Work

3.1 Introduction

In this chapter we will present different solutions to overcome the performance degradation problem TCP faces when working over lossy links as we explained in chapter 2. Many solutions have been proposed but we will concentrate on end-to-end solutions that require no help from the intermediate network.

3.2 Solutions for TCP Performance Over Lossy Networks

Many solutions have been proposed to overcome the problem of TCP bad performance over lossy links (like wireless networks). Some solutions were in the transport layer and some solutions were in lower layers like the link layer.

Balakrishnan in [15] divided the solutions into two general categories: 1- solutions

that make TCP unaware of the errors that happens in the link so TCP thinks that it works on reliable links with no transmission errors, for example Snoop agents [14]. 2- approaches to try to make TCP aware of the errors caused by the lossy link and make TCP avoid using congestion mechanisms for this type of errors. However other authors like [5] divide these solutions into the following more specific categories: Link Layer, Split Connection and End-To-End solutions.

In the link layer solutions the aim is to completely hide the errors that occur in the link so TCP will be unaware of them and hence it will not reduce its transmission rate as a reaction to those errors. In general, link layer solutions are used for wired-wireless networks and they can be located at the base station which connects the wired network with the wireless link just before the receiver. They monitor the packets that pass the base station from one end to another and keep record (and sometimes copies) of the packets sent and set a retransmission timeout for each packet. When the wireless link drops a packet either a timeout will occur or duplicate acknowledgments will be received at the base station. The base station then resends the lost packet and suppresses the duplicate acknowledgment at the base station so TCP does not notice the drop and hence will not need to reduce its transmission rate.

A good feature of this approach is that it preserves the end-to-end semantics of TCP since it does not break the connection (i.e. the connection negotiation and maintenance remains between the sender and the receiver only). However, the problem is that sometimes this method cannot completely hide errors from the TCP sender. For example when a mechanism like Snoop resends a dropped packet but the packet is dropped again due to high error rates and then the TCP timeout for this packet occurs before Snoop has a chance to resend it again. This could happen because of the mismatch between the TCP and Snoop retransmission timeout

mechanisms. In principle the Snoop RTO should be shorter than the TCP RTO but this is not always true [5, 15]. Moreover, sometimes Snoop's aggressive retransmission may cause congestion at the base station which may reduce the link utilization. Examples of Link layer solutions are TULIP [41], Snoop [14] and AIRMAIL [42].

In Split Connections protocols the aim is to divide the problem into two smaller ones. This is done by separating the wired link connection from the wireless link connection. This is usually done at the base station where two connections are maintained. One normal TCP connection from the wired host to the base station and another wireless connection from the base station to the mobile host where a new protocol that can handle wireless errors is implemented. The base station plays the role of the interface between the two connections [5] [17]. The TCP connection from the sender ends at the base station and then the base station starts a new connection with the receiver.

A good feature of this method is that we do not need to do any changes at the sender because the sender does not need to deal with the errors on the wireless link. However, the sender is not now negotiating the connection with the end receiver so the connection between the sender and the receiver is broken and the end-to-end semantics of TCP no longer hold. An example of this category is I-TCP [17] and M-TCP [43].

The last category is the end-to-end solution. We will talk about this next in more detail by explaining some solutions under this category.

3.3 End-to-End Solutions

In general, most of the end-to-end solutions, as the name indicates, try to deal with the problem at the end point of the connection (sender and receiver) and do not

expect help from the network so they look at the network as a black box. The main advantage of this approach is that it does not add overhead to the network. However, some proposed solutions from this category use some sort of indirect feedback from the network as we will see later.

Mainly the following techniques try to find ways to recover from drops (congestion and transmission) efficiently. Some of these solutions were designed for a wireless environment and some were introduced before introduction of wireless technology. However, since the aim of all these solutions is to recover from errors efficiently, they can be a used for improving TCP performance over networks with non-congestion errors.

3.4 Congestion Drops

In the following we will begin with techniques designed to recover from congestion drops. Later we will present techniques designed for congestion and transmission drops.

3.4.1 Retransmission Timeout (RTO)

In Retransmission Timeout TCP attaches a timer with each sent packet, and when the timer expires before receiving acknowledgment for that packet, TCP resends the lost packet and sets the congestion window to the minimum allowed size. For more about how RTO is calculated using the RTT see section 2.4.4.

RTO is one of the first methods provided to TCP to recover from errors. However, it is most efficient when the congestion is serious and the network needs more time to drain the congested nodes. On the other hand, if the drops are caused by transient congestion then it is better to resend the lost packet without waiting for a timeout

to occur. This idea is the base of the fast-retransmission mechanism which we will talk about in the next section.

3.4.2 Fast Retransmission (TCP-Reno)

In earlier implementations of congestion control mechanisms, there was an assumption that errors due to segment damage are rare (less than 1% of the sent segments [10]) and so it is assumed that most of the segment loss is because of congestion [20] [10]. As a result, when there is a high packet damage rate or when the congestion loss rate exceeds 1%, the TCP performance will suffer badly. According to Jacobson [11] TCP will lose between 50% and 75% of its throughput when the error rate reaches 1%.

This shows how congestion control mechanisms are intolerant to high error rates. This behaviour can be explained if we return to the combined slow start congestion avoidance algorithm explained in Stevens [20] and Allman [24]. In the algorithm Stevens [20] explained how TCP should react to congestion as follows: If there is a duplicate acknowledgment then TCP should set the slow start threshold *ssthresh* to half of current window size (the window size when the error happened) and then enters the slow start mode when a timeout occurs. This way all drops will be recovered by entering slow-start, however, TCP performance will decrease sharply.

For this reason, Jacobson [11] suggests that TCP can use the knowledge brought by duplicate acknowledgment to resend the lost packet (a fast retransmission) and then there is no need to enter slow start because the duplicate acknowledgments indicate that these packet have left the network and there is more space for new packets to be injected into the network [24] so no need to reduce the *cwnd* to one segment by entering slow start. Instead, TCP enters congestion avoidance by reducing the congestion window to the half of the current window size.

Another reason for not using slow start is given by Stevens [20]. He noted that because we know that there is still data flowing in the connection because of the duplicate acknowledgments we received, we do not want to cut this flow by entering slow start [20].

3.4.3 Fast Retransmission Phase (TCP-NewReno)

When Stevens [20], in the RFC 2001, explained the slow start algorithm, he indicated that the first step in the algorithm is to initialize the slow start threshold variable *ssthresh* to a high value (65535 bytes). Also Allman [24], in RFC 2581, indicated that *ssthresh* could be set to an arbitrary high value. However, Heo [31] noted a problem in this step of the algorithm that may affect TCP performance.

The problem is that during the start up phase of TCP connection (slow start), the sending rate grows exponentially until the congestion window reaches the *ssthresh*. So, giving *ssthresh* a high value will inject the network with high number of segments in a short period of time.

However, the network may be unable to handle that amount of data at once and, hence, some packets may be dropped due to congestion. Moreover, due to this congestion, more than one segment may be dropped from the same window [31] and this will create problems to the TCP fast retransmission mechanism proposed by Jacobson [11].

The fast retransmission algorithm [11] can handle only one drop per window and hence if more than one segment is dropped from the same window, only one will be resent by fast retransmission and TCP will recover from the other losses by using a retransmission timeout (RTO) which will initiate the slow start algorithm which will reduce the congestion window size to its initial value (usually one segment) and TCP performance will suffer badly [31].

To understand why the fast retransmission algorithm can not recover from multiple drops, Stevens [20] indicated that the fast retransmission algorithm is terminated whenever a new acknowledgment is received. This new acknowledgment is assumed to acknowledge all packets sent after the lost segment up to the window size. However, if multiple segments were dropped, this acknowledgment will acknowledge only the segments that have been received correctly up to the second drop. Hence, fast transmission will be terminated before resending all lost segments and TCP will enter a series of retransmission timeouts causing the performance to degrade.

As a solution, Heo [31], suggested a change in the fast retransmission algorithm so it will not exit until it receives an acknowledgment for all dropped segments. This is done by ignoring the new acknowledgments that acknowledge only part of the sent segments and repeating fast retransmission until the sender receives an acknowledgment for all sent segments. This way, there is no need to wait for the retransmission timeout (RTO) to force resending the rest of the lost segments. Floyd et al. [37] call the intermediate acknowledgments the partial acknowledgments.

A new variation of TCP was proposed based on these modifications and called TCP-NewReno [37]. Also, Floyd et al. [37] has introduced two options of NewReno regarding when to reset the retransmission timeout: the first option is called slow-but-steady NewReno and the second is called impatient NewReno. In the former the timeout clock is initialized after each partial acknowledgment. This way TCP will stay in fast recovery mode as much as possible but as the name indicates, the resending rate will be as low as one packet per round trip time (RTT). However, in the impatient NewReno TCP will reset the RTO only after the first partial acknowledgment so if there are too many packets dropped from the same window then RTO will eventually expire before receiving a new acknowledgment and, hence, TCP will enter slow start [37] and resend all dropped packets and cut the congestion window

at the same time.

3.4.4 Selective Acknowledgment (TCP-Sack)

The original idea of using selective acknowledgments (SACK) was proposed initially by Braden and Jacobson in [44]. However, detailed implementation and improvements to the idea were proposed later by Mathis et al. in [45].

Selective acknowledgment is a change to the way the TCP receiver reacts to receiving new packets. Usually when the TCP receiver receives a new packet it sends an acknowledgment to the sender that carries the received packet sequence number which indicates to the sender that all previous packets up to this one have been received successfully at the sender because of that it is called cumulative acknowledgment [4]. This way new acknowledgments will be sent only if the packets are received in order, otherwise the acknowledgment will be sent for the last in-order packet received (duplicate acknowledgment).

However using selective acknowledgments, the receiver will send an acknowledgment for each packet no matter in what order it has arrived. This way the sender will have a clear idea of what packets have been received successfully and this will solve the problem we described before when more than one packet is dropped from the same window [45].

Also using selective acknowledgment will allow TCP to resend all lost packets without the need to do unnecessary retransmission of packets already received [46]. Using selective acknowledgment does not require the overhead of extra traffic since it is sent over normal acknowledgments [45] [46].

However, a disadvantage of the implementation explained in [45] is that it requires the use of a retransmission queue to save unacknowledged segments. Also it requires the TCP sender to keep a record of the received acknowledgments. This

may requires more memory usage and perhaps more processing power for sorting and comparing sequence numbers for segments in the queue specially when TCP uses a large sending window (congestion window). Moreover, SACK is helpless when retransmission timeout occurs; all segments in the retransmission queue will be resent even if they have been sent before [45]. Last but not least, applying selective acknowledgment requires changes to both sender and receiver sides which may be hard in real networks.

On the other hand, accumulative acknowledgments which used in most TCP variations is simple and allows easy management of incoming packets with no need for extra memory or processing as in the case of selective acknowledgment. For example it is easy to discover receipt of duplicate copies of a packet by simply comparing the packet sequence number with the last acknowledged packet number [4]. This way TCP does not need to keep a record of the received packets and only needs to save the last in-order received packet sequence number. Another advantage of this approach is that if an acknowledgment is dropped then it is enough to receive another acknowledgment with higher sequence number since it acknowledges all packets with lower sequence numbers.

3.5 Congestion and Transmission Drops

In this section we will present techniques designed to improve TCP performance for congestion and transmission drops.

3.5.1 Congestion Predictors

In this type of solution, TCP uses techniques such as delays on the links (round trip time) like the CARD [47] technique (CARD stands for Congestion Avoidance using

Round Trip Delay) or the connection throughput in the case of the Tri-s scheme [48] and the Vegas scheme [49] to predict if there will be congestion and then control the inflation and the deflation of the congestion window based on this prediction. If the predictor does not see congestion happening in the near future then it suggests increasing the congestion window. On the other hand, if the predictor notices that congestion is coming then it suggests that TCP decrease the congestion window. As we can see, unlike TCP, drops are not used here to control the growth of the congestion window.

In theory, the perfect predictor will eliminate congestion errors since it will detect and avoid congestion before it happens. So, if an error happens then it can be considered to be caused by the link failure (like wireless errors) rather than by congestion. We will see later how congestion predictors can be used to build error discriminators. Following, brief explanations of some congestion predictors.

3.5.1.1 TCP-Vegas

TCP-Vegas [49,50] is a modification to the congestion control mechanism in standard TCP [10,11]. It aims to reduce the congestion losses and to increase TCP throughput by predicting the available capacity on the link and trying not to exceed it.

According to the Vegas authors in [49], Vegas has increased the throughput of TCP up to 70% more than older implementations of TCP (TCP-Tahoe & TCP-Reno). Also Vegas has reduced the losses in the link up to 50% [49].

We will give here an extended explanation of TCP-Vegas because of its importance and since some other solutions are based on Vegas as we will see later.

TCP-Vegas introduces changes to TCP in four areas as follows:

Timeout computation: The authors of Vegas have noticed from experiments over the Internet that the timing mechanism used in previous implementations of TCP

is not accurate and that computing round trip time (RTT) using current timing mechanisms has given higher RTT estimations. This makes TCP take up to three times longer to recover from losses [49]. A new mechanism has been introduced based on using a time stamp for each packet and computing the round trip time by comparing the packet's time stamp with its acknowledgment time stamp. This way a more accurate retransmission timeout can be computed.

Retransmission of lost packets: TCP-Vegas introduces a new retransmission mechanism by changing the way TCP responds to duplicate acknowledgments. TCP needs to receive three duplicate acknowledgments before it retransmits the lost packet. However when Vegas receives the first duplicate acknowledgment for a segment it compares the time stamp with the current time. If the difference is more than the computed timeout then it triggers retransmission without waiting for more duplicate acknowledgments to come [49]. As we can see this will add overhead to the system to record a time stamp for each segment and save it until it receives an acknowledgment. However, the authors indicated that the overhead of using TCP-Vegas will not exceed 5% more than older implementations [49].

The other area in which TCP-Vegas provides changes is in congestion avoidance: TCP-Vegas has made dramatic changes to the congestion avoidance mechanism used in TCP by making TCP to increase/decrease the sending rate, not based on packet drops as in TCP, but based on prediction of available link bandwidth.

Vegas estimates an expected throughput and an actual throughput for the connection. The expected throughput is computed using the current window size and the minimum RTT seen so far. The actual throughput is computed using current window size and last RTT reading.

Then Vegas compares the expected throughput and the actual throughput and updates the sender window according to the comparison results as following: If the

actual throughput is less than the expected one then TCP is unable to utilize the link because there is congestion and hence it should decreases the window size [49]. On the other hand, if the actual throughput becomes closer to the expected throughput then it is safe to increase the window size. The increase and decrease in the window size is linear unlike TCP which uses Jacobson's AIMD [10] mechanism (Additive increase multiplicative decrease) to update the congestion window.

The Vegas algorithm is expected to prevent congestion from occurring and, hence, reduce congestion drops dramatically.

Slow Start: In Vegas, the congestion predictor, explained above, is added to the slow start mechanism. Another modification Vegas makes to slow start is that the update of window size during slow start is not done every RTT; instead it takes two RTTs before increasing the window size. This is done to give the algorithm chance to measure the actual throughput between updating window size [49].

Hengartner et al. in [51] have reviewed each of the modifications Vegas did to TCP. Their results show that the new retransmission technique has improved the performance noticeably because it was able to avoid timeouts during multiple packet drops from the same window. It does this by performing retransmission when its new timeout mechanism expires even before receiving duplicate acknowledgments.

However, the results in [51] showed that TCP-Vegas suffers from performance degradation when it coexists with versions of TCP that use the AIMD mechanism like TCP-Reno. This is because the AIMD mechanism is more aggressive in grabbing the link bandwidth because it keeps increasing the window size until an error occurs while Vegas tries to prevent causing drops and hence it keeps smaller window size. This indicates that the congestion predictor in Vegas sometimes has a negative impact on the performance [51]. Also, we will see later how the authors in [52] have confirmed this fact (i.e. Vegas predictor poor performance) when we talk about

using the Vegas congestion predictor in an error discriminator.

3.5.1.2 TCP-Westwood

Mascolo et al. in [53] proposed a modification to the congestion avoidance algorithm used in TCP-Reno, which uses duplicate acknowledgment and timeout as an indicator for congestion and to update the sender window [10]. However, duplicate acknowledgments do not give indication of the type of the error (congestion or transmission error). For this reason Mascolo et al. [53] suggested that the TCP sender should do continuous estimation of the bandwidth and update the window size according to that estimation [53]. This way TCP will send in a rate that will occupy the available bandwidth only and hence any error could be considered safely as a transmission error. Westwood estimates the available bandwidth by monitoring incoming acknowledgments and assumes this rate reflects available link capacity in the forward path [53].

Also TCP-Westwood [53] suggested that TCP does not need to halve the window size when errors happen, like TCP-Reno. TCP-Reno halves the window size whenever there is an error and hopes this action will solve the congestion and, at the same time, it increases the congestion window linearly to utilize the available bandwidth without more investigation of the link status. In contrast, after each drop TCP-Westwood [53] uses the estimated bandwidth-delay product to set the sender window according to the current congestion level [54].

The authors of TCP-Westwood [53] reported big improvements in TCP performance, especially over networks suffering from transmission errors like Wired-Wireless networks [53]. This improvement has been confirmed by Grieco & Mascolo in [55]. Also the experimental results reported in both [53] and in [55] showed that TCP-Westwood has maintained fair sharing of the bandwidth and it does not lead

to starvation of TCP-Reno connections.

However Biaz et al. [56] did experiments on TCP-Westwood when coexisting with non-TCP traffic on the reverse link and their results indicate that TCP-Westwood could not estimate the link capacity correctly when a non-TCP traffic exists in the reverse path. This result can be explained since bandwidth estimation in TCP-Westwood is based on taking the average rate of received acknowledgments and, since the added traffic in the reverse path could add additional delay to the received acknowledgments, TCP-Westwood will underestimate the available bandwidth.

3.5.2 Error Discriminators

All methods that try to understand the cause of the error and to act differently to each type of error based on that understanding are called error discriminators.

Some error discriminators deal with the network as a black box and do not need any feedback from the network in order to discriminate errors. Other types of error discriminators use help from intermediate networks in order to understand the cause of the error.

In the following, we will talk about both types and we will start with error discriminators that depend upon the network to help distinguishing errors. As far as we know this is the first attempt to classify error discriminators.

3.5.2.1 Network Dependent Error Discriminators

Network dependent error discriminators are actually based at the end-point of the connection but use help from the intermediate nodes. However, although they are not totally end-to-end we mention them here for two reasons, first all network dependent error discriminators explained in this section use already popular active queuing mechanism techniques like the use of explicit congestion notification through

RED [57] (Random early dropping) queues. Second reason, is that we want to complete the picture about the error discrimination techniques since our work will use an error discriminator as we will see later.

The advantage of this approach is that both end hosts can have detailed information about the cause of drops and the network status.

However, if we want to apply this approach in a large network we may need a wide-scale change to the network components (i.e. mainly we need to change the routers if we want notification for congestion drops and we need changes in the wireless base stations if we want wireless drop notification).

Following I will explain briefly some network dependent error discriminators.

3.5.2.2 TCP-Casablanca

The key idea TCP-Casablanca introduces [56] is as follows: Congestion errors and transmission errors usually happen randomly and this is basically why it is difficult to differentiate between them. However, if we can *"de-randomize"* [56] the congestion errors by making congestion errors take a non-random form then it will be easy to discriminate the non-random congestion errors from the random wireless errors [56].

The mechanism works as follows: The sender marks each outgoing packet with one of the marks (in/out) in a consistent pattern, for example by marking four packets with (in) and the fifth packet (out) and so on. When congestion occurs at intermediate nodes there should be a biased queue-management mechanism that drops only the packets marked with the (out) mark. This way, the receiver receives the packets with a consistent pattern of drops, because only packets marked with (out) are dropped, so the receiver recognizes that the errors are congestion errors.

On the other hand, if a wireless error occurs, then the drops will be random among all packets (in-marked and out-marked) and, hence, the receiver can recognize

that these random errors are wireless errors [56].

If the receiver diagnoses a wireless error it marks the acknowledgments with an explicit loss notification (ELN). When the TCP sender receives a duplicate acknowledgment, because of error, it checks if the acknowledgment contains ELN and, if so, TCP considers the loss to be wireless loss; otherwise it considers it to be a congestion error [56]. In case of congestion error TCP cut the congestion window, otherwise it only resend the packet and does not cut the congestion window.

So, as we can see applying this mechanism requires mainly four changes to TCP sender/receiver and the network: First adding an error discriminator at the sender (which is called Casablanca) and acting according to the ELN signals it receives. Second, the receiver should be able to deduce when a random or non-random drop occurs and to send ELN if a random error occurs . Third, an active queuing mechanism should be implemented in the bottleneck, which will drop only packets marked with the (out) mark. Finally the packet format should be altered to add in/out marking and ELN. The reset of the protocol is based on the NewReno [37] version of TCP.

The authors indicated that the Casablanca discriminator has achieved high accuracy in discriminating between congestion and transmission errors and, using it in TCP, gave significant (above 100%) improvement in TCP performance [56]. Accuracy is a crucial component in this error discriminator since it uses an aggressive action toward non congestion drops by not cutting the congestion window size for these drops and keeps it as big as it was before the drop.

3.5.2.3 TCP-Ifrane

TCP-Ifrane [56] is a sender based version of TCP-Casablanca [56]. In TCP- Ifrane changes are made at the sender only and not the receiver. When the sender sends a

packet it records whether this packet is marked as out or in. If the sender receives a duplicate acknowledgment indicating that a packet is lost, it looks at its record and sees if that packet was marked out or in when it was sent. If the packet was marked out the error is considered to be congestion error otherwise it is considered as wireless error [56]. TCP-Ifrane was found to give higher throughput than TCP-Casablanca; this is because it has less congestion accuracy and hence it slows down less than TCP-Casablanca [56]. However, the effect of TCP-Ifrane's accuracy was not studied by the authors in [56].

3.5.2.4 Explicit Congestion Notifications

Explicit congestion notification [58] was first introduced to help TCP avoid congestion by allowing intermediate nodes to set a congestion notification bit in the IP header whenever congestion is expected. The TCP sender will respond to this notification by reducing its transmission rate. An Active Queue management mechanism (e.g. RED [57]) is placed at the congested nodes and becomes responsible for marking packets when congestion is expected (in case of RED the packets will be marked when the queue reaches a particular threshold).

Using ECN requires changes in both the TCP sender and receiver. Also it requires the use of AQM at the congested nodes. However, using ECN does not require changing the TCP congestion mechanisms since TCP responds to ECN in the same way as it responds to a packet drop.

Dawkins et al. in [59] has proposed the use of ECN to improve TCP performance over wireless links by modifying the way TCP responds to ECN. Biaz [60] explains the technique as follows: If a drop is detected by receiving duplicate acknowledgments, then we look if we have received an ECN in the near past. If ECN is received before the error happened, then this is a strong indication that this error

is caused by congestion. This is based on the understanding that, in ECN-capable connections, ECN should always happen before congestion drops. So, if the ECN preceded the drop then TCP considers this drop to be congestion drop and acts by reducing the senders window size in order to resolve the congestion.

However, if the drop happens while not preceded by ECN, its then considered as a wireless error and and TCP does not reduce the senders window size [60]. However, we still need to retransmit the lost packet. The authors in [59] argue that this approach will improve TCP performance over networks with transmission errors like wireless networks specially those suffers from high error rates.

However, Biaz in [60] studied the possibility of using ECN to distinguish between error types. He argues that this approach is not an accurate method to differentiate between congestion and transmission errors and showed that transmission errors can be random so that the probability that ECN will precede a congestion error is approximately the same as the probability that ECN will precede a transmission error [60].

So the authors in [60] proposed that instead of using ECN directly to infer the type of the error; TCP should also look at the state of the sender. If the sender was in congestion avoidance phase then the drop is probably a transmission drop. However, if the sender was in slow start phase then the drop is considered congestion drop. The new protocol is called TCP-Eaglet [60] and it showed improvement over standard TCP performance.

3.5.2.5 TCP-Jersey

Xu et al. [61] has suggested using the estimated bandwidth instead of errors to tell TCP when to decrease sender window size, which is an idea similar to Westwood [53] but with a different implementation. The available bandwidth is estimated based

on the rate of arrival acknowledgments. High acknowledgment arrival rate means packets can get to the other end fast and hence high network capacity. Moreover, in this approach the nodes in the middle should be able to mark packets when congestion is expected in order to notify the sender [61]. So this method is a combination of TCP-Westwood and ECN error discriminator [59] except that it differs in some implementation details in both cases.

However, like TCP-Westwood, TCP-Jersey may suffer from performance degradation when coexisting with non-TCP traffic on the reverse link because it cannot estimate the link capacity correctly since the added traffic in the reverse path can delay the acknowledgments, so it will underestimate the available bandwidth.

However, improvement has been done to TCP-Jersey to overcome this problem. The improved version is called TCP-New Jersey [61] and it uses acknowledgment timestamps [62] instead of acknowledgment arrival rate to calculate estimated bandwidth which solves the problem of delayed acknowledgments because each acknowledgment has a time stamp which allows the sender to calculate the forward path delay. The authors indicated that simulation results of TCP-New Jersey gave good results and show improvement in TCP performance particularly with reverse paths that suffers from congestion and lossy links [61].

3.5.2.6 Network Independent Error Discriminators

This kind of solution implicitly infers the cause of packet drop without the need of explicit notification from the network about the cause of the drop. In this kind, the solution is based at the end hosts (or one of them). The advantage of using this approach is to keep the changes to a minimum (to the end hosts) and there is no need to make changes to the network components, which may require wide scale changes. However, an obvious limitation to this approach is that the end hosts will

not have detailed information about the status of the congestion or the transmission drops and can only guess the situation using implicit signs from the network (like packet delay for example).

Some of these solutions are based on using congestion predictors like Vegas [49] or CARD [47] or Tri-s [48]. In this approach the discriminator works by taking input from the congestion predictor about the congestion status when a drop occurs. If the congestion predictor was predicting congestion then the drop is considered to be congestion loss. However if the predictor was suggesting increasing the sending rate, because it does not predict any congestion in the near future, then the drop is considered to be caused by link error [52] .

Also we must notice that as [52] indicated, designing an accurate error predictor is important since mistakes of distinguishing transmission errors from congestion errors could cause unnecessary congestion which is usually avoidable by using normal congestion control algorithms [52]. For example, if a congestion error is mistaken to be a transmission error then TCP will not decrease the window size and this will make the current congestion much worse.

Experiments were performed by Biaz and Vaidya [52] on three different error discriminators based on congestion predictors: the CARD [47], Tri-s [48] and Vegas [49]. Unfortunately the results obtained by Biaz and Vaidya experiments in [52] show that these congestion predictors are no better than a random loss predictor. From these results, Biaz came to the a conclusion that these three congestion predictors are not suitable as an accurate error discriminator.

The reason which leads to the failure of these methods to make a good error discriminator is that they assume that if one TCP increases its congestion window then the network delay will increase. So they assume that one connection can affect the whole network. Using this assumption, if TCP is able to gain high throughput

then this is an indication that the network is not congested. On the other hand, if TCP is able to gain only small part of the expected network throughput then this means that a congestion exists.

However, in [63] the authors showed that when TCP increases its sending rate the RTT could go either way (i.e. increase/decrease). They showed that the correlation between a single connection sending rate and the RTT is weak [63]. This is because usually a single connection forms a small part of the network aggregate traffic.

However, the authors in [63] also emphasized on the sensitivity of the network delay to the total load, which makes the measured RTT a good indication of congestion events and hence RTT can be used to build an effective error discriminator, as we will see in chapter 6.

In the following section we will present briefly some error discriminators based on congestion predictors and show how they work.

3.5.2.7 Error Discriminator Based on Vegas Congestion Predictor

Based on the Vegas predictor [49] described earlier, Biaz and Vaidya [52] proposed an error discriminator that computes the difference between expected throughput (link capacity) and the actual throughput in order to predict congestion and use this difference to define a new variable *fVegas*. The difference is computed as follows: D = expected throughput - actual throughput

If $D > 0$ this means that TCP throughput is less than what it should be to utilize the link and this indicates that congestion exists in the connection path and hence any drop is considered to be a congestion drop. On the other hand if $D \leq 0$ this means that TCP throughput is actually able to utilize the link capacity and hence there is no congestion and any error is considered to be a transmission error.

The simulation results in [52] show that the Vegas based error discriminator has

achieved low to medium performance in terms of accuracy in defining error types. As we said before, this due to the assumption that the network will respond noticeably to the changes in a single connection window. This is not always true since, in large networks, a single connection forms a small fraction of the whole traffic [52] and this will affect the error discriminator ability to discover congestion errors.

This also applies to the next two error discriminators based on CARD [47] and Tri-s [48] congestion predictors.

3.5.2.8 Error Discriminator Based on CARD Delay-Based Congestion Predictor

Congestion Avoidance Round trip Delay (CARD) [47] is an approach to update the TCP sender window size without the need to have any feedback from the network. It is called [47] a black-box approach since it deals with the network as a black box and does not require any explicit feedback from the network. It works by analysing the relation between the round-trip delay and the throughput of the connection in order to predict the optimum window size that gives maximum throughput with minimum delay. The authors in [47] call it maximum Power where the power is the ratio of throughput and delay : $Power = (Throughput/Delay)$ [47]. The aim is to have maximum $Power$.

Unlike TCP, this approach does not use errors to update the window size which is approach similar to TCP-Vegas [50]. However Jain [47] did not provide a complete TCP solution like TCP-Vegas, instead, it gives a mechanism that can be used to replace Jacobson's [10] congestion avoidance mechanism in TCP.

The CARD [47] measures the change of the increase/decrease rate in the connection throughput and delay. When the network is fully utilized then any small increase in the throughput will result in a big increase in the observed delay. This

gives a good indication that the network is congested. However, when the network capacity is under utilized then the increase in the throughput will result in a small (or non) increase in the network delay.

Using this approach will add no overhead on the network since it requires no feedback from the network [47]. This approach assumes there is a single connection that can utilize the whole network capacity and hence increase/decrease the network delay [47]. As we said before this assumption is not always valid in real networks.

Biaz et al. [52] designed an error discriminator based on the CARD [47] congestion predictor. The discriminator uses the assumption used in CARD that if the network is not congested then the rate of change in the delay will be zero. However, when the network starts building queues with the increase in the TCP window size then the delay will change rapidly. The discriminator monitors the delay and the window size changes; if both are increasing then the drop is considered to be congestion drop otherwise the drops is considered transmission drop.

The results presented in [52] indicate that the error discriminator based on the CARD predictor is poor in discriminating between error types [52]. Again this because of the assumption used in CARD that a single TCP window size will affect the network delay.

3.5.2.9 Error Discriminator Based on Tri-s Throughput-Based Congestion Predictor

The Tri-s [48] congestion predictor proposed an approach to predict congestions in the link based on the throughput rather than errors. Its difference than CARD [47] approach is that Tri-s monitors only the changes in the connection throughput. Also this approach tries to find the optimal window size only at the beginning of the connection and fix it through the rest of the connection period. Only when a

major change in the connection happens, like when a new connection starts or an old connection terminates, the optimal window size is recalculated. Small changes during the connection are dealt with by buffering in the network instead of changing the sender window size [48].

An error discriminator based on this idea has been proposed in [52]. This is based on the assumption that if the network is free of congestion then the connection throughput will increase rapidly and hence any drop will be considered to be a transmission drop. However, if there is a congestion in the network the TCP throughput will decrease and any error will be considered to be a congestion drop. The results presented in [52] show a poor discrimination level and this is for the same reasons mentioned before for the Vegas and CARD based error discriminator.

3.5.2.10 TCP-Veno

TCP-Veno [21] applies changes to the Vegas [49] congestion predictors in order to differentiate between *congestive states* [21] and *non congestive states* [21] of the connection. If a packet drop occurs during a congestive state then it is considered a congestion drop otherwise it is considered transmission drop.

TCP-Veno estimates the number of packets buffered in the network and if this number exceeds a predefined threshold (3 in this case) then the the system enters congestive state [21]. It uses Vegas [49] congestion predictors to estimate buffered packets and, instead of updating the congestion window based on this information like Vegas, it uses it to differentiate between errors and uses TCP AIMD to update the congestion window.

The other change TCP-Veno proposes is to reduce the rate at which the congestion window increases during the congestive state. So instead of increasing the congestion window every RTT, the window is increased every other RTT if the

system is in the congestive state [21].

The authors in [21] reported noticeable improvement (up to 80%) for TCP-Veno over TCP-Reno in different scenarios. However, TCP-Veno suffers from the bad performance of Vegas predictor mentioned before which may lead to classify errors wrongly.

An important feature of TCP-Veno is that it cuts the congestion window even for transmission errors by a fixed factor of 4/5 [21] which may reduce the effect of poor discrimination ability. We could not find any other error discriminator that uses a special action in case of transmission errors.

Later we will propose a method to cut the congestion window in case of transmission errors based on the number of dropped packets instead of using a fixed factor as in TCP-Veno.

3.5.2.11 Receiver Based Error Discriminators

Most of the previous solutions are based in the sender side of the connection. Following we will describe some solutions which are designed to be in the receiver side of the connection.

In [64] the authors proposed a receiver based error discriminator that uses a heuristic method to discriminate between transmission and congestion losses. In this method the authors assume that the lossy link will be always the bottleneck of the connection, for example a low bandwidth last hop in a wired-wireless network. Hence, in the case of congestion all packets will be queued in the bottleneck in the wireless base station. So, when the base station sends the packets they will travel back-to-back on the wireless link. As a result, the TCP receiver can compute the inter arrival time of the packets and use it to determine the cause of the drop.

For example, if we have packets 1,2 and 3, then in normal cases there will be T

time between consecutive packets. However, if one packet is dropped, say packet 2, then the time between packet 1 and 3 will be at least 2T. From that the reciever can know that a drop in the wireless links has occurred.

However, if packet 2 was dropped before the base station because of congestion, then packets 1 and 3 will probably be queued in the base station because the wireless link is the bottleneck, the time between packet 1 and 3 will be less than 2T and hence the receiver can recognize that this error is due to congestion error [64].

The problem with this method is that it requires the wireless link to be the bottleneck (the one with least bandwidth) [64]. Also, as we noticed from the example above, this method works only if the wireless link is the last hop in the path and directly before the TCP receiver and also if a non-stop stream of data is being sent (bulk data) [64]. However, the simulation results in [64] showed that by using this method TCP could discriminate between wireless and congestion errors, in most cases, as good as a perfect error discriminator i.e. with accuracy around 100% of discriminating both types of errors.

A similar approach has been proposed in WTCP (Wireless Transmission Control Protocol) [65] but without the constraint that the base station should be the bottleneck. This is achieved by computing an average inter arrival time at the receiver (AvgT). When a drop occurs instead of comparing with T we compare with average AvgT. If current inter arrival time is within a predefined threshold from AvgT then the error is considered a transmission error otherwise it is considered a congestion error. A promising result has been reported in [65] after using this approach.

Another receiver based error discriminator is proposed in [66] and called TCP-Real. TCP-Real uses the rate of receiving data at the receiver to detect congestion. It computes an expected receiving rate and an actual receiving rate based on the congestion window size and minimum RTT and current RTT. If the actual receiving

rate is less than the expected then the receiver signals the sender to increase its congestion window and if the expected rate is less than the actual the receiver signals the sender to reduce its congestion window (we can notice the similarity with TCP-Vegas [49] which uses same concept but at the sender).

Because this method uses the receiver to calculate the congestion window size it solves the problem when the return path is slower that the forward path by considering the the available bandwidth on the forward path only [66]. Experimental results in [66] shows that TCP-Real improves TCP performance when compared to TCP-Reno and TCP-Tahoe specially with the increase in the error rate. However, TCP-Real does not define a clear action for transmission drops and seems to keep the congestion window open.

3.5.2.12 Fast Recovery Plus

Fast recovery plus [67] has introduced a modification to TCP fast retransmission [11] and fast recovery [24] algorithms so it can discriminate between congestion and transmission errors. The idea is simple; the TCP sender maintains a counter of how many times the fast retransmission-fast recovery module is called by duplicate acknowledgments before receiving a new acknowledgment. The authors in [67] assumes that transmission errors will occur in small numbers per window of data compared to congestion errors. So the counting of the number of fast retransmission-fast recovery events can give an indication of the error type. If this number exceeds a preset threshold then the error is considered to be a congestion error otherwise it is considered a transmission error. The author in [67] did not explain how to choose the error threshold in order to decide the error type and we assume it is a fixed one that will be chosen based on the system experimental results.

The results shown in [67] presents a good improvement in TCP throughput when

Fast Recovery plus is used. However, like previous error discriminators, this method does not consider an action in case of transmission errors.

3.5.2.13 Spike Error Discriminator

The authors in [68] did a series of experiments on UDP performance in the Internet and they noticed that most congestion drops occur during specific periods related to noticeable increase in the packet trip time from the sender to receiver. They call these periods *spike-train periods* [68] since spikes appear in the packet trip time graphs when congestions occur. These spikes were found highly correlated with congestion events and hence congestion drops [68].

The authors in [1] used this idea to design an error discriminator which uses *spike-train periods* [68]. They define two states, the spike-state and, non spike-state. In the spike state the connection is considered in congestion state and any drop that occurs during this period is considered a congestion drop. During the non spike-state any drop is considered a transmission drop [1]. The system enters the spike-state if the packet trip time exceeded a threshold called $B_{spikestart}$ and ends when the packet trip time becomes below $B_{spikend}$ [1]. These thresholds are computed dynamically according to current relative one way trip time (ROTT) reading as follows:

$$B_{spikestart} = ROTT_{min} + \alpha(ROTT_{max} - ROTT_{min}) \tag{3.1}$$

$$B_{spikend} = ROTT_{min} + \beta(ROTT_{max} - ROTT_{min}) \tag{3.2}$$

Spike uses ROTT instead of round trip time (RTT) because it was designed for UDP applications where there is no acknowledgment so the authors used the relative one way trip time and since the sender and receiver clock may vary the term *relative* is used.

The Spike [1] error discriminator performed well under different scenarios where congestion and transmission errors were present. It was able to achieve high link utilization. However, its accuracy of distinguishing between error types was moderate (around 50%) and this has led to increased congestion in several cases [1].

In chapter 6 we will use an improved error discriminator based on Spike [1].

3.6 Summary: Action toward transmission errors

In this chapter our aim was to give an overview of the efforts to improve TCP performance in presence of errors (congestion and transmission). Some of the main end-to-end solutions are presented here and more related solutions can be found in [69–74].

From the solutions presented here wan can see that the main aim was to improve TCP performance when congestion and transmission errors coexist. However, we can categorize these solutions into two categories depending on how they solve the problem. The first category tries to distinguish between congestion and transmission errors and apply different actions for each case. All error discriminators like TCP-Casablanca [56] can come under this category. We will call them two actions solutions because in concept they can apply different actions at each case (i.e. congestion or transmission drops).

On the other hand other solutions apply one action which can only detects and response to congestion and will do nothing if there are no congestion drops (and only there are transmission errors). These kind of solutions usually apply techniques which by nature respond to congestion only like for example using TCP-Vegas [49] which uses expected and actual throughput to set the congestion window or TCP-Westwood [53] which uses Bandwidth-Delay product to set the congestion window

size which will be affected mainly by the change in the available bandwidth due to the congestion in the network. These solutions do not differentiate between error types but only respond to congestion (by increasing sending rate if there is no congestion and decrease the sending rate if there is congestion) so we call them the one action solutions.

However, in both one action and two actions solutions the TCP reaction to transmission errors is simply not to cut the congestion window and to keep the sending rate as it was before the error. Moreover, in the two action solutions when the protocol discovers transmission errors it implicitly implies that it should increase the congestion window (not just do nothing).

These assumptions give rise to questions about whether the current transmission action is enough or not. Authors like [56] indicated that the current transmission action used in error discriminators is a bad one. This is because it is simplistic and it ignores two facts: first it is very hard to have an end-to-end error discriminator with very high accuracy. Second, even with accurate error discriminators mismatches between error types can occur. Because of that some studies like [1,56,72] indicated that error discriminators usually increase the congestion loss rate noticeably.

Moreover, even the one action solutions can be affected by the lack of appropriate transmission action. This could happen when the technique used to discover congestion in the network fails to do so and hence no action is taken in case of congestion.

In our work we aim to propose a set of actions that can be added to error discriminators to use in the case of transmission errors. These actions should provide the following:

- These actions should be able to achieve the aim of any proposal which is to improve TCP performance when congestion and transmission errors coexist.

- These actions should be able to prevent increasing the congestion in the network which may occur because of the first aim.

In the next chapter we will discuss these actions in detail.

Chapter 4

Improving TCP Error Discriminators Reaction to Transmission Drops

4.1 Introduction

Many end-to-end proposals to improve TCP performance for transmission drops, particularly error discriminators, have been based on the idea that if we can discriminate between errors correctly then the reaction to transmission drops can be as simple as to not cut the congestion window. See for examples [1, 21, 52, 60, 64, 67]. Hence, the main aim was to design an accurate error discriminator.

Our proposal is that as well as trying to increase the error discriminator accuracy we will also implement an efficient action for transmission drops which should have the following properties:

- It will increase TCP performance in case of transmission drops by decreasing the rate of cuts in TCP sending rate.

- It will not increase the congestion rate because of error discriminator mismatches between error types (i.e. low accuracy).

The last point is the main reason why it is required to have an accurate error discriminator. Since if the error discriminator wrongly identifies an error as a transmission error when it is actually a congestion error then it will not cut the sending rate, so leading to increased network congestion. However, to our knowledge, there is no perfect end-to-end error discriminator (also this is supported by [75, P111]) which can discriminate between errors with 100% accuracy. However, if we can maintain an action that will not increase the congestion level on the network then even an error discriminator with medium/low accuracy will be able to increase TCP performance and can avoid causing unnecessary congestion on the network at the same time.

When drops occur many mechanisms in TCP are affected. In particular three main mechanisms: the congestion window update mechanism, the retransmission mechanism and the timeout mechanism. In this chapter we will study the effect of transmission drops on each of these mechanisms and we will propose improvements.

The idea, as we will explain in more detail later, is that we will not totally ignore the errors if they have been classified as transmission errors, instead we will decrease the TCP transmission rate based on number of dropped packets per window. Also in the case of multiple transmission drops we will retransmit all packets dropped from the same window. Moreover we will propose a timeout back-off computation that takes current network conditions into consideration. All these actions will be used to achieve the two aims mentioned earlier.

In this chapter we will explain the proposed algorithms and we will do experiments to study their individual behaviour when transmission errors exist. In chapter 5 we will test their combined effect on TCP when transmission errors exist.

Finally, in chapter 6 we will add them to an error discriminator and study the behaviour when congestion and transmission errors coexist.

4.2 New TCP Reactions to Transmission Errors

In this section we will propose three algorithms to control TCP reaction to transmission errors. These algorithms should be implemented in the TCP sender and we propose that they should be used by TCP error discriminators in case of transmission drops.

4.2.1 Congestion Window Action

4.2.1.1 Motivation

Do we need to cut the congestion window for transmission (non-congestion) errors? And assuming we can discriminate between error types, what is the proper action TCP should take when there is a transmission drop?.

The trivial action when transmission drops occur is to resend the lost packet and avoid reducing the congestion window. This approach has been the base of most sender based end-to-end solutions like [1, 21, 52, 60, 64, 67].This is based on the following reasoning: a general formula to compute TCP throughput by using the round trip time and the window size is [3]:

$$Throughput_i = \frac{W_i}{RTT_i} \tag{4.1}$$

where W_i is the window size in round trip time RTT_i. So the average throughput

can be captured as following :

$$AvgThroughput = \frac{AvgW}{AvgRTT} \tag{4.2}$$

So if we can keep the average window size as large as possible during the transmission drops then the throughput should be higher than the conventional TCP (where the window size is cut with every drop) assuming we have fixed $AvgRTT$ (i.e. the $AvgRTT$ does not increase with the increase in congestion window). This conclusion has been drawn based on the assumption that $AvgRTT$ is independent of $AvgW$ [76]. However, we think that a trivial solution is not always the answer. Sometimes it is better to cut the congestion window even for transmission drops.

In high speed networks TCP requires a big window size in order to make use of the link available capacity. However, if a connection suffers from transmission errors, the link layer will be busy resending the corrupted packets and hence its goodput will decrease. At the same time TCP will keep the congestion window open because the errors are transmission errors. This will create more delay since the link layer will be forced to buffer the packets until they are retransmitted correctly or even to drop them if the buffer size is not enough or after a timeout. So in practice it is desirable to control the increase in the congestion window in case of transmission errors for the following reasons:

- Controlling the increase of the congestion window even for transmission errors can prevent undesirably large number of packet drops during transient congestion phases. If congestion happens while the congestion window is large, a large number of packets can be dropped which makes TCP enter a series of timeout events. These timeout events will force TCP to wait idle and also will increase exponentially with each successive timeout.

- Uncontrolled increase in the congestion window can lead to increase in the network load which will lead to increase in the RTT.
- Any increase in the RTT has mainly two side effects on TCP throughput:
  - The rate of increase in the RTT can be more than the rate of increase in the congestion window size and this will tend to cancel any gain in the throughput. see equation 4.2.
  - Another effect of the increase in the connection RTT is that it increases TCP retransmission timeout timer and hence increasing the period TCP should wait after errors. see equation 2.4.
- In many networks the link layer is responsible for buffering and retransmitting lost packets caused by link failure. However, if the end point sender keeps sending at high rates with no regard to transmission drops, the link layer will be forced to buffer large amounts of data or even drop some of the packets which will lead eventually to increasing the end-to-end RTT [56].

All these factors will result eventually in increasing the per-packet delay and hence increasing the average round trip time for the whole connection ($AvgRTT$). From that we can see that if the error discriminator does not cut the congestion window in the case of transmission drops the RTT may increase in a way that could cancel any benefit gained from increasing the congestion window size. For this reason the authors in [56] indicated that not cutting the congestion window for transmission drops is a bad policy.

Our proposal is to reduce TCP congestion window size *cwnd* by the number of dropped packets in the last window in the case of transmission errors in order to prevent increasing network load, and hence increasing connection $AvgRTT$. And because we cut the congestion window for both congestion and transmission errors this

will help to prevent increasing congestion in the case when the error discriminator wrongly diagnosed a congestion drop as a transmission drop.

Finally, any packet drop, even transmission drops, indicates that the link cannot handle this amount of extra packets at the moment. For this reason it is desirable to reduce the congestion window by at least an equal amount of packets.

4.2.1.2 The Algorithm

We will call the proposed algorithm the congestion window action (CWA) and it works as follows:

As we explained in chapter 2, in case of packet drops TCP cuts the congestion window after receiving three duplicate acknowledgments, see figure 4.1.

We propose to delay the cut decision until TCP receives all duplicate acknowledgments for the current window (i.e. the packets in flight during the drop) see figure 4.2. The duplicate acknowledgment usually indicates a packet drop but

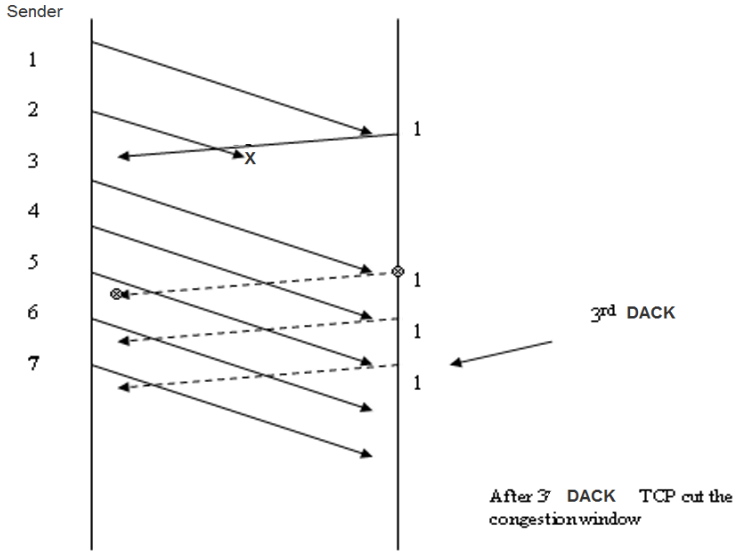

Figure 4.1: TCP duplicate acknowledgment action

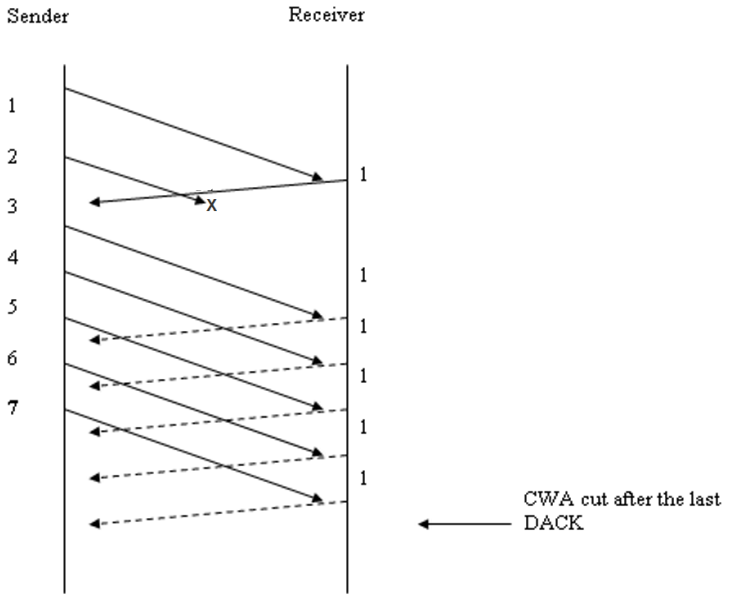

Figure 4.2: CWA duplicate acknowledgment action

also indicates that one packet has left the network (received by the other end). Using this information we can estimate how many packets were dropped per window ($droppedpackets = Windowsize - (No.ACKs + No.DACKs)$).

In order to make sure that we have received all duplicate acknowledgments TCP should send a new packet after receiving a number of duplicate acknowledgments and since this packet transmission happened after the previous window is sent its acknowledgment will be the last to be received so when TCP receives number of duplicate acknowledgments and then the acknowledgment for the closing packet it knows it has received all duplicate acknowledgments for the current window.

We call this packet the closing packet since it closes the previous window. Moreover, we can use the retransmission of the first lost packet as the closing packet and this way TCP can speed up the recovery process. Also for simplicity we assume the closing acknowledgment will follow the same path as previous acknowledgments.

So in the case of transmission drops, instead of cutting the congestion window to

half (as TCP) or not cutting it at all (as many error discriminators) we cut it only by the number of dropped packets. This way TCP cuts the congestion window in a rate related to the number of dropped packets. The benefit of this technique is that it improves the performance (especially for small error rates) and avoids increasing the congestion level at the same time by making TCP cut its sending rate even for transmission errors.

The algorithm is presented in figure 4.3. In the algorithm TCP checks if the current acknowledgment is a duplicate acknowledgment and if so it checks if it is the third duplicate acknowledgment in a row. It then saves the sequence number of last packet sent in *last_sent_3Dack* and resends the lost packet. Then when the receiver acknowledges the retransmitted packet TCP checks if it acknowledged all packets up to *last_sent_3Dack* and if it does not (i.e. $last\_sent\_3Dack > current\_ack$) then TCP computes the number of packets dropped and cuts the congestion window accordingly. However if the received acknowledgment is for all packets sent (i.e $last\_sent\_3Dack == current\_ack$) then TCP does nothing since there was only one drop and it was retransmitted and received safely. Moreover, if the retransmission failed and we have a timeout event then TCP cut the congestion window to one.

The CWA should be used in case of transmission errors only. However, if the error discriminator wrongly used CWA for congestion errors as well then the congestion in the network may increase. To solve this problem we will define another threshold we call it transmission drops threshold (*tthresh*). It will be used to record the congestion window size (*cwnd*) when the first drop occurs. It will define the area between the start of congestion avoidance phase (i.e. from *ssthresh*) and the first drop . Since this is the first drop then we call the *cwnd* size up to *tthresh* the safe area.

The information *tthresh* provides is that before this point there are no drops and so probably there is no congestion before this point and that after that point

```
1: Initialization: prev_ack = -1; last_sent_3Dack = -1
2: With every received acknowledgment Ack_i:
3: current_ack = Ack_i
4: if (current_ack == prev_ack) then                          ▷ Duplicate ack
5:     dackcount = dackcount+1
6:     if dackcount == 3 then                                 ▷ Packet drop
7:         last_sent_3Dack = P_max
8:         resend packet with seqNo = current_ack+1           ▷ No cut for cwnd
9:     end if
10: end if
11: if (current_ack > prev_ack) then                          ▷ no more DACKs
12:     prev_ack = current_ack
13:     if (last_sent_3Dack > current_ack) then               ▷ Some packets still not acknowledg
14:         compute number of drops and reduce cwnd:
15:         flight_size = last_sent_3Dack − current_ack
16:         num_drops = flight_size − dackcount
17:         cwnd = cwnd − num_drops
18:     end if
19: end if
20: if timeout==true then
21:     ssthresh = max(2,cwnd/2)
22:     cwnd = 1
23: end if
```

Variables:
current_ack: Sequence number of current acknowledgment.
prev_ack : Sequence number of previous acknowledgment (new acknowledgment only).
dackcount : Variable to keep track of how many duplicate acknowledgment TCP received so far. This variable is set to 0 whenever a new acknowledgment is received.
last_sent_3Dack: Variable to store sequence number of last sent packet after receiving 3 DACKs.
P_{max}: Last sent packet.
flight_size: Number of packets sent but not acknowledged yet.
num_drops: Number of packets dropped from this flight.
cwnd: Congestion window size.
ssthresh: Slow Start threshold.
RTO: Retransmission timeout timer. Calculated based in the average RTT.

Figure 4.3: CWA Pseudo code

drops occurred and hence there is chance of having congestion. Now when an error discriminator claims that the designated error is a transmission error, then before the error discriminator decides how to cut the congestion window CWA does another check (so it is two level check, one by the error discriminator and one by CWA) by comparing the congestion window when the drop is occur with the tthresh. If the congestion window is greater than tthresh then there is a higher chance that the error discriminator mismatch a congestion error for transmission error so the error discriminator then reacts in a conservative way by considering the drop as a congestion drop and cuts the congestion window to half (as normal TCP does). We do this because our aim is to increase the diagnosis accuracy of congestion errors as much as possible to avoid harming the network. However, if the error occurs while the congestion window is less than tthresh then it is safe to consider the error a transmission error.

However, although the use of *tthresh* heuristics can not guarantee improving the performance, it will prevent creating congestions.

The CWA algorithm with *tthresh* is presented in figure 4.4. As we can see in the algorithm 4.4, after the first drop the value of *cwnd* is saved in *tthresh*. Later when another drop occurs the error discriminator will check if $cwnd \leq tthresh$ and if so the drop is probably a transmission drop. Otherwise the drop is assumed to be a congestion drop. Also, TCP should recalculate *tthresh* after each timeout event because TCP will initialize the congestion window and will start building a new window. This is done in the algorithm by using *first_drop* which will be set to one after each timeout and hence allowing *tthresh* to take a new *cwnd*.

Note that timeout is added to the above algorithms just to show what happens in case of a timeout, however normally timeout will be in a separate function and will be called only when the timer expires. Also note that in case of transmission errors

```
1: Initialization: prev_ack = -1; last_sent_3Dack = -1; first_drop = 1
2: With every received acknowledgment Ack_i:
3: current_ack = Ack_i
4: if (current_ack == prev_ack) then                          ▷ Duplicate ack
5:     dackcount = dackcount+1
6:     if dackcount == 3 then                                  ▷ Packet drop
7:         last_sent_3Dack = P_max
8:         resend packet with seqNo = current_ack+1            ▷ No cut for cwnd
9:         if first_drop then
10:            tthresh = cwnd
11:            first_drop = 0
12:        end if
13:    end if
14: end if
15: if (current_ack > prev_ack) then                           ▷ no more DACKs
16:    prev_ack = current_ack
17:    if (last_sent_3Dack > current_ack) then                 ▷ Some packets still not acknowledg
18:        compute number of drops and reduce cwnd:
19:        flight_size = last_sent_3Dack − current_ack
20:        num_drops = flight_size − dackcount
21:        if cwnd < tthresh then
22:            cwnd = cwnd − num_drops
23:        else
24:            cwnd = cwnd / 2
25:            ssthresh = cwnd
26:        end if
27:    end if
28: end if
29: if timeout==true then
30:    ssthresh = max(2,cwnd/2)
31:    cwnd = 1
32:    first_drop = 1                    ▷ Initialize first_drop after each timeout
33: end if
```

Variables:
first_drop: Flag to indicate that first drop in this connection has occurred.

Figure 4.4: CWA+tthresh Pseudo code

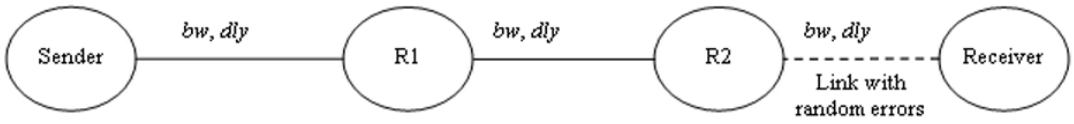

Figure 4.5: Simple Network Topology

we do not change *ssthresh* and only change the *cwnd* since the drop is not congestion error and probably the link capacity (indicated by *ssthreh*) has not changed.

In the rest of this thesis we will use the final version of the algorithm (CWA+tthresh) and we will call it CWA for simplicity.

Finally, one important aim of CWA is to increase TCP congestion window size by reducing congestion window cut rate in case of transmission errors. However, TCP only recovers the first dropped packet and leaves the rest to be recovered by timeouts as we explained in section 2.4.5. CWA will be affected negatively in this case because TCP resets the congestion window size to one segment after each timeout so any cut by CWA will be canceled. For this reason, later, we will propose an algorithm to recover multiple packet drops per window of data which aims to reduce the effect of timeout events on CWA.

4.2.1.3 CWA Performance

An important aim of CWA is to improve TCP performance by keeping a bigger congestion window in case of transmission errors. In order to measure the improvement we will measure the average congestion window size during the connection life time.

We assume only transmission errors are present in the connection so we used a simple topology presented in figure 4.5. Similar topologies are used by other authors like [77] to test TCP modifications in presence of transmission errors only . Each link bandwidth *bw* is fixed to 45Mbps (T3 link). The total link propagation delay

is 12ms so each link delay *dly* takes 2ms. In chapter 6 we will test the final system with different bandwidth and delay values.

Transmission errors are generated in the last link using a two state model to simulate error and error-free phases and error rates range from 0.001 (0.1%) to 0.1 (10%) with increase of 1% each time. The same error range is used for all experiments in this chapter except for RTA where we increased error rate up to 20% to show the improvement under heavy errors as we will see later. In all experiments we repeat the experiment a sufficient number of times each with different seed for the random number generator. The 95% confidence intervals are very small and not visible in the graphs (we repeat the experiment until the upper and lower limit interval is no more than 5% from the average value) so we draw the average value only in the graphs.

We run the experiment to measure the TCP congestion window and then we add CWA to TCP (to replace the standard TCP reaction to errors which is to cut the congestion window to half) and repeat the experiment with the modified TCP. The chart in figure 4.6 shows that after adding CWA, TCP gained a higher average congestion window. This is achieved by CWA preventing unnecessary congestion window cuts and limiting the cuts to the number of lost packets in case of transmission drops. Figure 4.6 uses log-linear scale because the values of the congestion window size takes a large range so we use log-linear (or semi-log) scale to make it easy to see low as well as high values in the y axis.

The increase in congestion window size increases TCP sending rate. Also it will reduce the chance to have timeouts because with a bigger window TCP gets more duplicate acknowledgments after drops. These duplicate acknowledgments will trigger lost packet retransmission and will increase the congestion window during the fast recovery phase.

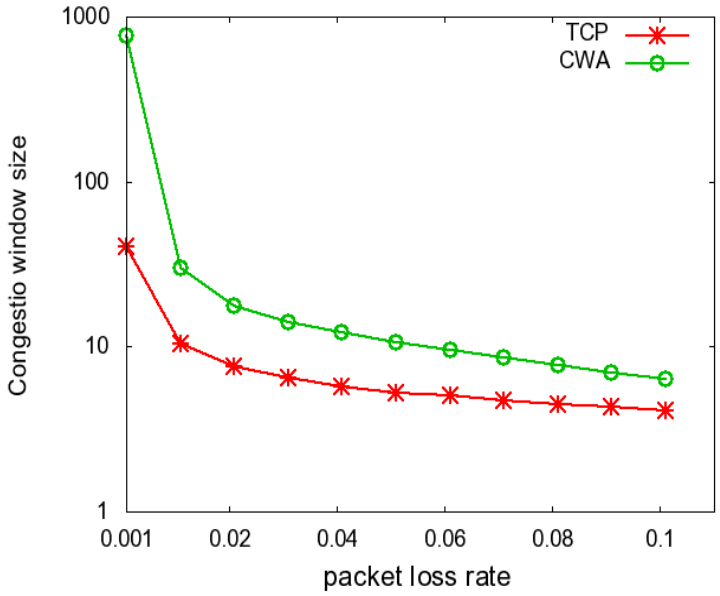

Figure 4.6: TCP vs. CWA. semi-log scale congestion window size (packets)

However, due to the fact that the increase in the error rate will increase timeout durations as we will see later when we test the RTA algorithm, the congestion window will not have a chance to grow after a timeout event because TCP will wait idle for longer times. Moreover, with the increase in error rate many packets will be dropped more than once and more longer retransmission timeouts will occur due to multiple packet drops. This is another problem that decreases the performance of CWA which is multiple drops per window of data. Since TCP resends only one dropped packet per window the rest will be recovered through timeouts. This will increase the number of timeout events which will affect the performance of CWA negatively. In the following section we will solve this problem by using a multiple drops action algorithm MDA.

Limitations: *tthresh* is proposed as a second line of defence against creating unnecessary congestion caused by error discriminators which may diagnoses congestion errors as transmission errors and hence not responding to congestion. *tthresh*

tries to prevent this by monitoring drops and recording the congestion window size (*cwnd*) when first drop occurs and consider it as an indication that this is where congestions happens. So if later another drop occurs while the congestion window is above this size then it is probably a congestion drop.

However, this method may reduce the TCP performance gain from using an error discriminator if transmission errors occurred while $cwnd > tthresh$. This is because if $cwnd > tthresh$ the CWA uses standard TCP action by cutting the congestion window to half.

However, using *tthresh* protects the network from unnecessary congestion caused by error discriminators mistakes. Moreover, in the worst case scenario if all errors where transmission errors and at the same time occurred while $cwnd > tthresh$ and hence treated by CWA as congestion errors, the performance will be at worst as standard TCP (i.e. cut congestion window to half for each drop).

Moreover, we recalculate *tthresh* after each timeout event because timeout indicates that *tthresh* value is probably not good to prevent creating congestion since a timeout usually indicates severe congestion.

Also, since CWA only cuts the *cwnd* and not *ssthresh* this will help recovering *cwnd* quickly because *cwnd* is increased exponentially when it is less than *ssthresh* (as we explained in chapter 2 under Slow Start). This faster recovery will help to balance the performance cut that is caused by *tthresh*.

4.2.2 Recovering From Multiple Drops

4.2.2.1 Motivation

TCP sends only the first packet dropped from a window of data and leaves the rest to be recovered by the retransmission timeout mechanism after a waiting period of

time. However, multiple congestion drops can occur because of bursty behaviour of TCP (exponential growth during slow start for example) or because of severe congestion.

When more than one packet is dropped from the same window TCP resends the first one dropped and then stops sending packets and remains idle until a time-out occurs. This timeout will trigger resending the rest of the dropped packets. However, this costs TCP valuable time in two ways: first the time it remains idle without sending any data until the timeout occurs and second after any retransmission timeout TCP reduces its congestion window to the minimum (usually one segment).

To solve this problem for congestion drops, authors in [37] propose a change to the fast retransmission mechanism in TCP in order to make TCP resend all packets dropped from the same widow by resending one packet each round trip time; they call it TCP-NewReno [37]. This method will allow TCP to stay in retransmission mode until all lost packets are retransmitted. However, the problem here is that TCP only resends one packet per RTT because it resends a packet for each partial acknowledgment it receives (a partial acknowledgment is a new acknowledgment that does not acknowledge all outstanding packets). So if N packets are dropped from a window TCP will need N RTTs to recover that window.

Another solution proposed by [77] for wireless networks suffering transmission errors is to resend all non acknowledged data after TCP receives the first partial acknowledgment; they call it TCP Bulk Repeat [77]. This is based on the assumption that if more than one packet is dropped from the same window then probably there are more packets dropped also and hence it is better to resend the whole window since we expect that other drops have happened. However, although this solution will cause TCP to recover more quickly in heavy loss networks, it will increase the

resending rate largely in networks with light/medium losses, especially when using large TCP windows. For example if the window size is 200 packets and only the first two packets were dropped then TCP Bulk Repeat will resend the rest of the window (198 packets) unnecessarily.

4.2.2.2 The Algorithm

Wireless drops usually happen in bursts [3,26], a set of consecutive packets dropped at once and, since an important source of non-congestion (transmission) drops in today's networks is wireless links, in our design we assume that most transmission drops happen in bursts. Also, other authors like [77, 78] use the same assumption (i.e. wireless errors occur in bursts) in their work.

Using similar concepts as in CWA, in this algorithm TCP will use the set of duplicate acknowledgments which has been received after the first drop to estimate the number of dropped packets per window and, assuming this number represents consecutive dropped packets, we resend that number of packets starting from the first packet dropped. As we did in the CWA algorithm, TCP will compute the number of dropped packets by subtracting the number of duplicate acknowledgments from the number of actual packets sent from that window. This will give us the number of packets dropped from this window.

So after the first partial acknowledgment we do not send one packet like NewReno [37] or the whole window like TCP Bulk Repeat [77]; instead we send only the number of drops we calculated. This way if the packets dropped were consecutive then we might recover the whole window in one RTT with no unnecessary retransmissions. However, if there are more dropped packets in this window or the dropped packets were scattered then we will propose another retransmission mechanism which can send them one by one as NewReno [37] or resend the rest of the window as Bulk

Repeat [77] based on a simplified estimation of the network error rate. Figure 4.7 shows the TCP reaction to a burst of drops and Figure 4.8 shows the proposed idea. As we can see from the figures the proposed idea will recover the window in less time than both TCP and NewReno [37] and with less unnecessary retransmissions than Bulk Repeat [77] providing that there is one burst of errors per window.

The algorithm is presented in figure 4.9.

This proposed algorithm is to be used by the error discriminator to recover from multiple transmission drops per window. The main benefit is that it increases TCP performance by increasing the resending rate for lost packets. It allows TCP resend all packets lost from the same window and this will reduce timeouts especially when a burst of packets are dropped on the lossy link. Reducing the number of timeout events is important to improve TCP performance since each timeout event will reset the congestion window to the minimum and will make TCP to wait idle. However, in some cases the transmission errors are persistent as in the case of long link disconnection, and even the retransmitted packets are dropped so there is no point to keep retransmitting the lost packet so we need to wait (stop sending) for a period of time to allow the connection to startup again. For this reason the algorithm retransmits the lost packets only once per window and then it allows retransmission timeout to occur between windows if the errors are persistent. The variable *first_dup* in the algorithm is used to allow resending the packets only once per window.

Although the algorithm is directed to transmission errors, we believe it can also give good results with congestion drops so in chapter 6 we will test both cases.

4.2.2.3 Nonconsecutive Drops

If drops are not consecutive then MDA resends one packet per RTT (as in NewReno [37]). This is a conservative approach but it will guarantee there is no

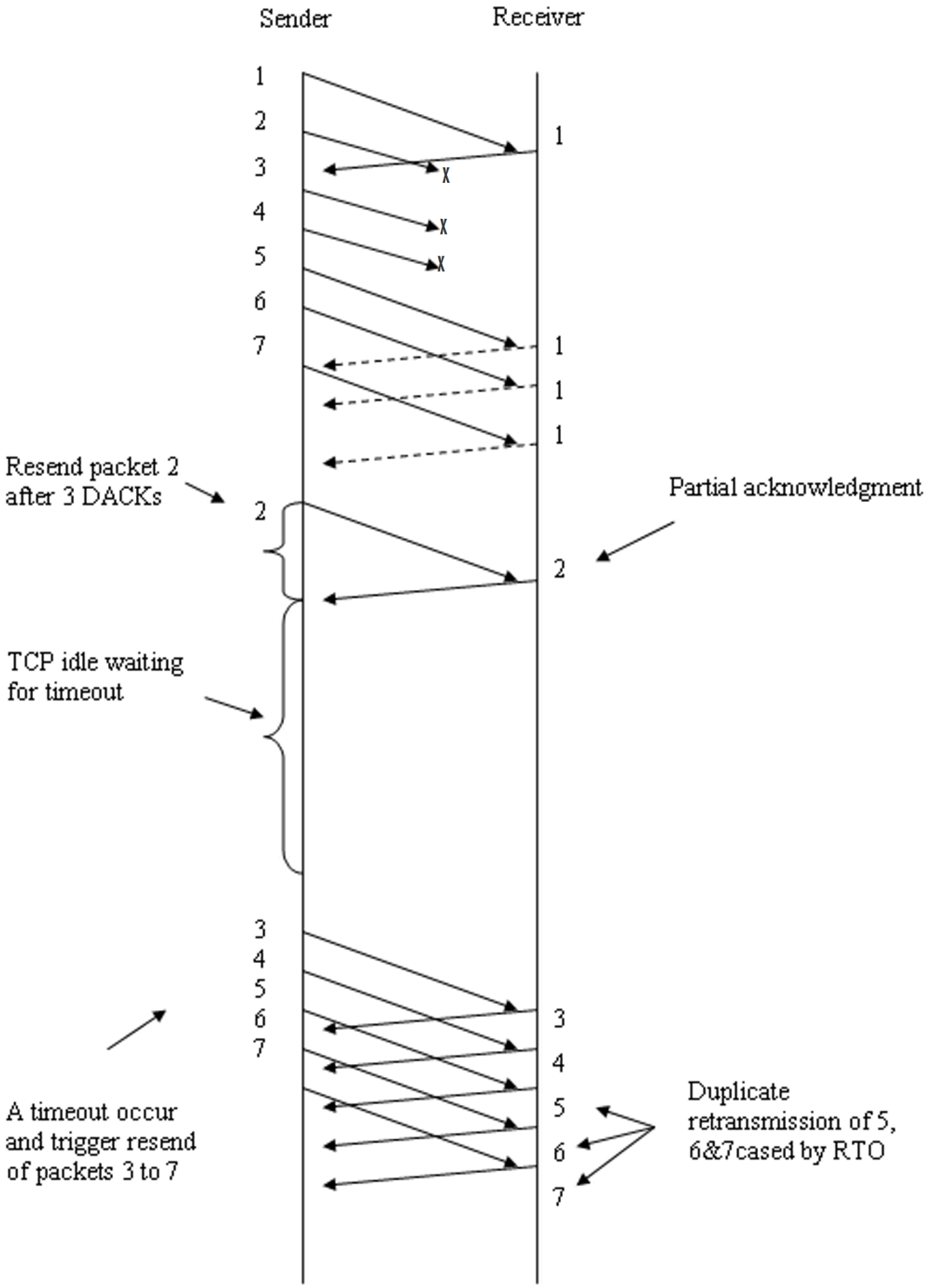

Figure 4.7: TCP and burst of drops

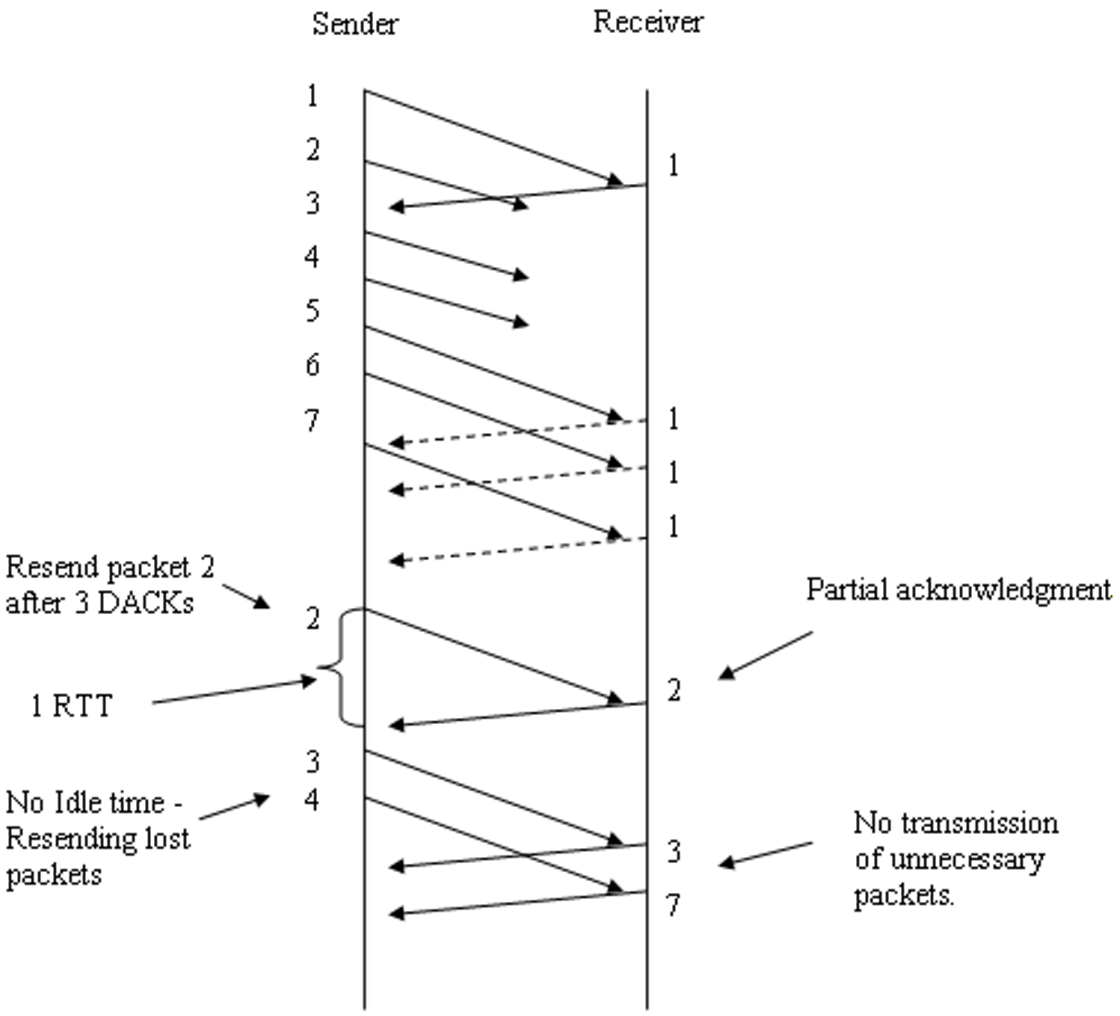

Figure 4.8: Proposed multi drop retransmission

```
1: Initialization: prev_ack = -1; last_sent_3Dack = -1; first_burst = 1
2: With every received acknowledgment Ack_i:
3: current_ack = Ack_i
4: if (current_ack == prev_ack) then                          ▷ Duplicate ack
5:     dackcount = dackcount+1
6:     if (dackcount == 3 && first_burst) then                ▷ Packet drop
7:         last_sent_3Dack = P_max
8:         resend packet with seqNo = current_ack+1           ▷ No cut for cwnd
9:     end if
10: end if
11: if (current_ack > prev_ack) then                          ▷ no more DACKs
12:     prev_ack = current_ack
13:     if ((last_sent_3Dack > current_ack) && first_burst) then
14:         compute number of drops:
15:         flight_size = last_sent_3Dack − current_ack
16:         num_drops = flight_size − dackcount
17:         for i=1;i≤(num_drops);i++ do
18:             resend prev_ack + i
19:         end for
20:         first_burst = 0                                   ▷ One retransmission per window
21:     else
22:         if (last_sent_3Dack ≤ current_ack) then
23:             first_burst = 1
24:         end if
25:     end if
26: end if
27: if timeout==true then
28:     ssthresh = max(2,cwnd/2)
29:     cwnd = 1
30:     last_sent_3Dack = -1
31:     first_burst = 1
32: end if
```

Variables:
first_burst: Flag to indicate if this is the first burst of drops in this window or not

Figure 4.9: MDA Pseudo code

unnecessary retransmission. In appendix B we present an idea to resend lost packets either one per RTT or as a bulk based on error rate estimation per window. Because we did not test this idea in our simulation we did not include it in the main text.

4.2.2.4 MDA Performance

In order to measure the improvement using MDA we will measure the number of timeout events during the connection lifetime for TCP before and after adding MDA since the MDA action is concerned about reducing the number of timeouts TCP requires by trying to resend all dropped packets from the same window. We use the same experiment settings as in CWA performance evaluation (section 4.2.1.3).

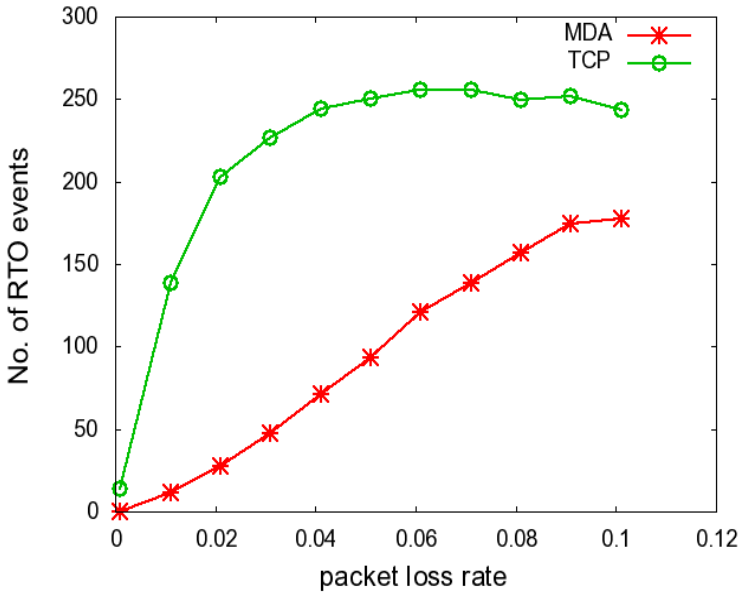

Figure 4.10: TCP vs. MDA. No.RTO

Figure 4.10 shows number of timeouts for TCP and MDA. As we can see the number of timeouts in case of MDA is much less than in TCP. However, even when using MDA, timeouts are sometimes expected to occur especially with higher error rates where the same packet may be dropped more than once and as we explained

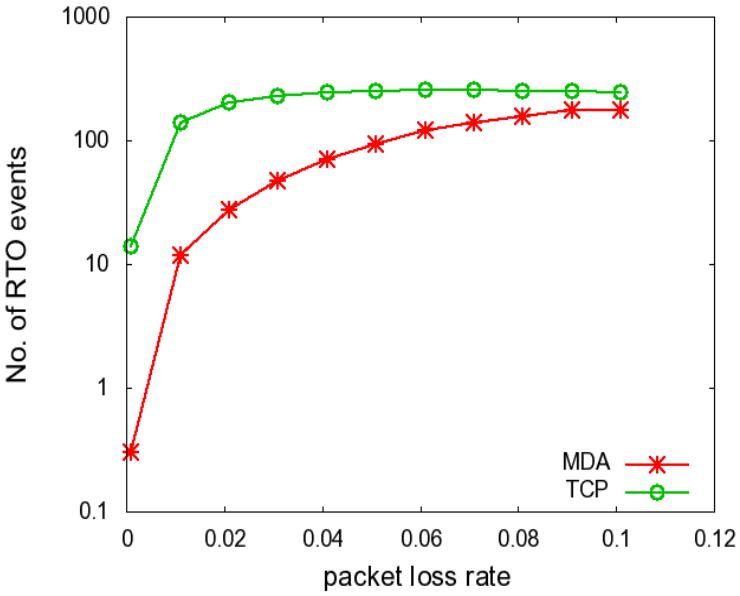

Figure 4.11: Semi-log scale TCP vs. MDA. No.RTO

before MDA resends the packet only once per window to prevent unnecessary retransmission in case the connection will be dead for long time.

When we draw the logarithmic scale on the number of RTO we can see that the two curves take the same shape as we can see in figure 4.11. This indicates that both TCP and MDA suffer from the effect of multiplicative congestion window cuts used in TCP (i.e. cutting congestion window to half after each drop) which will result in smaller congestion window size and hence less acknowledgments which leads to more timeouts. We solve this problem in the CWA action described in the previous section. This shows us that CWA and MDA are complimentary to each other and for that when we advise an action for the error discriminator we will combine them together along with the RTA action.

Note that in figure 4.11 number of events for MDA start with value less than one (0.3), this is because as we explained before that we run each experiment multiple times and then we take the average of all runs so in this lower error rate the MDA

was so effective in some runs the number of RTO was 0 and some of them was 1, so the average was 0.3.

Moreover, we can notice that in figure 4.10 with the increase in the error rate the number of timeouts start to settle around 250 (i.e. no big increase) in case of TCP. The is natural because with the increase in the error rate the connection dries quickly due to large number of dropped packets so the number of sent packets decreases and hence number of timeouts does not increase too much.

Finally, the fact that even when TCP used MDA the number of timeout increases shows that MDA is not able to completely hide the effect of multiple packet drops from TCP leading TCP to fall into timeouts. However, MDA is certainly able to reduce that effect on TCP especially under lower error rates which leads to the reduction in timeout events we saw in figures 4.10 and 4.11.

4.2.3 Improving RTO back-off for Transmission Drops

4.2.3.1 Motivation

When a series of timeout events occur in sequence TCP increases its retransmission timeout (RTO) exponentially (i.e. increases waiting time after drops). This increase is called retransmission timeout back-off and it is an estimation of the time needed for the network to empty its buffers after congestions [10,36]. The reason for choosing exponential increase is to make the TCP sender more careful while the congestion still exists [29]. However, if there is no congestion and the drops are transmission drops this exponential back-off could lead to unnecessarily long periods of inactivity.

For this reason previous works like [73] [79] have proposed to give RTO a fixed value in case of transmission drops and to ignore the TCP back-off policy.

However, unchanging RTO can lead to unwanted congestion caused by TCP

ignoring the signs of a severe congestion (i.e. multiple timeouts) especially if the error discriminator used is of low accuracy. Moreover, choosing what RTO value to use will not be easy and will depend on how accurate RTO estimation was before the first dropped packet.

The authors in [80] suggested that TCP should not rely on exponential back-off in case of heavy drops and should use an estimation of the network available bandwidth to compute RTO. In our work we will use the same idea but with different implementation as we will explain later. Moreover, we will propose a method to integrate the new mechanism into TCP by implementing it in an error discriminator as we will see later in chapter 6.

Our proposal is that in the case of transmission errors the TCP error discriminator should increase waiting time (i.e. to back-off) with each timeout but in a way that considers the network status by estimating the network available bandwidth and to compute the back-off level based on that estimation. The resulting back-off level will oscillate between exponential back-off and fixed value back-off based on the estimated bandwidth.

The following section explains the proposed algorithm.

4.2.3.2 The Algorithm

After each timeout event TCP increases RTO (backs-off) as follows [30]:

$$\text{RTO} = \text{RTO} * 2 \tag{4.3}$$

So after n timeout events we can calculate RTO as following:

$$\text{RTO} = \text{RTO} * 2^n \tag{4.4}$$

where n also represents the number of failed retransmission attempts so far (i.e. no *new acknowledgments* received).

In our work instead of doubling RTO with each failed retransmission we first estimate the network available bandwidth and then instead of using n in equation 4.3 as the power of 2 we use a function *f(n)* which reflects the available bandwidth just before the first timeout occurred. So after n timeout events the new back-off policy will be computed as following:

$$\text{RTO} = \text{RTO} * 2^{f(n)} \tag{4.5}$$

To calculate *f(n)* we will first estimate the available bandwidth. Following [53] we estimate the available bandwidth *bw* by calculating the rate of received acknowledgments where each acknowledgment represents one segment size that has been delivered successfully (this is true even for duplicate acknowledgments). see equation 4.6.

$$\text{bw} = \frac{\text{segment\_size}}{T_i - T_{(i-1)}} \tag{4.6}$$

Where T_i is the time of receiving Ack_i and $T_{(i-1)}$ is the time of receiving $Ack_{(i-1)}$. Using this equation we can estimate the available bandwidth *bw* as rate of packets that can pass the network bottleneck successfully during a time equals to $T_i - T_{(i-1)}$. The increase in this rate means the increase in the available bandwidth and vice versa. The authors in [53] have tested the effectiveness of this bandwidth estimator and they showed that it responds well to the increase and decrease in the available bandwidth. Also authors like [31] [71] have presented a similar method to estimate the network capacity.

We then calculate a weighted average of the available bandwidth readings *bw* in

order to filter sudden changes and we call it *avalb_bw*:

$$\text{avalb\_bw} = \beta \times \text{avalb\_bw} + (1 - \beta) \times \text{bw} \tag{4.7}$$

However, since the estimation of the bandwidth is very related to the estimation of the network round trip time (RTT) we will use vales between 0.8 and 0.9 for β which are the same values used in TCP to compute RTT (see section 2.4.4). The final value that have been used for β was tuned during simulation.

Now using the readings from the bandwidth estimator we compute *f(n)* as following:

$$f(n) = n * (1 - \frac{avalb\_bw}{max\_avalb\_bw}) \tag{4.8}$$

where *max_avalb_bw* is the maximum value of the available bandwidth seen so far.

The algorithm to calculate available bandwidth is presented in figure 4.12

```
1: Initialization: β = 0.9; current_time =0; max_avalb_bw =0
2: With every acknowledgment Ack_i:
3: prev_time = current_time                          ▷ T_(i−1) = T_i
4: current_time = get current time
5: bw = segment_size / (current_time - prev_time)
6: avalb_bw = β * avalb_bw + (1-β) * bw
7: if avalb_bw > max_avalb_bw then
8:     max_avalb_bw = avalb_bw
9: end if
```

Figure 4.12: Available bandwidth estimation

This algorithm should be called only after the TCP sender receives at least two acknowledgments in order to be able to calculate $T_{(i-1)} = T_i$. Also after calculating the first *bw* value we should set avalb_bw = bw.

The algorithm to calculate RTO is presented in Figure 4.13.

1: Initialization: $F_n = 0$; ORTO = CRTO
2: After each timeout:
3: n = n+1
4: F_n = n * (1- (avalb_bw/max_avalb_bw))
5: CRTO = ORTO * 2^{F_n}

Variables:
n: Number of timeout events so far. Note TCP should initialize n to 0 when it receives a new acknowledgment.
ORTO: Original timeout: timeout calculated based on RTT readings and without back-off as in equation 2.4.
CRTO: Current timeout: timeout with back-off.

Figure 4.13: RTO back-off algorithm

The idea is that in case of transmission errors, the ratio *avalb_bw*/*max_avalb_bw* will be used to estimate the back-off level of the error discriminator retransmission timeout. So if there is no congestion then the *avalb_bw* will be very close to the *max_avalb_bw*. As we can see in the algorithm in figure 4.13, this will result in smaller F_n and hence a smaller retransmission timeout value (*CRTO*).

Now when the error discriminator decided that the error is a transmission error then instead of using a long retransmission timeout as in TCP it uses a timeout calculated based on the available bandwidth which make it more related to current network conditions.

This will help TCP if there is no congestion and there are only transmission drops since it will decrease waiting periods during timeout events and hence will increase the retransmission rate which will give packets more chance to be delivered over the lossy link. Also reducing the waiting time in case of non-congestion errors will allow TCP to send the same amount of data in shorter periods.

Moreover, since the congestion window after a timeout is reduced to one segment the increase in retransmission rate will not increase the risk of congestion on the network since the sender window is restricted to one packet only. This will make the use of an error discriminator possible even with low discrimination accuracy since TCP will start with a small congestion window after each RTO so even if the error discriminator mismatches a congestion error for a transmission error the impact on the network will be minimal.

Moreover, in case of congestion the estimated available bandwidth will decrease and this will increase the value of F_n in the algorithm. So even if the error discriminator mismatched congestion error for transmission error, the back-off will be high and near to standard TCP (i.e exponential increase in the RTO).

Finally, we want to note that our work to use bandwidth estimation to compute the back-off value is based on the work presented in [80]. However an important difference between our method and the one in [80] is that it uses an additive increase of the RTO instead of multiplicative increase as we used (i.e. it uses + instead of * in equation 4.3). However, although using additive increase decreases the length of RTO during heavy transmission losses, we do not think it is a good choice when both congestion and transmission errors coexist because this low rare of increase in RTO length will make the error discriminator to cause congestion in case of error mismatch since the increase in the RTO will be small and the network will not have time to flush the congestion. However, in our algorithm, using multiplicative increase of RTO length will prevent this, as well as make minimum changes to TCP implementation since TCP uses multiplicative increase to compute the back-off level. Another difference is that in our implementation we calculate maximum available bandwidth dynamically while in [80] the maximum bandwidth is limited to 128 KBps. Also, in our work we propose a practical method to integrate the

new mechanism into TCP by implementing it in an error discriminator as we will see later in chapter 6 which will allow using it in low and high error rates and also in presence of congestion errors. However, the scheme in [80] is proposed to work under heavy errors only (special case).

Notes and issues:

- We call this algorithm the Retransmission Timeout Action (RTA).
- In the algorithm in figure 4.12 we recalculate the available bandwidth every time TCP receives an acknowledgment. Typically the estimation should be calculated only when TCP receives new acknowledgments since using duplicate acknowledgments may result in lower estimation of the bandwidth since the time between the last new acknowledgment and the first duplicate acknowledgment is more than the time between two new acknowledgments. However, we think that using the wighted average to calculate the available bandwidth and the fact that consecutive duplicate acknowledgments after a drop will probably have the same arrival rate as consecutive new acknowledgments; so we decided to use new as well as duplicate acknowledgments in our algorithm.

4.2.3.3 RTA Performance

Here we will compare the total time TCP stays idle because of RTO before and after adding the RTA. We use the same experiment settings as in CWA performance evaluation (section 4.2.1.3) except that we will increase the error rate range up to 20% because with higher error rates TCP will have longer timeout events because packets are dropped more than once.

The results are presented in figure 4.14. As we can see, when we use RTA there is a noticeable decrease in the total length of RTO events especially with the

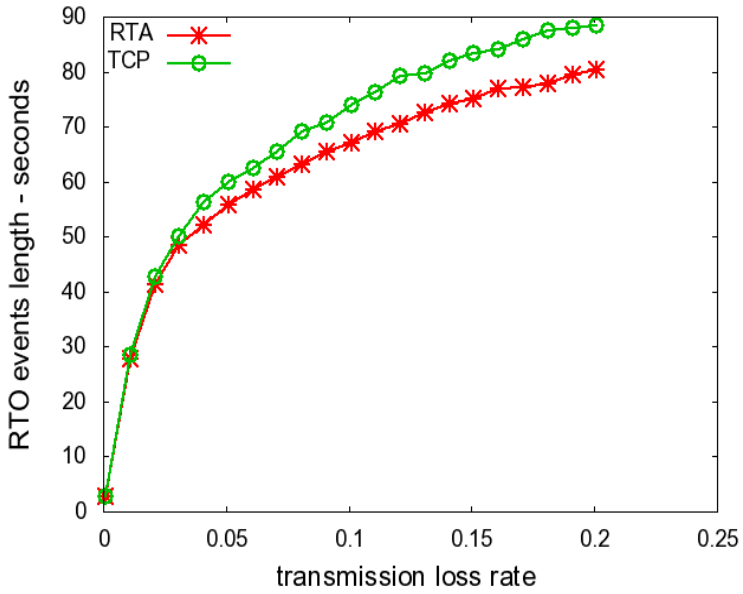

Figure 4.14: RTO length

increase of the error rate (the improvement reaches 10% over 20% error rates). This is due to the fact that back-off policy is most important when multiple drops of the same packet occur which leads to multiple failed retransmission events. Each failed retransmission increases the TCP waiting time exponentially. Since multiple packet drop events increases with the increase in the error rate we can see how RTA reduces the effect of multiple drops on TCP since the back-off (waiting time) is no longer exponentially increasing with each drop but instead it depends more on the actual network condition and estimated bandwidth. Figure 4.15 shows a snapshot of TCP back-off and RTA back-off compared to the estimated bandwidth under heavy transmission drop rate(20%). As we can see TCP back-off responds to the high drop rate by increasing the back-off level to the highest value 64 regardless of the network conditions. However, in case of RTA the back-off takes a value related to the available bandwidth (bandwidth measured in Mbps) which changes according to the changes in the network available bandwidth (i.e. it decrease with the increase

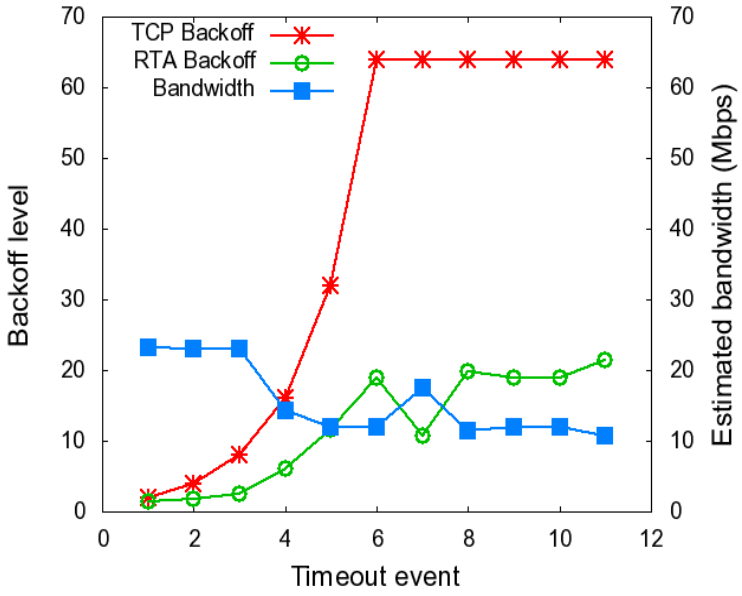

Figure 4.15: Available bandwidth and backoff level

in the available bandwidth and vise versa).

However, under low error rates there is no improvement as we can see in figure 4.14 because under these low error rates we have less timeout events and it is harder to have multiple drops of the same packet and hence TCP does not increase RTO. However the algorithm still able to reduce RTO length and hence TCP idle time with the increase of the error rate.

Limitations: The limitations we present here are related to the bandwidth estimator proposed in [53] which we used in RTA. If the acknowledgments are dropped the bandwidth estimation will not work probably. However, this problem came originally from TCP design. TCP uses acknowledgments to compute RTT and RTO. If the acknowledgments are dropped TCP will not compute RTO properly and since our algorithm will be added to TCP it will suffer the same problem.

However, the assumption that acknowledgments will be delivered in most cases is supported by the fact that acknowledgments are usually very small packets compared

to normal TCP packets (usually 4% of the normal TCP segment size) so there is less chance to have them dropped because of transmission or congestion errors. Also, in the case of acknowledgments drop, the RTA estimation of the bandwidth will decrease so the performance will be similar to TCP and this will guarantee no harm on the network since TCP backs-off exponentially.

4.3 Transmission Window Action (TWA)

In the previous sections we proposed three algorithms, CWA, MDA and RTA. We will call these algorithms in combination the transmission window actions TWA. Although the functionality of each algorithm is independent, these algorithms were designed to form a transmission action for an error discriminator to help it to react to transmission errors.

Figures 4.16 shows general design for error discriminator and figure 4.17 shows

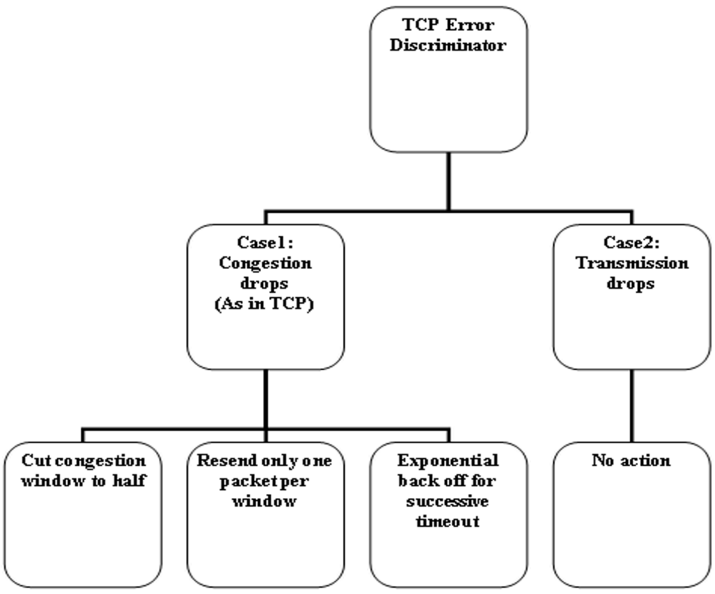

Figure 4.16: Error discriminator general functionality

the changes after adding TWA.

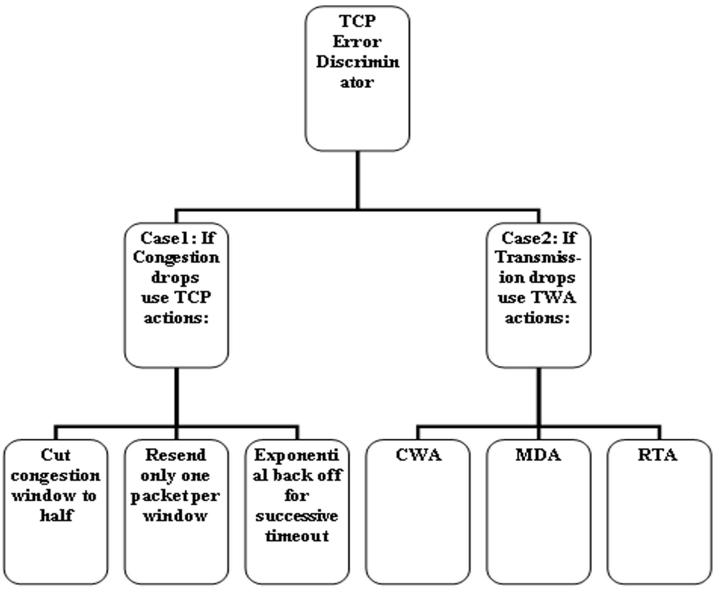

Figure 4.17: Error discriminator after adding TWA

It is of our interest to see CWA, MDA and RTA work together before adding them to the error discriminator. So, in the next chapter we will test the performance of TWA to see how it improves the performance in case of transmission errors only. In chapter 6 we will add TWA to an error discriminator and test the performance in presence of congestion and transmission errors.

4.4 Summary

The aim of any end-to-end TCP error discriminator is first to improve TCP performance over lossy link and second to avoid causing any harm to the network by not increasing the level of congestion. In our work we realize the need to have a proper transmission window action (action in case of transmission errors) that will help an end-to-end error discriminator to reach the above aims. We proposed three algorithms: Congestion window action (CWA), Multiple drops action (MDA) and Retransmission timeout action (RTA) and we call them the transmission window

actions (TWA) and they will be used by error discriminators in case of transmission errors.

The CWA computes how many packets were dropped in a single window by subtracting number of duplicate acknowledgments from the window size. Then it reduces the window size according to number of dropped packets only.

The MDA works by resending only the number of packets equal to number of packets dropped. This will help to recover the TCP window in one RTT when dropped packets are in sequence (burst of drops).

The RTA algorithm estimates the available bandwidth and uses this estimate to decide the RTO back-off level instead of using exponential back-off as in TCP.

Each proposed algorithm helps to achieve the above two goals as follows:

- Goal:Increasing TCP throughput over lossy links:
  - CWA algorithm increases the average congestion window size for transmission errors because it does not cut the congestion window to half of its original size after each error but makes the cut based on number of dropped packets.
  - MDA algorithm reduces number of RTO events by resending all lost packets in the same window.
  - RTA algorithm reduces the length of RTO events for transmission errors by estimating the available bandwidth so if there is no congestion or the congestion is low then the back-off level will not increase exponentially as in the case of TCP.
- Goal: Avoid increasing congestion in the network:
  - CWA algorithm allows TCP to reduce the congestion window even for

transmission errors which reduces the effect of error discriminator mistakes between congestion and transmission errors.

- MDA algorithm resends lost packets only once per RTT and allows RTO to occur between windows which will allow TCP to slow down if there is severe congestion or the transmission errors are persistent.
- RTA algorithm allows TCP to increase RTO based on the estimated available bandwidth which will increase the RTO with the decrease in the available bandwidth because of congestion.

These actions will help any end-to-end error discriminators to improve TCP performance in case of transmission errors as we saw in this chapter. Also they will help to prevent causing unnecessary congestion in the network as we will see later in chapter 6.

Chapter 5

Simulation Model and Performance Evaluation of TWA

5.1 Introduction

In this chapter we will explain the simulation model we used in our work. Then we will do an evaluation of TWA to show the improvement it adds to TCP performance in presence of transmission errors only. Also we will test each component of TWA namly CWA, MDA and RTA to show the role of each one in improving the performance.

In chapter 6 we will add TWA to an error discriminator where the tests will be done in presence of congestion and transmission drops.

5.2 Simulation Model

In this section we will try to answer questions like: Why we use simulation to test our proposal? and what simulation software we are using to build our simulation

and why?

After that, we will explain the topology and simulation settings used during our experiments.

Following we will start by explaining why we used simulation in our work.

5.2.1 Rationale Behind Simulation: Creating a Controlled Environment

It is important to concentrate on the variables that have a direct relation to our study and to eliminate or minimize the effect of other variables which have no direct relation.

Achieving this goal is not easy in real life, especially in systems like computer networks which usually contain many elements. For example, link bandwidth, routers processing power and buffering capacity are variables that may change from one component to another and from time to time in the network and are hard to control in a real system.

Because of that we need to create a controlled environment in which we can control all the variables that may affect our experiment, simulation is one solution. We can use simulation to create a simplified model of the network where we can exclude unwanted variables by fixing them and allow only related variables to affect the results. Also sometimes simulation is the only option to test new protocols especially if the required change in the network is of wide scale [29].

However, simulation is not as good as real life experimentation because in simulation we exclude many variables that may be present in the real life. Moreover, the quality of the simulator itself and how well it is programmed may also affect the experimental results.

Considering previous points we chose to base our simulator on ns2 [81] which is one of the widely used simulators in computer networks research. Many studies have used ns2 in evaluating new and existing protocols, for examples see [5, 37, 56, 82, 83].

Moreover, since my work is strongly related to TCP, it is interesting to know that the TCP implementation in ns2 has been developed by the Computer Systems and Engineering Group at the University of California[1] who implemented some of the first and popular versions of TCP like TCP-Tahoe in UNIX 4.3BSD-Tahoe and TCP-Reno in UNIX 4.3BSD-Reno [20, 84] and it is a direct implementation of TCP specification as explained in [20, 24]. This has added more motivations for using ns2 as a base for the simulation model.

5.2.2 Topology and Simulation Settings

We use a single bottleneck dumb-bell network topology commonly used in TCP evaluation [85] and presented in figure 5.1. In this topology we have two kinds of sources the TCP sources (TS) and UDP sources sources (US). In all experiments the first TCP source (TS1) will apply the modified protocol we want to test.

Destinations for TCP sources are named TD and destinations for UDP sources are named UD. The path to the destinations pass through two intermediate routers R1 and R2. The routers use drop-tail queues and following [11] recommendations we set all buffer sizes to at least the Bandwidth-Delay product of the bottleneck link (Bandwidth-Delay product = bandwidth in bits per second multiplied by total delay in seconds and divided by packet size [29]). This will prevent creating uncontrolled congestion drops along the connection path. The path to TD1 contains transmission errors at the last hop which will be used to test the proposed algorithms.

Link Bandwidth/Delay:

[1] See copyright statement in file tcp.cc in ns2

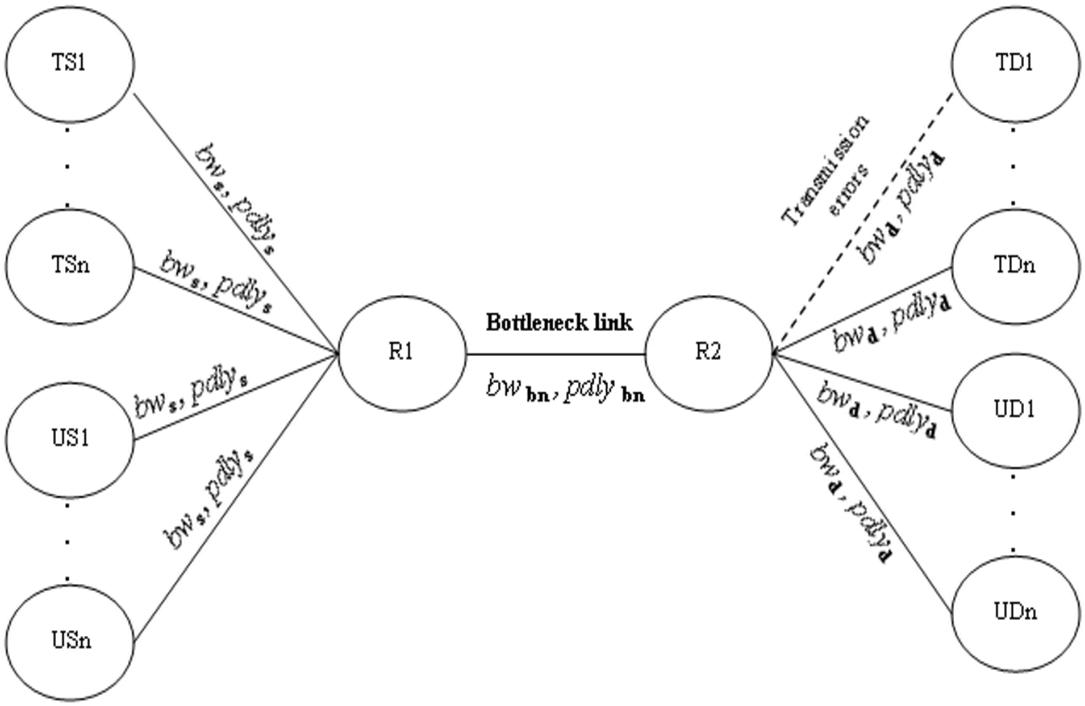

Figure 5.1: Network topology

In our experiments we vary the bandwidth and delay only when we want to measure their effect on the performance. However, our main focus is on the effect of transmission errors so when we vary the error rate the bandwidth is fixed to a typical T3 connection (45Mbps) and the maximum total propagation delay is fixed to 48ms (this value represents our attempt to obtain an average delay for UK-Europe connections as we will explain later). However, the actual total delay can vary due to queuing delay even if the propagation delay was fixed.

The bandwidth of each source is presented in figure 5.1 as bw_s and the propagation delay is denoted as $pdly_s$. The bottleneck bandwidth bw_{bn} and delay $pdly_{bn}$ are set according to whether we want to create congestion drops or not. If we do not want congestion drops then bw_{bn} is set to the aggregate bandwidth of all sources so that $\text{bw}_{bn} = \text{N} * \text{bw}_s$ where N is number of sources. However, if we want to create

congestion drops then we set bw_{bn} to be no more than 80% of the sources aggregate bandwidth and we tune $pdly_{bn}$ until we get the required level of congestion drops.

In this topology we create one bottleneck between R1 and R2 because of that the bandwidth and delay of the links from R2 to destination nodes, bw_d, $pdly_d$, takes same values as the bandwidth and delay from sources to R1, bw_s, $pdly_s$, in order to prevent creating bottlenecks at the last links.

Traffic:

The traffic for TCP sources is generated using FTP applications in ns2. We assume that FTP has always data ready to send (bulk data transfer). TCP packet size is chosen to be 1KB (1024 bytes) which is close to the default value in ns2 (1000 bytes). However, we set it to 1024 because the packet size is always rounded in TCP to the lower multiple of 512 bytes [35, P897] [23]. Choosing the optimum packet size is an open question and depends largely on the underlying network [86] so we do not deal with it here. TCP congestion window size is set to the Bandwidth-Delay product.

It has been accepted that the traffic in the Internet is of bursty nature and it shows self-similarity and correlation over large time scales [87]. This property of Internet traffic is called Long Range Dependence (LRD) [87]. One method to simulated LRD traffic is by using multiple On-Off sources where the length of On-Off periods follows long tailed distribution like Pareto [87].

However, a more recent model to generate LRD traffic is proposed by Muscariello et al [88]. The authors in [88] indicated that their model can approximate Internet traffic (burstiness and correlation) by generating multiple sessions and each session produces multiple flows and each flow produces multiple packets and this hierarchical model of sessions, flows and packets arrival processes are Poisson each with different arrival rates [88]. The authors showed that by multiplexing these Poisson arrival processes this model was able to approximate real Internet traffic traces presented

in [88].

So in our experiments the cross traffic is generated by multiplexing multiple UDP sources which generate packets using Muscariello et al. [88] traffic generator.

We choose this traffic generator because we found it to generate traffic with high degree of LRD. However, before using it in our simulator we validated that it is able to approximate Internet traffic as we present in appendix C. Also it is simple and easy to include in our simulator. This is largely because the authors provide the source code [89] which can be added easily to ns2 C++ implementation.

The Error Model:

One of important sources of non-congestion (transmission) errors in today's networks are wireless links.

We model transmission errors using a two state Markovian chain which is a simple model widely used to model existence of burstiness in wireless link errors [90]. Many authors indicated the bursty nature of transmission drops in wireless networks [3, 26, 90] which can lead to multiple packet drops from the same window. The error model consists of two states: a good state (error-free) g and bad state (error) b. The system can be at one state at any time. The duration in each state is a random number with exponential distribution.

Following [91], we calculate the average length of the good state, g, based on the length of the bad state, b, and the required error rate e using following formula [91]:

$$g = b \times \frac{1 - e}{e} \tag{5.1}$$

We calculate the average bad state length, b, based on the packet arrival rate to the lossy link. For example if packets arrive to the lossy link at a rate equal to 1 packet every 0.001 seconds, then to have an average burst size of 10 packets we

make $b = 0.01$ then we obtain the good state length using equation 5.1 based on the required error rate e.

Many authors used a similar two state Markov model to approximate wireless errors in their experiments; see for examples [56] [77] [72] [90] [61].

In most of our experiments we run the experiment for a minimum of 100 seconds (100000 ms) and the collection of data starts after a 5 seconds in order to remove any initial effects and to allow the network to settle. From our experience we found that in most cases 100 seconds is enough to show all aspects of TCP behaviour. Moreover since we run each experiment multiple times, we calculate the 95% confidence interval and use it to decide if number of runs and length of runs is sufficient as we will explain in more details in section 5.2.6.

This topology and all previous settings will be the basic design for all experiments in the rest of the thesis. However, during each experiment we will specify any settings specific to that experiment and any change we need to do in the general structure described above in order to make it easy to replicate any experiment we did.

As we said before the main aspect we want to test is the effect of errors on TCP performance and how our proposal will improve TCP performance and prevent increasing congestion level. Although this topology may seems simple we think it is sufficient to evaluate these aspects. A more complex topology will be used later to study other aspects of TCP performance.

Moreover, although similar topologies are used by other authors like in [1, 85, 92] to evaluate TCP variations, we must emphasize that all the results obtained are limited to any limitation found in our topology and we hope we can explore more complex topologies in future work.

5.2.3 Round Trip Time - RTT

Round Trip time is the time between sending a packet and receiving its acknowledgment by the sender (RTT = summation of the links propagation * 2 + processing time in the intermediate nodes). Because RTT is important to the performance of TCP since TCP performance depends heavily on the correct estimation of the RTT in order to calculate the retransmission time out at the TCP sender, we gave RTT careful consideration and we work to select the values of RTT in our experiments based on data from real networks.

In order to decide the minimum and maximum values for the RTT in our experiments we have extracted real Internet RTT readings from the Internet End-to-End Performance Monitoring Project (IEPM) [93, 94] which aims to monitor the performance of the Internet using a tool called PingER (Ping End-to-end Reporting) [94]. PingER uses simple packet echoing massaging mechanism where the sender sends a packet and waits for an echo from the receiver. Information like round trip time and packet loss rate can be then calculated.

We have extracted data about round trip times that cover twenty four months for the period from April 2005 to March 2007. These data are RTT readings from two different sites in the UK to 76 different Internet sites in North America, South America, Europe, East Asia and Africa. These data provide monthly readings of the Average RTT for the monitored sites. We have extracted the monthly average RTT for the period from April 2005 to March 2007 in order to decide the Max and Min RTT that we will use in our experiments. Figure 5.2 shows the Min, Max and Average RTT during the 24 months period (April 2005 to March 2007). The Max RTT represent the maximum RTT reading for each month and the minimum represent the minimum reading. The Average RTT represents the average RTT during each month.

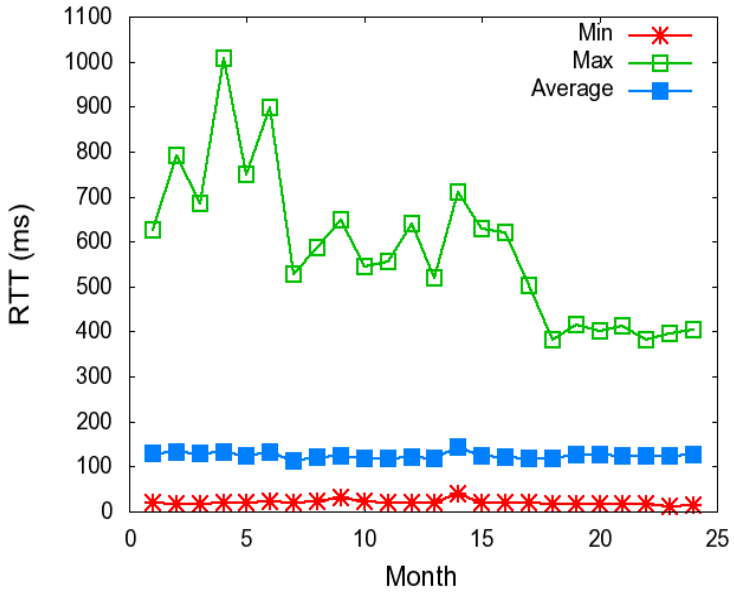

Figure 5.2: Min, Max and Average RTT values

Based on the data we have got and the compilation we did, RTT was chosen to be between 10ms to 700ms and during the experiments the RTT will vary between these values. These values are expected to present the Minimum and Maximum RTT that can be experienced in a real Internet connection.

When the RTT is fixed in the experiments it takes value of 48ms. This value was chosen in same way as Min and Max RTT but by taking readings for connections between UK and twelve different European countries . We take the average RTT for each month and then we compute the average RTT of the twelve months. The result is the average RTT between UK and the twelve European countries during the year 2006.

5.2.4 Performance Metrics

Different performance metrics are used in our work depending on each experiment. Following we list the metrics we used.

Throughput/Goodput: Throughput is a metric to measure the transfered volume of data in a given amount of time (for example number of bytes per second). However, throughput includes both new and retransmitted data.

Goodput is a subset of the throughput which considers only the data that have reached the destination safely without considering the retransmissions and it is a more accurate metric to measure the performance when we want to know the rate of actual data that has been delivered (which is what matters for the end user). Because of that we use goodput as a main performance metric in our experiments. However, since throughput is more common term we will use it to mean goodput unless stated otherwise.

Also, in many cases we use *Normalized Goodput* which is the per-flow goodput over the maximum achievable goodput and ranges from 0 to 1. The maximum achievable goodput is the bottleneck bandwidth.

Congestion window size: Congestion window is a metric unique to TCP which controls how many packets TCP can send before receiving acknowledgments. It is tightly related to the TCP sending rate. So a bigger congestion window is desired so the average congestion window size is an important metric to measure how fast TCP is sending data.

Number/Length of timeout events (RTO): When drops occur and acknowledgments stop arriving to the TCP sender, the sender waits for a timeout to occur to restart sending. However, the length of this timeout period affects the performance of the sender and the network at the same time. Too long timeouts will reduce TCP performance and too short timeouts may increase the congestion in the network. Also the frequency of timeouts is important since with each timeout event the TCP congestion window is reset to one segment.

Congestion loss ratio: Is the number of dropped packets from the bottleneck

router over total number of transmitted packets.

Transmission loss ratio: Is the number of dropped packets because of non-congestion errors over the total number of transmitted packets.

Other performance metrics like fairness between different flows are used also in some experiments as we will present later.

5.2.5 Simulation Validation

Since our simulation is based on ns2 [81] we assume that the underlying network simulation functions are validated. This is based on the fact that ns2 is a widely used simulator in computer network research [95] largely because it is an open source software so any bug in the code can be traced and notified to ns2 developers and the next release can have that update.

However, since most of our work is based on the TCP model in ns2 we use an analytical model provided by Padhye et al. in [96] to validate it. The Padhye [96] analytical model computes an approximation of TCP throughput taking into account the factors that affects the performance like errors and round trip time. We use the analytical model to compute the expected results and then compare them with our experiments results.

The Padhye [96] model is:

$$\mathrm{Thput(e)} \approx \frac{1}{\mathrm{RTT}\sqrt{\frac{2ae}{3}} + \mathrm{T}_0 \cdot min(1, 3\sqrt{\frac{3ae}{8}})e(1+32e^2)} \tag{5.2}$$

Where RTT is the round trip time, a is the number of packets acknowledged by a single acknowledgment, e is the error rate and T_0 is the duration of the first (minimum) timeout occurred during the connection.

Mainly the model can be divided into two parts one to compute the effect of

congestion window cuts on the performance and another to compute the timeout effect. The first part in formula 5.2 is :

$$RTT\sqrt{\frac{2ae}{3}} \tag{5.3}$$

In this part the effects on congestion window size is calculated.

The second part considers the effect of the timeout:

$$T_0 \cdot min(1, 3\sqrt{\frac{3ae}{8}})e(1 + 32e^2) \tag{5.4}$$

As we said, in order to validate TCP in ns2 we use formula 5.2 to compute TCP expected throughput with different round trip time values from 32ms cross country delay [75] up to 630ms global Internet delay and with packet drop rate of 1%. Then we use ns2 to run an experiment using same conditions and compare the results as we can see in figure 5.3.

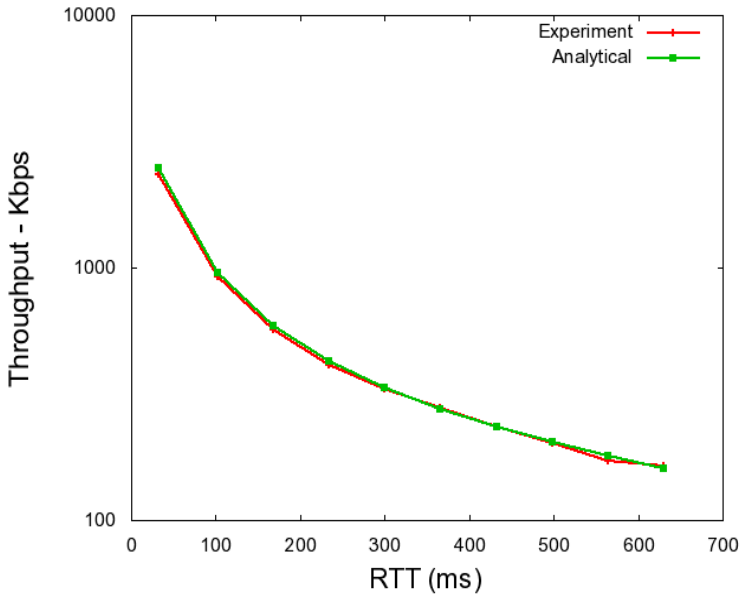

Figure 5.3: Analytical and Experimental models comparison

Since formula 5.2 is an approximation of the expected throughput we do not expect to get exact match between the experimental and analytical results. However, the experiment results should follow a similar pattern as the analytical results and it is clear from figure 5.3 that experimental and analytical results follow similar reaction to packet drops and to RTT increase. Moreover, both results follow expected behaviour of TCP since typically TCP throughput decreases with the increase in RTT for fixed window size. One contribution of our work is the validation of TCP model in ns2.

Since in our experiment our aim is to see how a single TCP connection reacts to different RTT values and the drop rate so we used a subset of the topology presented in figure 5.1. In this topology there is one TCP sender, *dly* takes values from 32ms to 630ms and the last link suffers from a transmission error rate of 1%. The bandwidth in all links is set to 45 Mbps.

Moreover, Biaz et al. in [60] gave a general explanation of how to use the Padhye [96] model to compute TCP performance after adding congestion and transmission errors. They explained how to introduce both error types as follows : consider the transmission error rate to be e_t and the congestion error rate to be e_c. Then the error rate e in formula 5.2 is computed as following $e = e_t + e_c$.

We believe that validating TCP in our simulator model gives validation to the models we derived from it. Also, during our simulation we use extensive tracing for many variables in order to spot any unexpected patterns. This process is continuous and goes with each experiment. We will mention any unexpected behaviour and explain it when we discuss experimental results.

Finally, we implement some mathematical functions in our simulation like the correlation, covariance and slope functions. In order to validate these functions we run the simulation using these functions and then compare the result we got

with the result using the same function from a known mathematical software like MatLab [97] or Ms-Excel [98]. By using this method we can ensure the correctness of our implementation.

However, the reason why we implemented these functions directly in ns2 instead of using external software was because we needed instant results during the simulation itself since some algorithms use these functions.

5.2.6 Confidence Intervals and Relative Precision

Usually when we run a simulation we use different random numbers for each run in order to produce different traffic and error patterns. Then we calculate an average value of all outputs from all experiments we have done. This average is an estimation of the actual average we get if we run the experiment for all possible random numbers. However, since we cannot run the experiment for all random number we need to know how accurate is our average (i.e. how close to the actual average). Here comes the role of confidence levels which is a statistical method to give us confidence on the simulation output average value.

According to [3] the simulation result can be considered accurate enough if we have at least 95% confidence level. The 95% confidence intervals can be computed by running the experiment multiple times as follows:

- First run the experiment for N times each with different seed for the random number generator.
- Consider X_i the output of each run where $1 \leq i \leq N$.
- Consider $\bar{X}$ the average of Xi.
- Compute the standard deviation of all runs $\sigma[X]$. The standard deviation is

computed using the following equation: $\sigma[X] = \sqrt{\frac{\sum_{i=1}^{N}(X_i - \bar{X})^2}{N}}$.

- Compute the confidence intervals as following : $CI = \frac{\alpha \times \sigma[X]}{\sqrt{N}}$. Where α depends on the confidence level we desire which is in our case 95% and according to [3] $\alpha = 1.96$ for confidence level 95%.

Now, after computing CI we can say that we are 95% confident that the actual mean is in the range $\bar{X} - CI \leq Actualmean \leq \bar{X} + CI$.

However, the confidence interval CI can be very large which indicates that we need more runs of the experiment. For that we can use the relative precision RP [3] which is the ratio of the confidence interval CI to the simulation average output $\bar{X}$, $RP = \frac{CI}{\bar{X}}$, and using it we can know when we can stop repeating the experiment.

Following [3] [99] [100] we choose RP be at most 10%. So the experiment should be repeated until RP $\leq$ 10%. However, in many experiments we try to reduce RP to be $\leq$ 5% which will give more accurate average value.

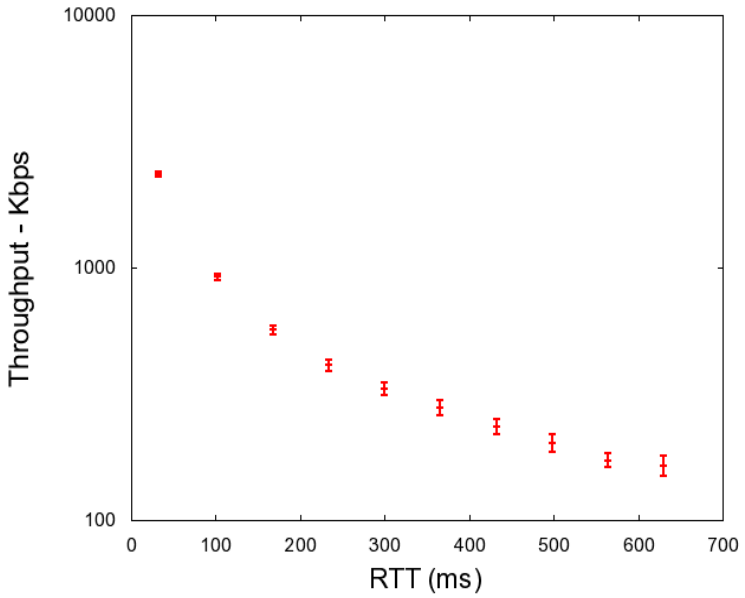

Figure 5.4: Experimental results with 95% confidence intervals

Following we will give an example of using 95% confidence level. We will run

RTT	Throughput	Standard deviation	Confidence Intervals	Relative precision
32	2338.59	113.72	49.84	0.021
102	920.30	62.07	27.20	0.029
168	566.61	56.92	24.94	0.044
234	410.05	46.77	20.50	0.049
300	333.03	44.27	19.40	0.058
366	279	40.83	17.89	0.064
432	234.67	36.63	16.05	0.068
498	202.47	39.37	17.25	0.085
564	172.57	24.40	10.69	0.061
630	164.33	33.97	14.89	0.090

Table 5.1: Confidence intervals and relative precision for 95% confidence level

TCP with different RTT values (as we did in the previous section to validate TCP) and this time we will add the 95% confidence intervals with relative precision no more than 10%. Figure 5.4 shows the results. As we can see in most cases confidence intervals are very small which indicates that we did repeat the experiment sufficient number of times at each point.

Table 5.1 shows the 95% confidence intervals and relative precisions for the points in the figure.

5.3 Performance Evaluation of TWA

In this section we will test the performance of the transmission window actions TWA (CWA, MDA and RTA) presented in chapter 4. As we said in chapter 4 the main objectives of TWA are to increase TCP throughput over lossy links and to avoid increasing congestion in the network.

We will add TWA to TCP and test if it can improve TCP performance in presence of transmission errors (first aim). We will use the topology explained in section 5.2.2 but the assumption here is that all drops can be correctly diagnosed by an error discriminator and TWA will be called only for transmission (non-congestion) errors.

Because of that bottleneck bandwidth will be equal to the aggregate bandwidth of all senders (i.e. for N sources $bw_bn = N * bw_s$) in order to avoid creating congestion.

Later in chapter 6 we will incorporate TWA in an error discriminator and show how it can avoid increasing network congestion (second aim) as well as improving the performance when transmission and congestion errors coexist.

In order to show the effect of adding TWA to TCP we compare it with the throughput of TCP using the same topology and experimental conditions. We run the experiment multiple times each time with different seed for the random number generator so we can generate different traffic and error patters. Transmission error rates generated as explained in section 5.2.2 and ranges from 0.001 to around 0.1 with 0.01 increase step (so exact errors rates used are 0.001, 0.011, 0.021, 0.031, 0.041, 0.051, 0.061, 0.071, 0.081, 0.091 and 0.101 error rates). This range covers low, medium and high transmission error rates as presented in [1] for wireless networks.

We found that the experiment run time and number of runs are sufficient that the 95% confidence intervals are very small and do not appear in the charts.

Figure 5.5 shows the performance of TCP before and after adding TWA in a semi-log scale. Goodput is normalized by each flow fair share of the bottleneck bandwidth.

Also we compare TWA with two TCP variations, TCP-NewReno [37] and TCP-Sack [45] (both explained in chapter 3). However, TCP-NewReno and TCP-Sack has an advantage over TCP that they can recover multiple drops from the same window.

In figure 5.6 we present the performance with TCP-Sack [45]. In figure 5.7 we add TCP-NewReno [37]. As we can see in all cases TWA has the highest performance.

However, we can notice that due to different techniques used by NewReno and Sack to recover form multiple packet drops, Sack has a higher improvement

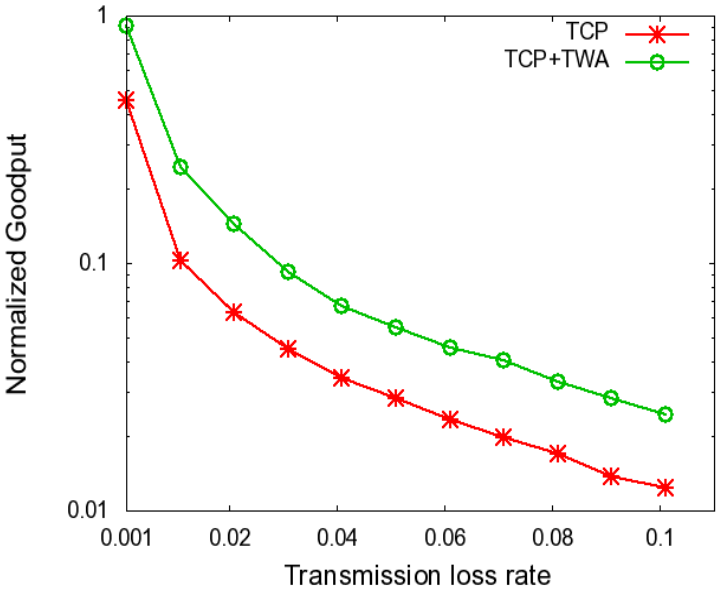

Figure 5.5: TCP and TWA semi-log scale normalized goodput

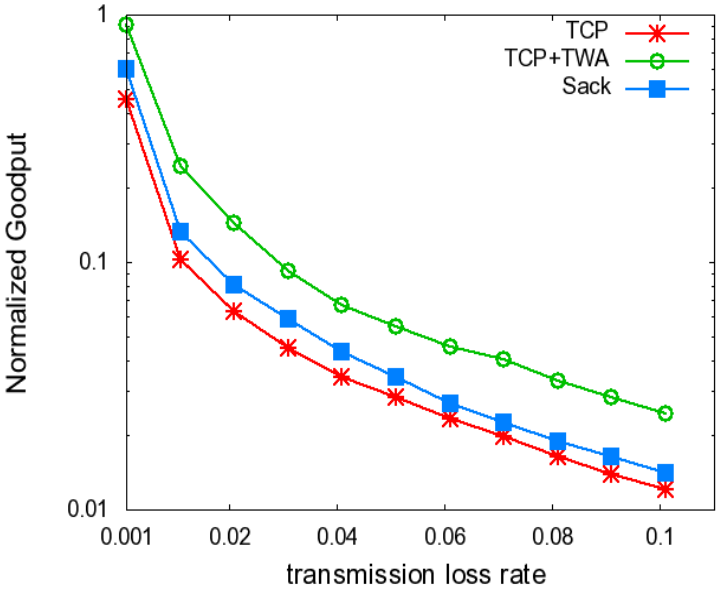

Figure 5.6: TCP, TWA and Sack semi-log scale normalized goodput

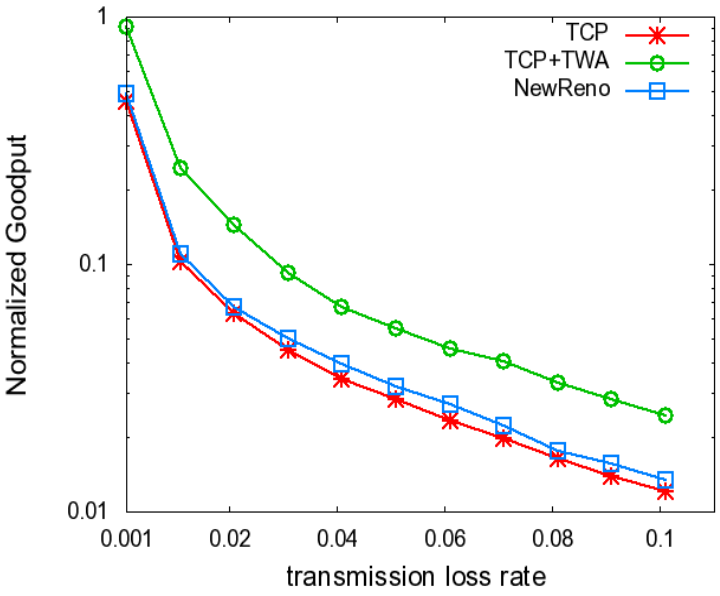

Figure 5.7: TCP, TWA and NewReno semi-log scale normalized goodput

over TCP. The reason why Sack has a higher performance than NewReno is that NewReno is able to recover one packet every RTT while Sack is able to recover multiple dropped packets in a single RTT [46].Moreover, since NewReno can recover one packet per RTT it has very small improvement over low error rates (over 0.001 error rate NewReno is higher than TCP around 6% and the peak improvement is 16% over 0.061 error rate) where number of drops is small compared to higher error rates.

The average improvement of TWA over TCP is 105%. Although this improvement seems high it is common to have similar improvements when TCP avoids cutting the congestion window. For example TCP-Casablanca [56] has similar improvement, however, TCP-Casablanca [56] does not implement action for transmission errors and hence avoids cutting congestion window in case of transmission errors. However, the merit of our technique is that we do not avoid cutting the congestion window completely which will help preventing congestion as we will see

later.

The main reasons why TWA is able to also outperform both Sack and NewReno although they can recover multiple dropped packets is that besides MDA which allows TWA to recover multiple drops, TWA also applies CWA which reduces the congestion window cut rate from 50% each time a drop occurs to a rate equal to the number of dropped packets from each window. Also, the use of RTA in TWA has reduced the length of timeout events.

To explore more the effect of each component of TWA (CWA, MDA and RTA) we show in figure 5.8 how TWA increases the average *cwnd* size compared to TCP.

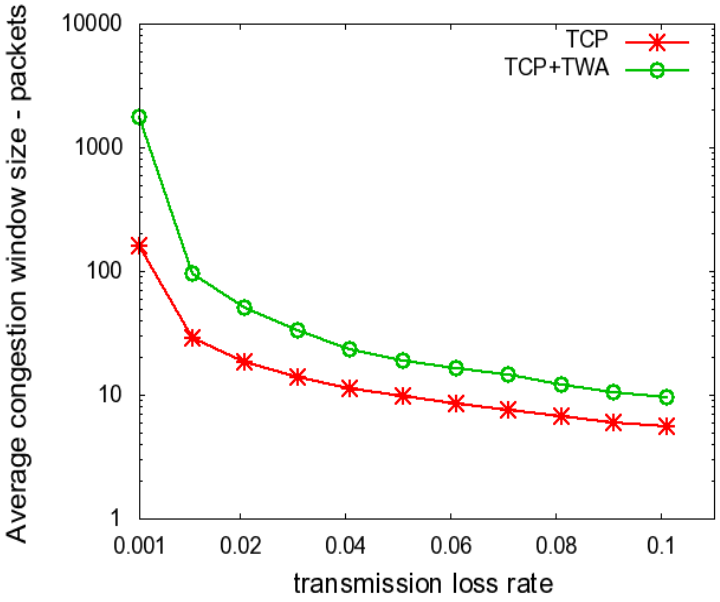

Figure 5.8: TCP and TWA average congestion window size

This increase is mainly due to the CWA component of TWA.

Also in figure 5.9 we show the decrease in number of RTO caused by applying MDA. Finally, RTA reduces the length of RTO and hence the time TCP stays idle as we can see in figure 5.10. All these factors contributed in the performance gain presented in figure 5.5.

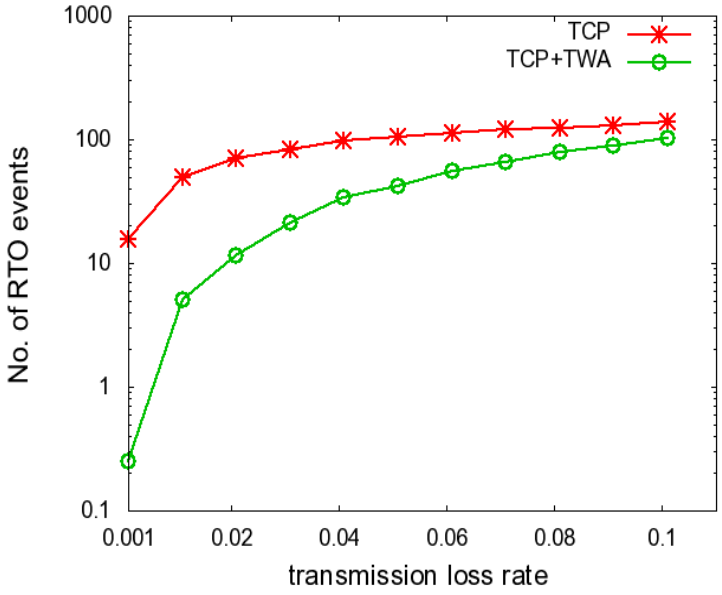

Figure 5.9: Number of retransmission timeout events

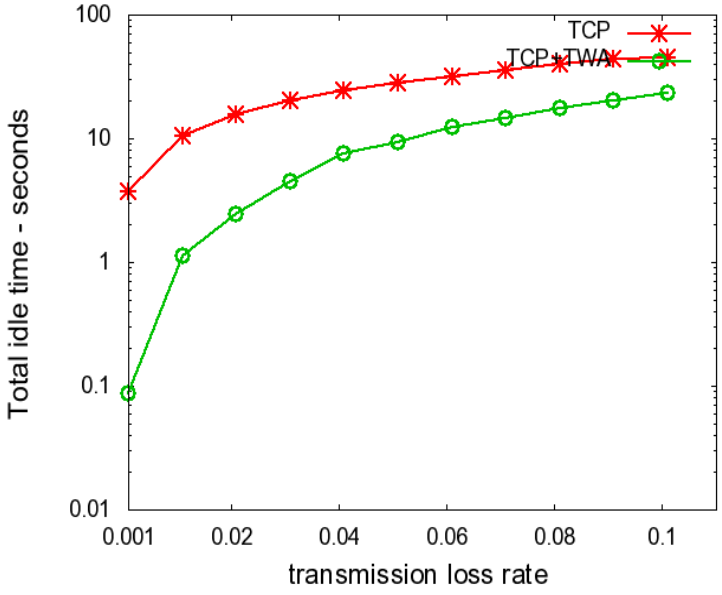

Figure 5.10: Total idle time for TCP and TWA

5.4 Summary

In this chapter we started by presenting our simulation model settings and the techniques we used to validate it.

We then tested the performance of TCP after adding the new transmission window action TWA. The results show that TWA has a positive impact on TCP performance in presence of transmission errors.

Then we tried to answer the question of why TWA is able to improve TCP performance by looking at each algorithm that forms TWA namely: CWA, MDA and RTA and seeing how each one contributes to improving the performance.

However, the TWA has been tested so far only in the presence of transmission errors. Real systems may suffer from transmission as well as congestion drops. In the following chapter we will add TWA to an end-to-end TCP error discriminator to see how TWA affects the performance.

Chapter 6

End-to-end TCP Sender Error Discriminator with New Transmission Drops Action

6.1 Introduction

Our aim in this chapter is to design an error discriminator which can discriminate between errors that occur during congestion phases and errors that occur during non-congestion phases, and use the transmission drop actions (TWA) we proposed in chapter 4 to implement the error discriminator reaction to non-congestion (transmission) drops. This will allow us to test the performance of TWA in the presence of both congestion and transmission errors. Also it will allow us to see how TWA can improve upon TCP performance and at the same time prevent/reduce any increase in the network congestion level which could occur when we use error discriminators.

We call the proposed error discriminator TCP-RTT because it depends totally on the round trip time (RTT) in its operation. In the following section we will

describe the design of TCP-RTT.

6.2 TCP-RTT

Using round trip time RTT to predict congestion is not new; TCP-Vegas [49] for example uses round trip time to compute expected throughput and to adjust congestion window accordingly. It uses RTT and TCP window size to infer congestion by varying the sender window size and measuring changes in network conditions (i.e. RTT changes).

In our design we will use the increase in RTT as an indication of congestion. Studies like [72,101] have reported the presence of strong positive correlation between the increase in network load (congestion) and increase in round trip time in both wired and wireless networks.

We aim here to design a congestion predictor based on RTT estimation and to use it to aid TCP in order to improve its throughput by discriminating between error types when transmission errors co-exist with congestion errors.

Our design will be based on a sender based end-to-end error discriminator called the Spike [1] error discriminator (presented in section 3.5.2.13). The reason why we based our design on Spike [1] is that it is sender based, totally end-to-end and uses only delay information to predict congestion. Having an end-to-end solution will limit the changes to the end points and leave the network unchanged. Also, we want it to be sender based in order to limit the changes to the server side only, as the number of servers is usually much lower than the number of clients. .

Also, Spike [1] has shown that it has moderate accuracy in many of the scenarios presented in [1] so we can test the effect of adding the new transmission window actions (TWA) on the network congestion level.

Spike [1] defines two sates: Spike-state and non- spike-state. In the spike-state the connection is considered to be in congestion state and any drop occurring during this period is considered to be a congestion drop. During the non-spike-state any drop is considered to be a transmission drop [1] see figure 6.1. The system enters

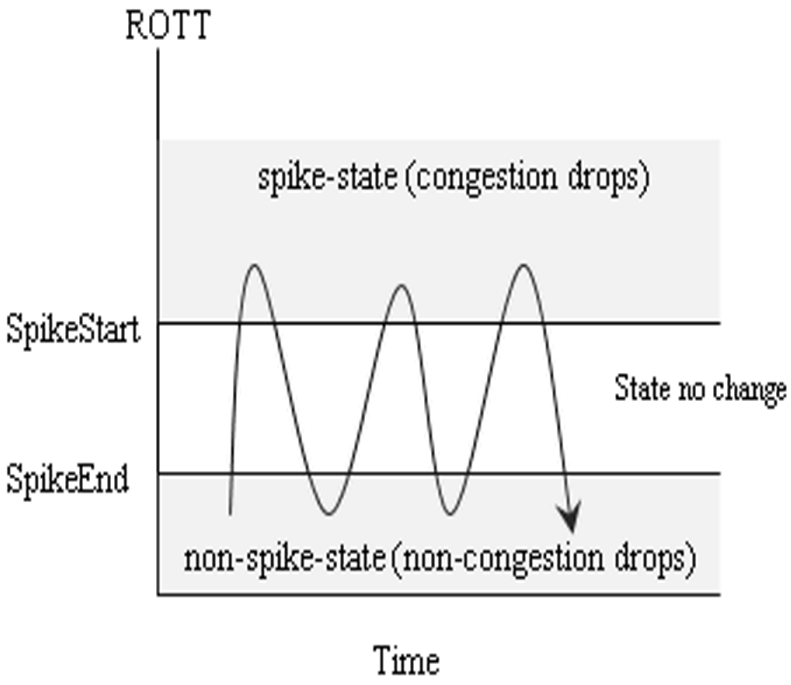

Figure 6.1: Spike states [1]

spike-state if the relative one way trip time (ROTT) exceeds a threshold, *SpikeStart*, and ends when the ROTT becomes below *SpikEnd* [1]. *SpikeStart* and *SpikEnd* are computed dynamically as follows:

$$SpikeStart = min(ROTT) + \alpha(max(ROTT) - min(ROTT)) \tag{6.1}$$

$$SpikEnd = min(ROTT) + \beta(max(ROTT) - min(ROTT)) \tag{6.2}$$

where the best result is obtained when α is 0.5 ad β is 0.33 according to the authors

in [1].

However, since there is no published implementation of Spike [1] error discriminator (as far as our search went), we have to design and implement the error discriminator from scratch and then to add it to ns2 [81]. This gave us a good opportunity to look at many aspects of the error discriminator functionality in great detail, and it allowed us to understand how it actually works and to understand the many difficulties an end-to-end error discriminator may face.

In the following section we will explain our design.

6.2.1 System Design

We propose a simple end-to-end error discriminator that will be used to incorporate the transmission window actions (TWA), as described in chapter 4 and tested in chapter 5, in order to test the positive impact these algorithms have on TCP performance and on network congestion.

We will present a congestion prediction mechanism that uses packet round trip time to give TCP more information about the network congestion status and that thereby enables TCP to discriminate between congestion drops and transmission drops.

We will use packet delay information to predict congestion as follows:

- The packet delay is composed of:
  - Link propagation delay
  - Queuing delay
- The link propagation delay depends on the medium (the link) type, and we assume that it is fixed for a single connection.

- The queuing delay is the time the packet spends on the intermediate nodes and it consists of two components: the queue waiting time and the service time. In our work we are concerned with the total time the packet spends in the intermediate node (queue waiting + service time) and we will refer to the total as queuing delay.

- Any increase in the network load will affect the queuing delay. When a node starts building a queue the Queuing delay will increase and hence the total packet RTT will increase.

- In order to capture congestion the TCP sender will compute an exponential weighted average of the RTT readings, AvgRTT. Using a weighted average will allow us to control whether the recent sample or the old samples have more effect on the AvgRTT.

We will define a variable called congestion edge (Cedge). The congestion edge is basically the RTT value where we consider any drop occurring after this point to be of high probability of being caused by congestion. First we compute a threshold based on the network total delay, and then the value of congestion edge is computed based on the minimum and maximum RTT experience so far, as follows:

$$Cedge = minRTT + midalpha * (maxRTT - minRTT). \tag{6.3}$$

Using this formula, the *Cedge* will be a value between *minRTT* and *maxRTT* and the value of *midalpha* will determine how close *Cedge* is to the *minRTT* or *maxRTT*. When *midalpha* increases the *Cedge* will go toward the *maxRTT* and when *midalpha* decreases *Cedge* will move toward *minRTT*. This feature has an important role in the discriminator function because any increase in the *Cedge* value will tend to

make the discriminator classify more errors as transmission errors and any decrease in *Cedge* will tend to make the discriminator classify more errors as congestion.

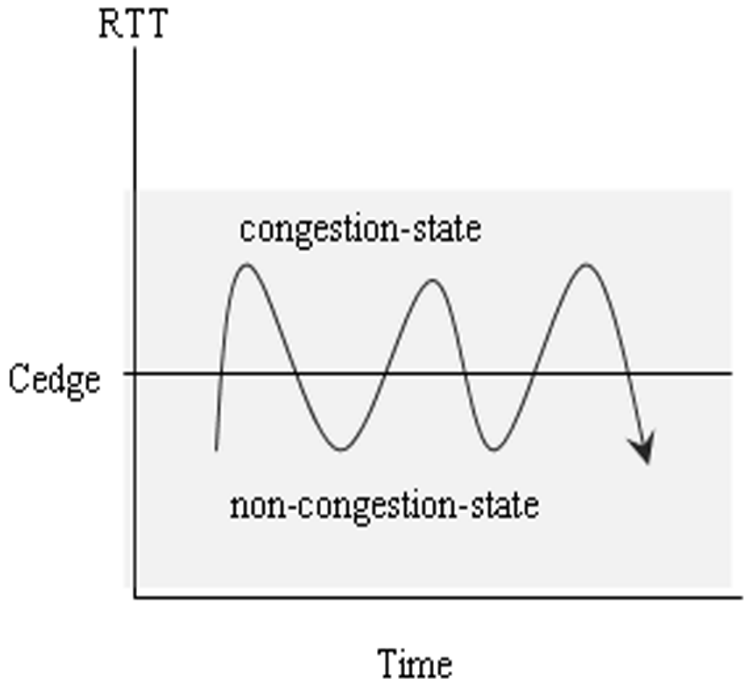

Figure 6.2: RTT states

The system can then be either in congestion state or non-congestion state depending on the current RTT value, as we can see in figure 6.2.

We assume that if the a drop happens and the RTT is below the Cedge then it is safe to consider the drop as a transmission drop even if it is congestion drop. This is because having a congestion drop under such low RTT (when compared to the maxRTT experience so far) indicates that it is a transient congestion and it may have been resolved allready.

However instead of using RTT, we use a weighted average RTT we call it AvgRTT. The AvgRTT is computed as in standard TCP by using an exponential weighted average with weight = α, which decides if the average has high or low

response to RTT readings (see [47, P69] for benefits of using weighted average)

$$AvgRTT = \alpha * AvgRTT + (1 - \alpha) * RTT. \tag{6.4}$$

We do this to avoid effects of sudden changes in the RTT readings. Following TCP specification [29,35] to compute average RTT we can set α between 0.8-0.9. In our experiments we set $\alpha = 0.9$ so that AvgRTT responds to genuine RTT changes only.

Now when an error occurs the error discriminator does the following:

- First it computes the AvgRTT and Cedge.
- If AvgRTT > Cedge then the drop is congestion. In this case we follow standard TCP procedure by reducing the congestion window to half.
- If AvgRTT $\leq$ Cedge then the error is considered as a transmission error and the transmission window actions (TWA) are used instead of TCP congestion control.

One aim of this error discriminator is to avoid having a congestion collapse in the network because of TCP-RTT not responding to network signals for congestions (mainly packet dropping) and hence causing congestion collapse. TCP-RTT avoids this by applying a technique that allows TCP to reduce its sending rate even for transmission drops. When TCP-RTT diagnoses an error as a transmission error it increases the retransmission rate in a balanced way, as follows: when a packet is dropped and the average RTT is below the *Cedge* then the error is considered potential transmission drop. TCP-RTT then applies CWA which does another check to see if the current congestion window is bigger or smaller than the *tthresh* (a mark for the first drops occurred). If the congestion window is bigger than the tthresh

then the error is considered congestion, otherwise it is considered transmission error.

This way a two level check is done before deciding the error type. The first level is by using the connection average RTT and the second level by using TCP congestion window size. The congestion window size, combined with the *tthresh* gives TCP a good indication of whether the drop is caused by congestion or not. Using this method, we follow a conservative approach that prefers the network over the single connection.

This way, in the case of heavy congestion the priority is given to congestion even if there are transmission drops. Thus if the error discriminator mistakes congestion drops for transmission drops, the network will not be affected; this will also not affect the end user to a great extent because the congestion in the network will slow down the sender in all cases.

Moreover, the MDA action will resend packets at only one per RTT, and this means that no more resending will happen until a timeout occurs. This way, even if TCP-RTT misclassifies the drop type, it will follow a conservative approach which will allow a timeout to occur if there is serious congestion.

Also by using RTA, when congestion occurs in the network, even if the TCP-RTT classifies the error to be transmission, the low bandwidth estimation will force TCP-RTT to slowdown (timeout back-off) just like a traditional TCP and hence reducing the chance to increase the congestion in the network.

As we said before TCP-RTT is based on Spike, so in the following we highlight the main differences between our design and Spike:

- The Spike [1] error discriminator uses two thresholds to decide the start and end of the Spike state. However, in our version, we use one threshold called *Cedge* which creates a border between the congested and non-congested states. This will simplify the algorithm and make it more pertinent to our aim, which

is to discover congestion phases. Originally the Spike [68] scheme was designed to discover different congestion levels [1] in order to control UDP sending rate. In our case the important thing is to discover the existence of congestion and not the level of congestion. For this reason we think that one threshold, *Cedge*, is sufficient.

- The Spike [1] error discriminator uses the relative one way trip time as an indicator of current network state and of whether or not we should enter Spike state. However, this makes the system not immune to sudden and short changes in the trip time caused by temporary congestions. So to filter sudden changes in packet trip time readings we use an exponential weighted average (EWA) of round trip time readings instead. This will reduce the oscillation between the congested/non-congested states and make the system more stable.

- The original Spike [68] was designed for UDP applications where there is no acknowledgment, so the authors used the relative one way trip time instead of the round trip time. Also because the sender and receiver clock may vary, the term *relative* is used. However, using the one way round trip time requires changes in both the sender and the receiver to allow them both to communicate the measured trip time. Since we use TCP, which uses acknowledgments, there is no need to use the one way trip time and instead we use the round trip time RTT where only the senders measure the trip time. However, we assume here that the return path is not congested.

- The Spike [1] error discriminator measures the maximum experienced ROTT and uses it during the whole connection life time. However this can cause a fake notion of non-congestion state if we have multiple buffers as presented in [1]. If a long congestion phase occurs or multiple buffers in the path became

congested for a period of time then the $max(ROTT)$ in this case will get a very high value, which will remain during the connection life time even if the buffers are eventually emptied. However we solve the problem by recalculating the maximum round trip time after each timeout event because a timeout event indicates the occurrence of a congestion state which may have not been noticed because of previous high RTT readings.

- The final and most important change is the addition of the CWA, MDA and RTA (TWA) which will represent TCP reaction during non-congestion state.

We call the proposed error discriminator TCP-RTT because it depends totally on the round trip time (RTT) in its operation. In the following section we will test TCP-RTT performance.

6.3 Performance Results

There are two main aims we want to achieve from using TCP-RTT: one is increasing TCP performance when congestion and transmission errors coexist. The other is to prevent/reduce increasing network congestion level because of error discrimination mismatches. We will try to show the achievement of these two aims in the following experiments.

6.3.1 Experiment Settings and Assumptions

In the following experiments we used the topology and experimental settings explained previously in section 5.2. However, below we list some assumptions and settings of this experiment:

- In these experiments we combine the transmission window actions (CWA,MDA

and RTA) described in chapter 4 with the error discriminator in order to implement better action for transmission drops.

- We use total propagation delay of 48ms. The bandwidth is fixed to 45Mbps. Later we will apply different delay and bandwidth values.

- Transmission errors range from 0.001 to 0.101.

- Congestion drop rate at the bottleneck is adjusted experimentally to be around 0.001. According to the results and observations of [72], even a perfect error discriminator will not give any noticeable improvement if the congestion drop rate goes too much above 0.001. This is because with high congestion rates TCP will be reducing its performance most of the time as a reaction to the congestion, and any improvement will not be noticeable. The drop rate of each connection will be proportional to its share of the bottleneck. However, the congestion drop rates of the monitored flows were found around the desired rate of 0.001.

- In the following experiments we repeat each experiment until the relative precision RP (RP: the ratio between the 95% confidence interval and the average goodput) reach values between 5 and 10% as we explained in section 5.2.6. Therefore we use confidence intervals and relative precision to determine how many runs we need. In each run we change the seed for the system random number generator. To ensure fair comparison we use the same set of seeds for TCP-RTT runs and for TCP runs. In all the following experiments we draw the 95% confidence intervals only if it is clear enough to appear on the chart. However, in most experiments, we repeat the experiment until the confidence intervals becomes no more than 5% of the average value, which in most cases has no appearance of significance in the charts.

- The traffic is generated in this experiment by using one TCP-RTT sender, one TCP sender and four MMPP sources(see section 5.2 for explanation of MMPP sources).

  Using these settings we was able to generate a traffic with Hurst parameter of around 80% in most cases.

- For simplicity, we assume that there are no drops on the reverse path (other authors such as [70] use the same assumption).

- In some charts we will use the log scale in the y axis instead of the normal linear scale if the values are spread over a large range. In this case the log scale will help to clarify the behaviour and make the range more understandable.

6.3.1.1 Performance Metrics

In the following experiments we will be using four experimental metrics, which are: Goodput, Congestion window size, Number of timeout events and Retransmission timeout length.

6.3.2 Results With No Congestion

In this experiment we will test the performance of TCP-RTT in a network with no congestion and with transmission errors only. This is important to show the improvement gained by the new method in case of transmission errors and to test the error discriminator response to transmission errors. If the error discriminator diagnosed errors as congestion errors while they are actually transmission errors then we do not expect any improvement in TCP performance. However, if the error discriminator succeeds in discovering that there is no congestion but actually there are transmission errors then we expect to see improvement in the performance. In

this experiment we will use the same experimental assumptions and performance metrics mentioned above, except that there are no congestion drops.

6.3.2.1 Goodput

One performance metric is goodput. As we said before, the goodput is the actual throughput the user will see. Thus the increase in goodput will be reflected directly on the application that uses TCP(like FTP for example). Figure 6.3 shows the

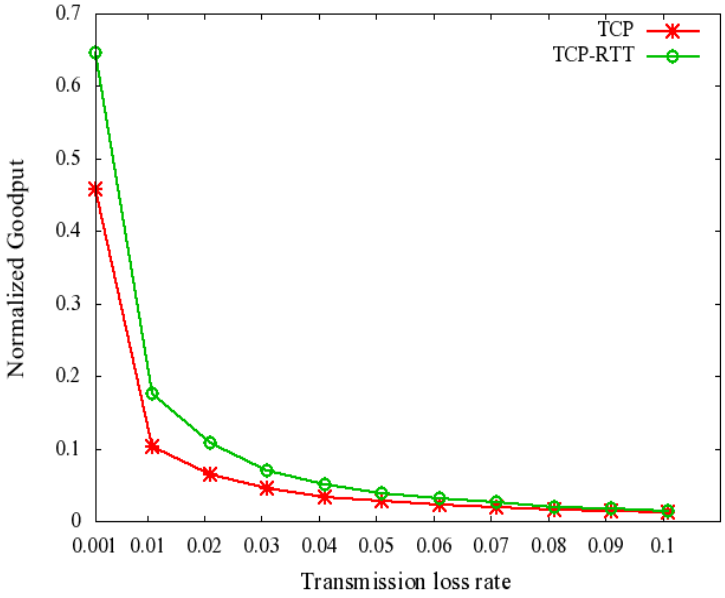

Figure 6.3: TCP vs. TCP-RTT normalized goodput

goodput normalized by the bottleneck bandwidth for TCP and TCP-RTT. Since there is no congestion the bottleneck bandwidth is set to the aggregate bandwidth of all senders. In this case we have one sender so the bottleneck bandwidth is set to 45Mbps.

As we can see in figure 6.3, TCP-RTT has higher goodput with different transmission error rates. However, because the results are spread over a large range, it appears that the improvement is only major at lower error rates and becomes minor

with increases in the error rate.

However, when we take the log scale for the goodput, as in figure 6.4 , we can

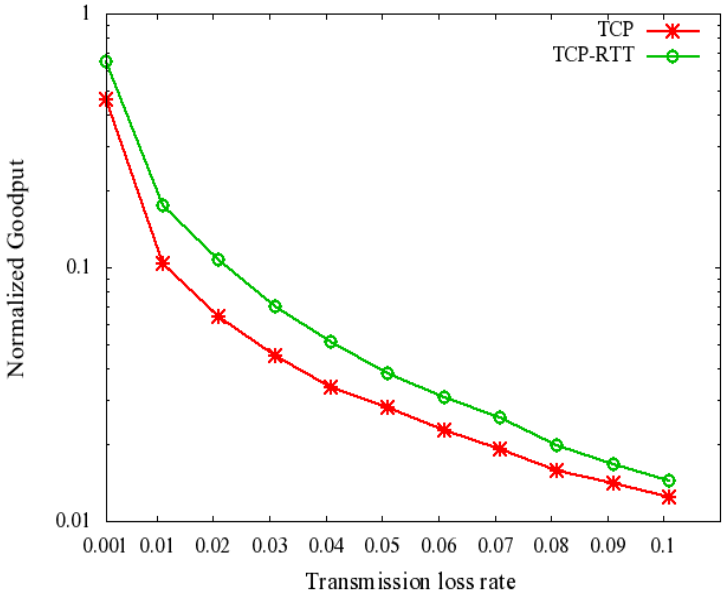

Figure 6.4: TCP vs. TCP-RTT semi-log scale normalized goodput

see that the improvement covers all error rates.

The peak of TCP-RTT performance improvement is at 1% (about 70% higher than TCP). Before and after 1% there is less improvement. The reason why there is less improvement before 1% is that under such low transmission error rates there is not much for the error discriminator to do as there is such a small number of errors. Add to this that, if some of these errors are classified wrongly as congestion drops, as indeed they may be, the performance of TCP-RTT will be reduced.

On the other hand when transmission errors increase to more than 1%, the actual number of transmission errors becomes higher so that some drops will not be recovered due to the high number of drops. This will cause timeouts, even with the error discriminator, so the connection will be idle for longer periods due to longer retransmission timeouts. However, TCP-RTT reduces these effects by using TWA

and hence we can see improvement in the performance in figure 6.4.

6.3.2.2 Congestion Window

The congestion window size reflects the actual sending rate of TCP. The increase in the congestion window size will result in increasing the goodput of TCP. In figure 6.5 we can see that the TCP-RTT average congestion window size is bigger over different

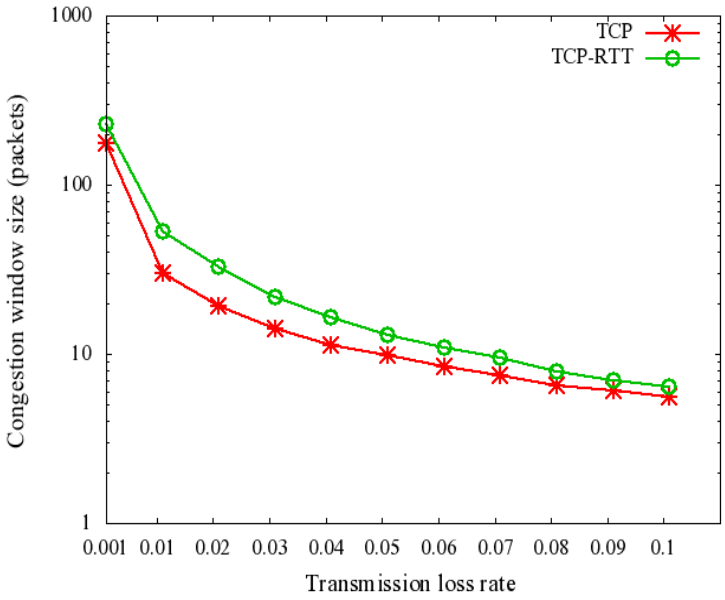

Figure 6.5: TCP vs. TCP-RTT semi-log scale congestion window size

error rates. This explains the increase in the goodput we noticed before in 6.4.

The improvement in the congestion window size indicates that TCP-RTT is able to detect transmission errors and hence is able to prevent cutting the congestion window to half, as TCP does. Instead TCP-RTT cuts the congestion window using CWA, which cuts the window according to the number of dropped packets per window.

Figure 6.6 shows a snapshot of the congestion window for TCP and TCP-RTT.

The figure shows the evolution of the congestion window size during the first

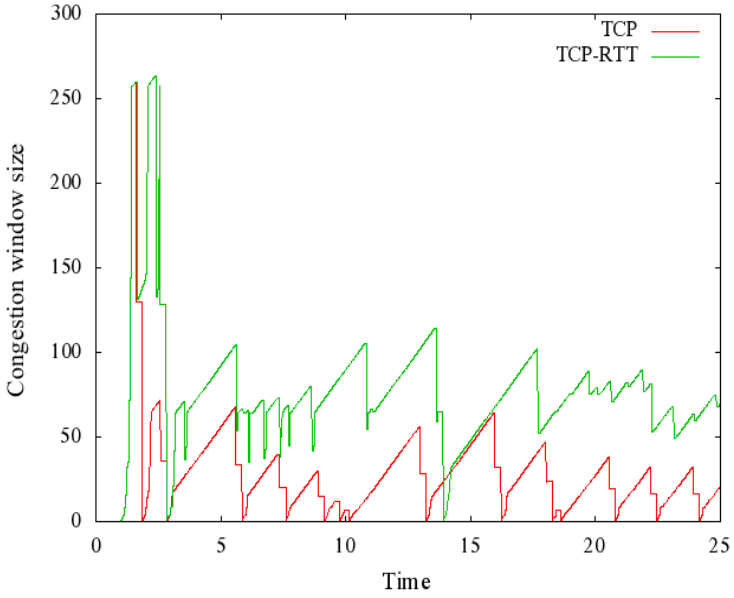

Figure 6.6: Congestion window evolution

25 seconds of connection life time and as we can see the TCP-RTT has a bigger congestion window most of the time.

Figure 6.6 shows two different behaviours of the TCP-RTT congestion window. Firstly the congestion window is cut to half when a drop is considered as congestion (for example at time 5), and secondly the congestion window is cut to the number of dropped packets (as we can see around time 20).

6.3.2.3 Number of RTO Events

Each RTO (retransmission timeout) event will result in reducing the congestion window size to minimum size (one segment). In figure 6.7 we can see that TCP-RTT has reduced the number of RTO events. TCP-RTT reduces the number of RTO events because of the multiple drop action (MDA) which tries to resend all lost packets from the same window, thereby reducing the number of timeout events.

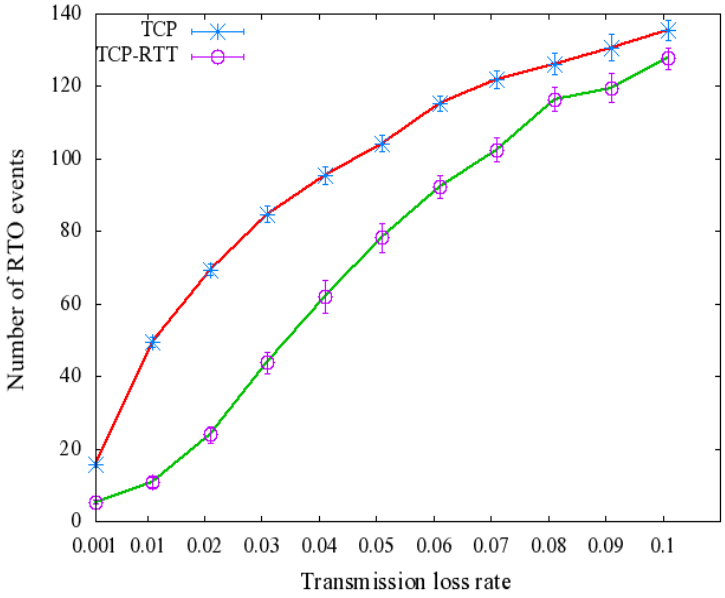

Figure 6.7: TCP vs. TCP-RTT number of RTO events

However, as we indicated in chapter 4 that the increase in the number of timeouts indicates that when the error rate increases and more packets are dropped, timeouts may occur before MDA is able to resend all lost packets. Additionally, TCP-RTT only uses MDA when it thinks that the a drop is transmission drop; when a drop is considered congestion drop, MDA will not be used.

Moreover, timeouts are also expected to occur with higher error rates where the same packet may be dropped more than once, and as we explained in chapter 4 that MDA resends the packet only once per window to prevent unnecessarily retransmission when the link is dead for long time, so even with MDA timeouts can occur when a packet is dropped multiple times.

All these factors lead to increase the number of timeouts, even with MDA. However, MDA was able to reduce the number of timeouts, especially for low error rates, and this is reflected in the increase in the TCP-RTT performance that we saw.

6.3.2.4 RTO Length

We measure here the total time TCP stays idle without activity, which is reflected directly in the performance: the longer TCP stays idle, the lower the performance (goodput) will be. As we said before, TCP-RTT uses MDA action to reduce the number of timeouts by trying to resend all lost packets from the same window. This will result in reducing the total length of RTO as well.

Also, the RTA action will reduce the idle time by computing the TCP back-off level based on the available bandwidth, and in the case of transmission errors, the available bandwidth should be higher than in the case of congestion errors. Both the MDA and RTA will participate in reducing the total RTO length and hence in increasing TCP performance.

Figure 6.8 shows RTO length for TCP and TCP-RTT for increasing transmission

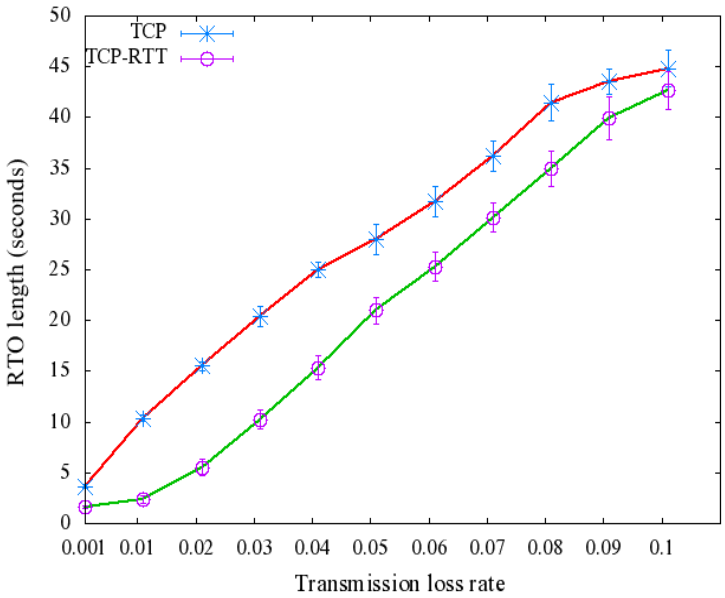

Figure 6.8: TCP vs. TCP-RTT RTO length

error rates. As in the case of number of RTO, TCP-RTT is able to reduce total TCP idle time leading to more activity periods and hence to more average goodput.

6.3.2.5 The Effect of Packet Round Trip Time

The increase in the packet end-to-end delay (or round trip time) means that TCP needs more time to increase its congestion window because TCP increases the congestion window based on the rate of received acknowledgments and if these acknowledgments take more time because of longer delays, this will cause the congestion window increase to take longer.

However, the effects of RTT on TCP performance increase when there are drops because longer RTT means more penalty on the TCP congestion window when errors occur. If a drop occurs at low RTT, the recovery time will be less than when an error occurs under higher RTT.

Here we will show the performance of TCP-RTT under different RTT values, namely 24ms, 48ms and 96ms (note that we only control the propagation delay and we do not control the queuing delay). The aim is to show that TCP-RTT is able to work under different RTT values.

However, reducing RTT is not usually the job of TCP and it is mainly the job of the network (Queuing mechanism, Routing protocols, etc.) to provide services with lower delay and hence lower RTT.

Figure 6.9 shows that TCP-RTT goodput decreases with the increase in the RTT value. However, TCP-RTT still outperforms standard TCP when they both have the same RTT value.

6.3.3 Results With Congestion

In the previous section we showed the performance of TCP-RTT in case of transmission errors only and there was no congestion in the connection path. However, in real networks, both congestion and transmission errors may coexist in the same

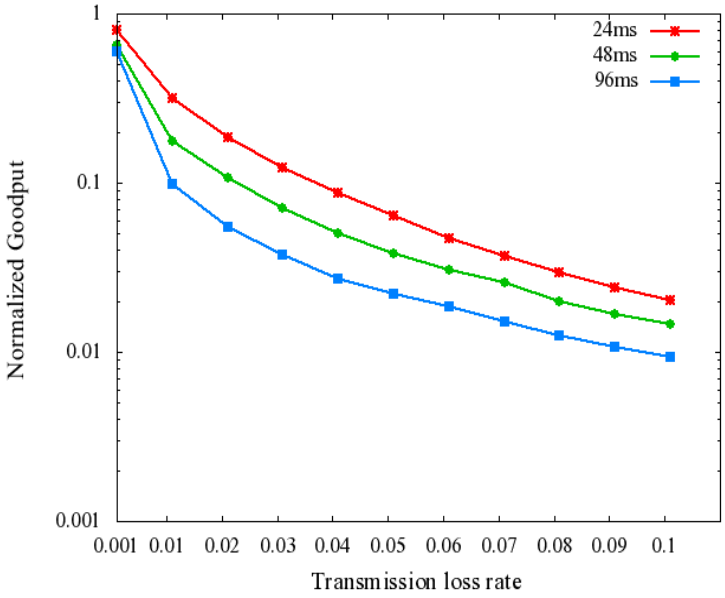

Figure 6.9: TCP-RTT under different RTT values

path. For this reason, in this section we measure the performance of TCP-RTT with transmission errors and congestion errors created by cross traffic. We will use the same experimental assumptions and performance metrics mentioned above in section 6.3.1.

6.3.3.1 Goodput

Figure 6.10 shows the comparison between TCP and TCP-RTT when both suffer from a congestion drop rate of 0.001 and transmission errors ranging from 0.001 to 0.10. When the transmission error rate is small (0.001), there is no improvement because under such a small number of transmission errors the congestion losses dominate TCP-RTT actions so that most of the time TCP-RTT responds to congestion losses and cuts the sending rate. However, the improvements can be seen with the increase in the transmission error rates.

Unlike in the previous experiments, in this experiment TCP-RTT is competing

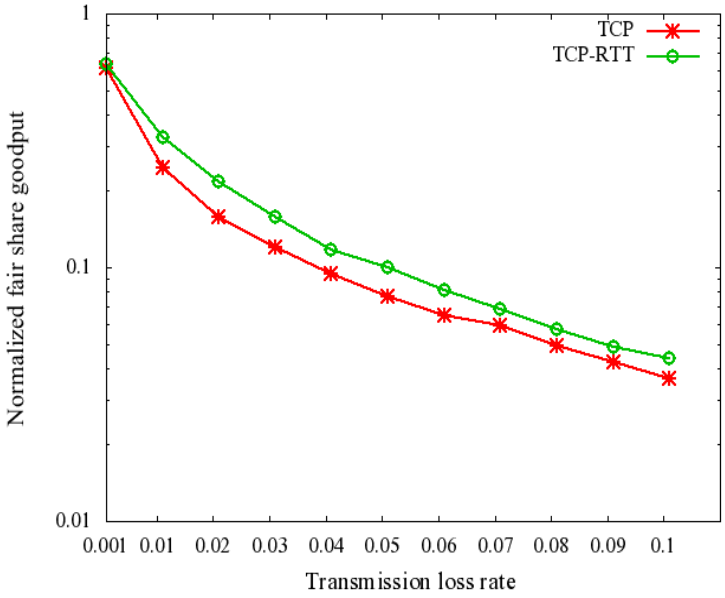

Figure 6.10: TCP vs. TCP-RTT semi-log normalized fair share goodput

with other cross traffic. Because of this, in figure 6.10 we use the normalized bottleneck fair share of competing connections (i.e. we divide each connection goodput by its fair share of the bottleneck bandwidth).

6.3.3.2 Congestion Window and Retransmission Timeout

With the increase in the packet drop rate TCP will get more congestion window cuts and more timeout events. However, in case of TCP-RTT the discriminator will avoid too many cuts in the case of transmission errors by using the CWA. Also TCP-RTT will reduce number of times it falls into timeout events by using the MDA action for transmission errors, thereby reducing the number of times for setting the congestion window to minimum values because of timeout. Hence the effect of the increased number of timeouts will be eased on the TCP sender.

Also the use of RTA will reduce the length of each RTO event in the case of transmission errors if the network is not congested. Figures 6.11, 6.12 and 6.13

show the improvements in the congestion window size, the number of retransmission timeout events and the timeout length respectively.

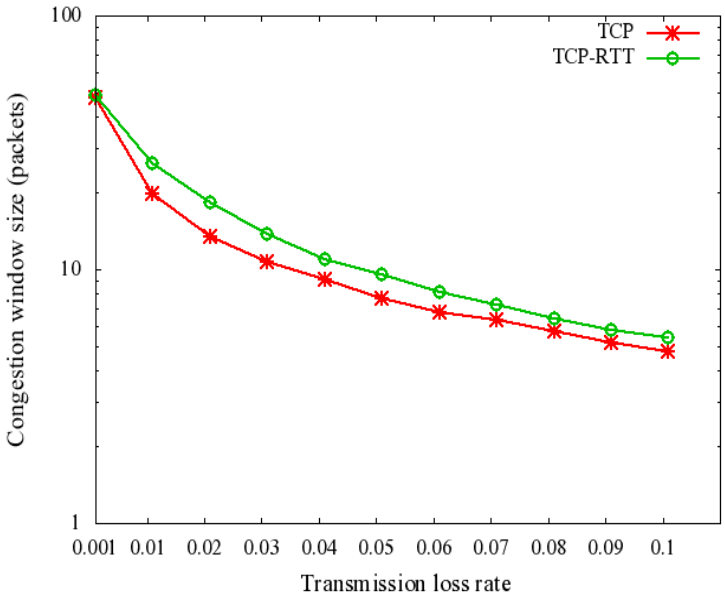

Figure 6.11: semi-log congestion window size

As we can see in figure 6.11 TCP-RTT has higher congestion widow size which explains the improvement in the goodput seen in figure 6.10. Moreover, in figure 6.12 TCP-RTT has reduced the number of RTO events which reduce TCP idle time and reduce the cut on the congestion window size which happens after timeouts. However, at 10% error rate TCP stops increasing the number of RTO events. This is expected because with such a high error rate the connection dries quickly from packets due to the large number of dropped packets; when the number of sent packets decreases, the number of timeouts does not increase too much.

Moreover, the increase in the error rate will increase the probability of having multiple packet drops in the same window, and this will have a negative effect on TCP (timeout will increase exponentially). However, TCP-RTT will avoid using the exponential back-off timeout procedure directly whenever the loss is detected

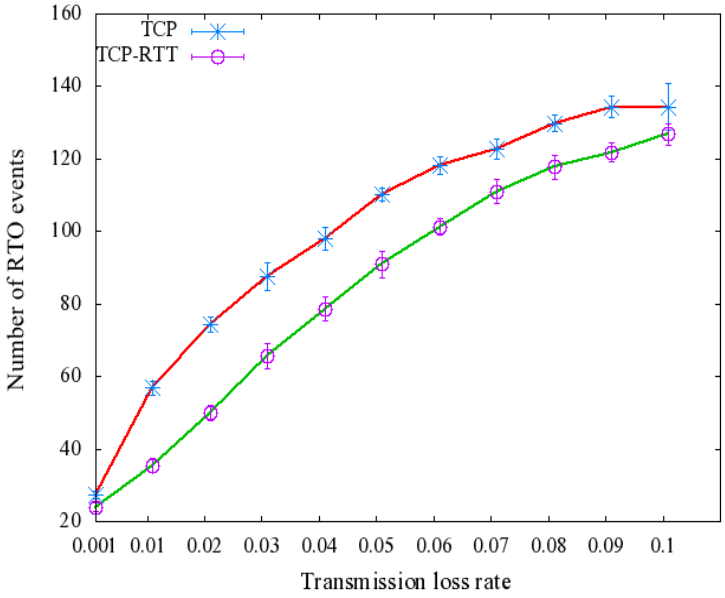

Figure 6.12: Number of RTO events

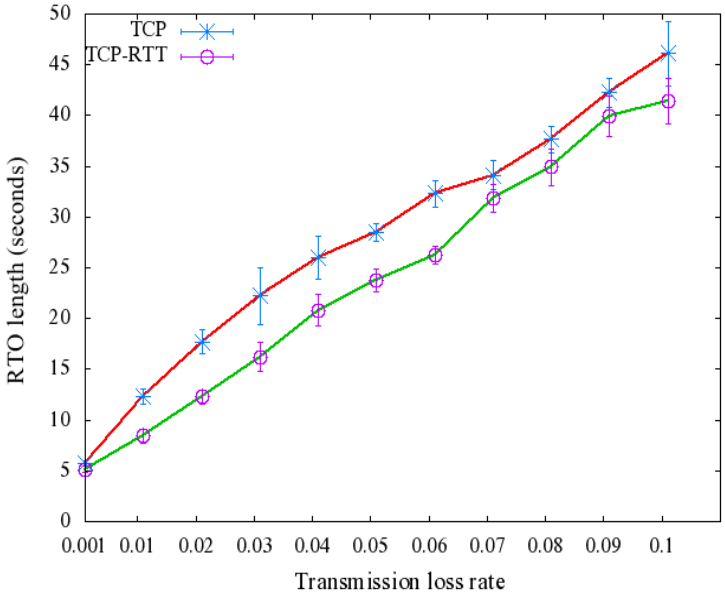

Figure 6.13: Timeout length

as transmission, and will use RTA action instead. This has led to reduction in the timeout length for TCP-RTT with different error rates compared to TCP as we see in figure 6.13.

A question may rise over why the number of RTO for TCP stops increasing at 10% in figure 6.12 and at the same time the timeout length still increases for TCP in figure 6.13. The reason for this is the exponential back-off policy. Under high error rates the sending rate decreases so does the number of RTO events. However, due to the high error rates, a single packet may be dropped more than once, and this triggers the exponential back-off in TCP. This will increase each RTO event length so even if the number of RTO events is reduced, their lengths increases because of the exponential back-off policy.

6.3.3.3 The Effect of The Error Model

In the previous experiment we used a bursty error model that produces more than one consecutive packet drop on average each time errors occur.

In this section we will apply a uniform error model that produces one packet drop on average each time an error occurs, specially under low error rates. For this we use the standard error model in ns2 [81] (*ErrorModel*) and we use a uniform random number generator to generate errors with a specific rate (like for example 1%). The uniform error module gives equal drop probability to each packet. This will create uniformly distributed errors along the connection.

We repeated the same experiments (with congestion) and the results are presented in figure 6.14. We can see that TCP-RTT outperforms TCP under uniform transmission errors. This indicates that under different error models, TCP-RTT still outperforms TCP

Another observation is that under uniform errors, TCP gives less performance

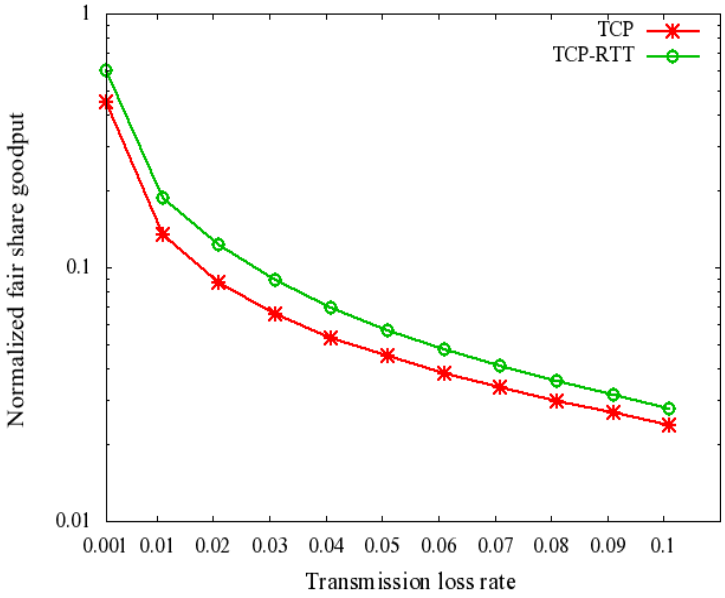

Figure 6.14: TCP vs. TCP-RTT semi-log normalized fair share goodput - Uniform transmission errors

gain compared to when we used the bursty error model. Figure 6.15 compares the performance of TCP under uniform errors (TCPU) and under bursty errors (TCPB). The same thing also can also be observed in TCP-RTT, as we can see in figure 6.16.

We think that the reason why the performance is different with different error models is related to the nature of the dropping pattern produced by each model. In the case of bursty drops, the bursty model drops multiple packets each time it enters an error state, so it needs less number of error states in order to reach the required dropping rate (like 1% dropping rate for example) giving longer times between error states.

However, the uniform model drops fewer packets (one packet on average) in each error state, so it needs more error states, and hence TCP has less time to increase the congestion window between drops. This will affect the way the congestion window grows and will give it less chance to grow in the case of the uniform error model

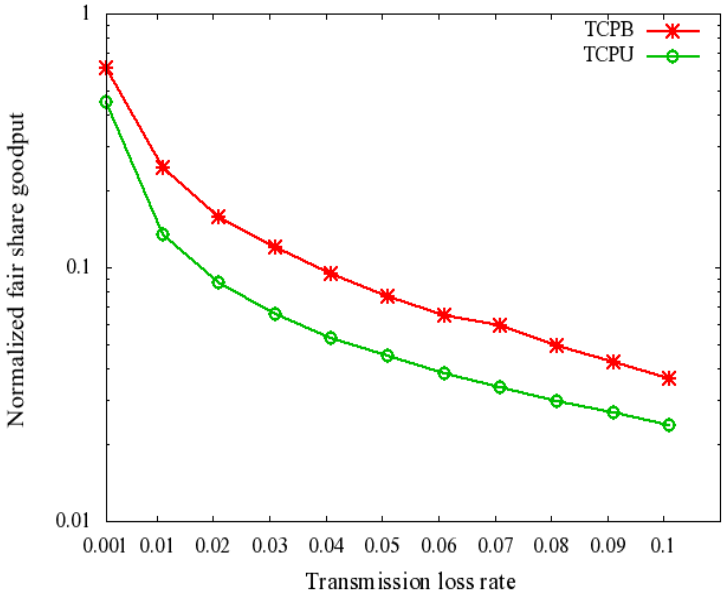

Figure 6.15: TCP performance under uniform and bursty transmission errors

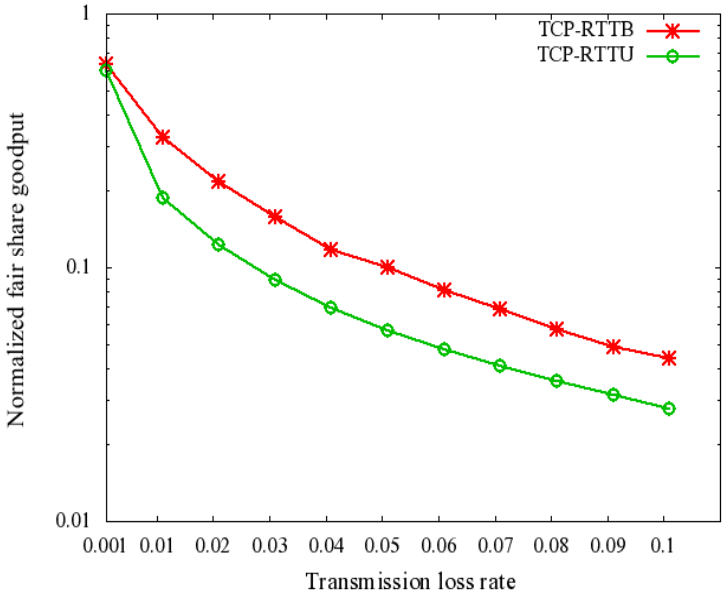

Figure 6.16: TCP-RTT performance under uniform and bursty transmission errors

compared to the bursty error model since the time between dropping events is longer.

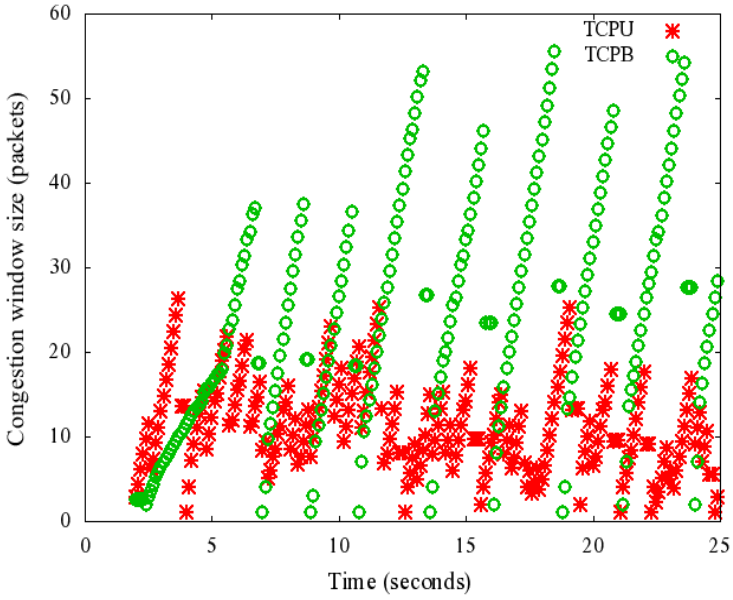

Figure 6.17: TCP congestion window size growth under uniform and bursty errors

In order to support our argument, in figure 6.17 we show a comparison between the TCP congestion window growth under uniform error model (TCPU) and the TCP congestion window growth under bursty error model (TCPB). The bursty model creates less dropping events than the uniform model so the congestion window has more opportunity to grow bigger before any dropping event occurs.

One finding from these results is that the frequency of dropping events can affect TCP performance more than number of drops per event.

This can be explained because after each dropping event (no matter how many packets are dropped) a maximum of one congestion window cut and one timeout can occur. This is because when TCP discovers the first packet drop (single or first one in a train), it will cut the congestion window and will then timeout if no acknowledgment is received. Then, TCP retransmits the whole window which will

recover the rest of the dropped packets.

So even if the number of dropped packets per error event is big the frequency of these drop events will cause more damage than the actual number of packets dropped.

Another observation is that under bursty errors, the number of timeouts is bigger. We can notice that also in figure 6.17 as TCPB frequently takes a small value of one, which indicates a timeout event. This is also expected because TCP can recover only one packet and when more than one packet is dropped, it triggers a timeout.

Finally, although most of the previous effects on TCP also apply to TCP-RTT, this did not affect the rate of improvement gained by TCP-RTT over TCP. Also, we can notice from figure 6.16 that the type of the error model has a smaller effect on TCP-RTT under low transmission rates (under 0.001% the difference between TCP-RTTU and TCP-RTTB is about 5% compared to a 35% difference between TCPB and TCPU). However with the increase in the transmission error rates, it seems the error model type affects both protocols similarly.

6.4 Impact of TCP-RTT on the Network

The addition of TWA to the error discriminator plays an important function in creating traffic with less variability compared to current error discriminators, which jump between two extremes: one is to cut the congestion window and the other is to avoid cutting at all. On the other hand, TWA will make TCP-RTT cut the congestion window at a rate that is related to the number of dropped packets. We anticipate that this technique will produce more stable connections which will have a positive effect on the network. Figure 6.18 shows a comparison between the congestion window oscillation of TCP-RTT and the Spike [1] error discriminator.

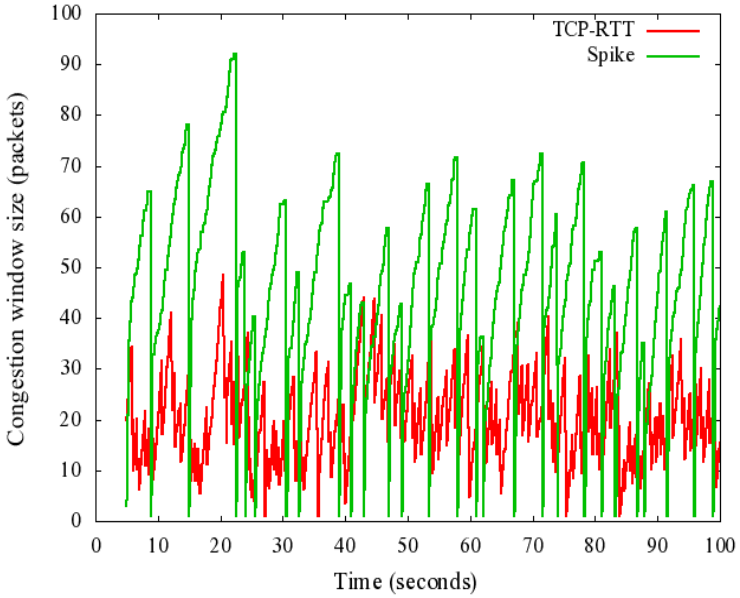

Figure 6.18: Congestion window variability

As we can see, TCP-RTT reduces the variability in the congestion window size, and hence the variability in the sending rate.

In the following section we will discuss the effect of TCP-RTT on the network in terms of congestion drop rate at the bottleneck, average queue size at the bottleneck and end-to-end delay. Also we will discuss the fairness of TCP-RTT when it shares the bottleneck with other flows.

6.4.1 Impact on Network Congestion Loss Rate

One aim of this study is to create a transmission drop action that will prevent or reduce the effect of actions taken by current error discriminators on the network congestion drop rate. For this, in this section, we will compare the network bottleneck congestion loss rate when using TCP-RTT (an error discriminator that has the transmission window actions TWA) with Spike (an error discriminator with no transmission action).

We measure the congestion drop rate, which is the number of packets dropped at the bottleneck divided by the total number of packets actually arrived at the bottleneck router. In our study we used a single bottleneck network as described in chapter 5.

6.4.1.1 Single Flow Case:

Here a single TCP-RTT connection will run concurrently with a mixed cross traffic of TCP and UDP connections. We compare the congestion loss rate of TCP-RTT with the congestion loss rate of the Spike [1] error discriminator. Also we use TCP in the comparison as a reference, so that the one (i.e. TCP-RTT or Spike) that has a closer congestion loss rate to standard TCP is preferable.

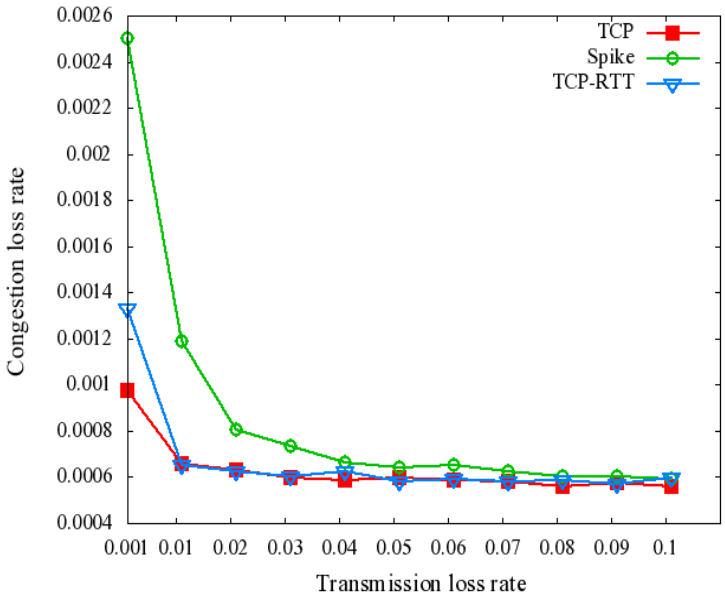

Figure 6.19: Network congestion loss rate (TCP, TCP-RTT and Spike)

Figure 6.19 shows that TCP-RTT has a much lower congestion loss rate compared to Spike, especially under transmission error rates from 0.001 to 0.04. Also starting from a 0.01 transmission error rate, the congestion loss rate of TCP-RTT

is as low as standard TCP, while Spike becomes closer to TCP starting from 0.05 transmission error rate.

Moreover, the reason that Spike reduces the congestion loss with the increase in the transmission error rate is that fewer packets are being sent due to the increase in the transmission error rate. Next, we will show how multiple flows from the same protocol (i.e. TCP-RTT and Spike) will affect the congestion loss rate.

6.4.1.2 Multiple Flows Case:

Here we will test the aggregate effect of multiple instants of the same flow (i.e. TCP, TCP-RTT or Spike) on the bottleneck congestion rate. This will give us an indication of how TCP-RTT works when it is used in a wider scale.

In this scenario we will not increase the bottleneck bandwidth with the increase in the number of flows in order to see the effect of the increase in the congestion loss rate with the increase in the number of flows.

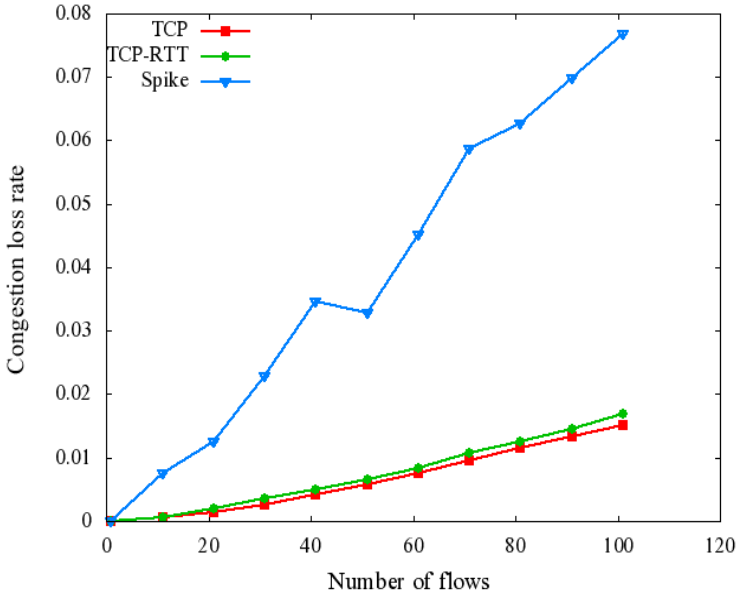

Figure 6.20: Network congestion loss rate for multiple flows

Figure 6.20 shows the congestion loss rate caused by increasing the number of flows. As we can see TCP-RTT has reduced the congestion loss rate to be as much as standard TCP in most cases. We must keep in mind that this reduction in congestion loss rate also comes with improvement in the performance, as we have seen in the previous experiments. The slight increase in the congestion loss rate with the increase in the number of flows is inevitable due to the fact that TCP-RTT will miss some congestion drops as transmission drops, and will not cut the congestion window as aggressively as TCP.

However, compared to Spike, it is clear that TCP-RTT has reduced the congestion loss rate noticeably. The transmission loss rate used in this experiment is 1%. Similar results were observed with different transmission error rates so we report only results with 1% transmission error rates.

6.4.2 End-to-end Delay

Here we present the end-to-end delay and the bottleneck queue size for TCP, TCP-RTT and Spike.

In figure 6.21 we present the end-to-end delay and in figure 6.22 we present the average queue size. As we can see in these figures, TCP-RTT was able to reduce the queue size, which resulted in a noticeable reduction in the end-to-end delay when compared with SpikeR.

However, the reason why TCP-RTT has higher delay than standard TCP in figure 6.21 is that, even though TCP-RTT has reduced the congestion loss rate when compared to other error discriminators like SpikeR, it still increases the congestion slightly when compared to TCP and hence increases the queue size. However, as we saw before, TCP-RTT outperforms standard TCP and at the same time it does not increase the congestion loss rate to very high levels. Also, from these results we can

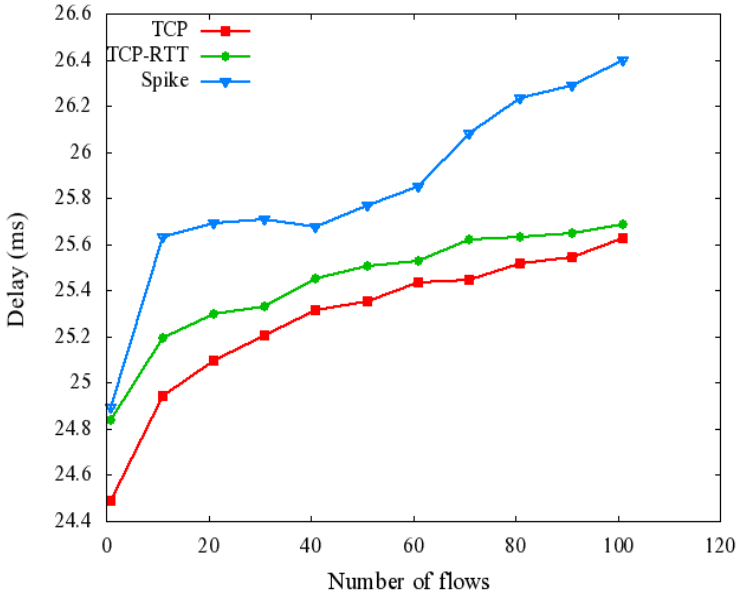

Figure 6.21: end-to-end delay

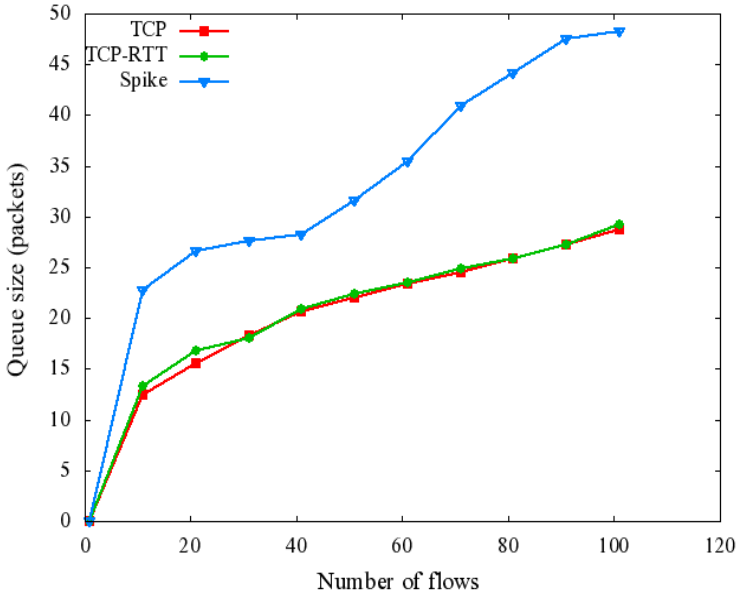

Figure 6.22: Average queue size

see that TCP-RTT reduces the end-to-end delay and the average queue size when compared with SpikeR.

Finally, in figure 6.22 we use the forward path end-to-end delay. We used the forward path delay because the congestion takes place at the forward path only (from sender to receiver).

6.4.3 TCP-RTT Fairness

What we mean by fairness here is when TCP-RTT allows other connections competing with it on the bottleneck to have a fair share of the bottleneck bandwidth.

Standard TCP provides fairness by responding to congestion signs and reducing its transmission rate. However, if the error discriminator cannot respond to congestion drops then it will not reduce its sending rate, which will cause other flows to throttle back constantly until most or even all bandwidth is eaten by the error discriminator protocol.

However, we avoid this by making the error discriminator slow down for congestion and transmission drops as well. Using this technique, TCP-RTT could provide improvement to TCP performance but we still need to test its fairness toward other traffic.

In order to do that we use the Jain fairness index [34], as follows:

$$F(X) = \frac{(\sum_{i=1}^{m} X_i)^2}{m(\sum_{i=1}^{m} X_i^2)} \tag{6.5}$$

where m is the number of competing TCP sources sharing the same bottleneck, and X_i is the goodput (or throughput) for source i. So if there are m TCP sources sharing the bottleneck in a fair manner then F(x) should be close to 1.

In figure 6.23 we show the fairness index for TCP-RTT and Spike. As we can

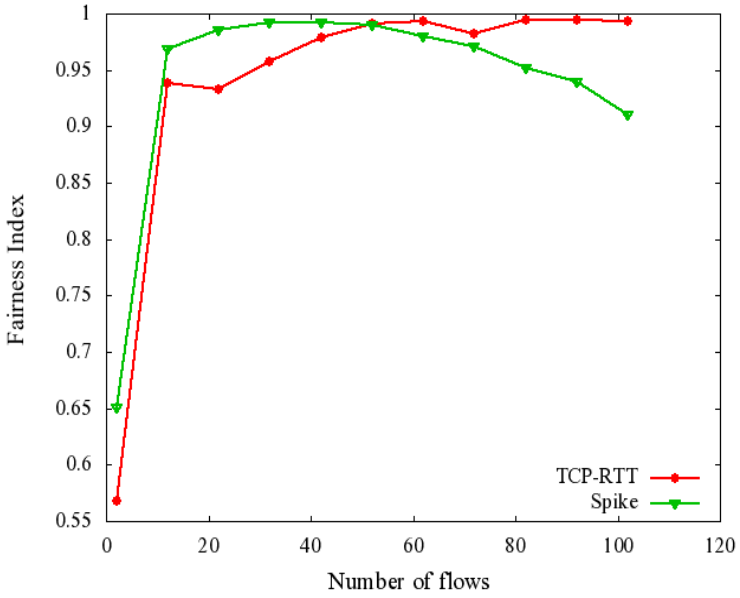

Figure 6.23: Fairness index - 180Mb bottleneck

see, TCP-RTT starts with lower fairness than Spike, however, with the increase in the number of flows and hence the increase in congestion loss rate, TCP-RTT gained higher fairness. On the other hand, Spike tends to have decreasing fairness with the the increase in number of flows.

The reason why the fairness is low when number of flows is only two (first point in the figure) is because TCP-RTT is not able to achieve high performance since it suffers from transmission errors, unlike the second flow which does suffer from small or no errors because of the small number of flows. However, with the increase in number of flows all flows suffers from drops caused by the congestion.

It seems that TCP-RTT gets better in terms of fairness with the increase in the congestion loss rate (represented here by increased number of flows). To show this, we repeated the previous experiment but this time we decreased the bottleneck bandwidth so that the congestion loss rate increases from 0.006 to 0.07 on average (we increased the congestion loss rate by reducing the actual bottelneck bandwidth

from 180Mb to 36Mb). Figure 6.24 shows the fairness comparison between TCP-

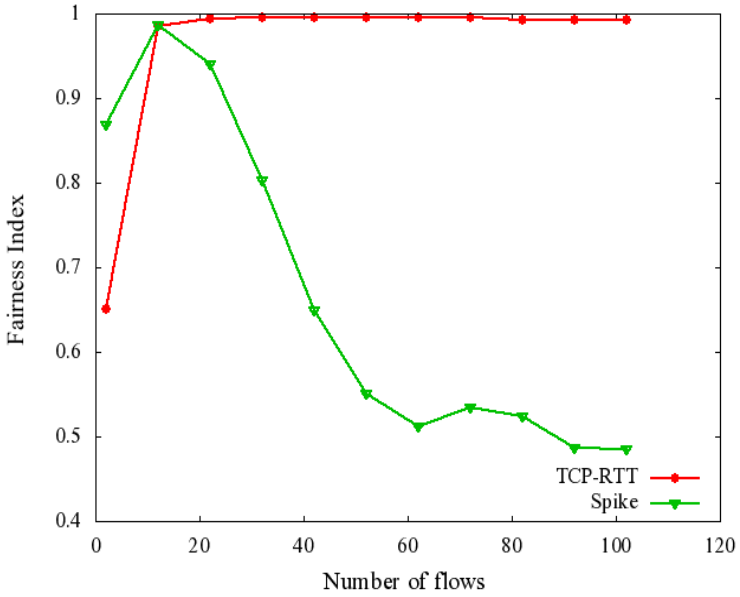

Figure 6.24: Fairness index - 36Mb bottleneck

RTT and Spike with the new bottleneck. As we can see, TCP-RTT has a higher fairness index with the increase congestion loss rate.

TCP-RTT was able to achieve higher fairness because it responds to all error types (congestion and transmission errors), while Spike only responds to what appears to be congestion errors. However, Spike may miss some congestion errors which will prevent it from reducing its sending rate in a genuine congestion situation. Also, when a new flow starts sending, Spike may not give it the opportunity to take a fair share from the bottleneck because Spike may miss the signals being sent from the network indicating new flows (these signals are packet drops).

This is a good indication that the improvement gained by TCP-RTT has minimal effect on the network and other individual traffic. This is confirmed by the fairness results, which show that TCP-RTT is far from being greedy.

Notes:

Since we did not find any published implementation of Spike [1], we implemented

Protocol	Throughput improvement
Spike [1]	80%
Our implementation	77.7%

Table 6.1: Spike Throughput improvement

two versions: one based on TCP-Reno (SpikeR) which we used above and another based on TCP-NewReno (SpikeNR), which we will use later in chapter 7. We then validate our implementation by comparing the performance of SpikeNR with the results published in [1]. The comparison shows that our implementation of Spike gives similar results in terms of throughput improvement over TCP with the results presented in [1]. Table 6.1 shows our comparison. The slight difference in the results could be due to differences in some unknown variables in the experimental settings. For example the exact value of transmission error rate is not presented in [1], the author indicated that it approaches 0.03. However, the values we present in table 6.1 are for an error rate of 0.031.

Moreover, the results reported in this section were produced using the uniform error model for the transmission errors. We found that SpikeR does not work as expected when the bursty error model is used. The reason for this is that under bursty errors, the performance will be dominated by the high number of timeout events. This is because Reno can recover one packet only and leave the rest to be recovered by timeouts. Thereby, SpikeR will not be able to improve the performance since after each timeout event the congestion window will be cut to the minimum.

Later in chapter 7 we will add Spike to NewReno which will allow us to use bursty errors since NewReno is able to recover multiple drops.

6.5 Modeling TCP-RTT Behaviour

In this section we will present an analytical model to approximate TCP-RTT behaviour. The main attribute the model will try to capture is the congestion window behaviour since it governs the TCP-RTT sending rate. We will derive our model from a well-known model to approximate TCP behaviour presented in [102]. Other authors, such as in [3], have studied this model and explained the reasoning that leads to this model in a fairly easy way so we will follow similar reasoning in advising our model. The model proposed by [102] to approximate the TCP sending rate is:

$$Averagesendingrate = \frac{1}{RTT} \times \sqrt{\frac{3}{2p}} \tag{6.6}$$

where RTT is the round trip time and p is the average drop rate. This model is known as the *inverse square root p* law [3]. The difference between this model and our model will be that the model in 6.6 assumes standard TCP cut mechanism where the congestion window is cut by 50% after each drop. However, TCP-RTT uses CWA to cut the congestion window after each transmission drop. As described in chapter 4, CWA only cuts the congestion window size by the number of dropped packets from each window which depends on the error rate in the connection path. So in our model, if the congestion window is W and the error rate is p, then the congestion window cut will be a function of current congestion window and the the error rate $f(W, p)$. However, for simplicity, we will assume that the amount of congestion window cut depends on the error rate directly so that $f(W, p) = W * p$

In CWA after each drop the window size is decreased as follows:

$$nW = W - ndp. \tag{6.7}$$

where nW is the new window size, W is the old window size and ndp is the number of dropped packets. Since ndp depends on the error rate, nW can be computed as follows:

$$nW = W - f(W, p) \tag{6.8}$$

$$nW = W - (W * p) = W(1 - p) \tag{6.9}$$

This means that after each drop, CWA cuts the congestion window by $W * p$.

Now according to [3], it is safe to assume that: in the steady state if the error rate is p, then we expect to have $1/p$ packets on average to be sent before the next drop occurs.

Also, after each cut the time needed to increase the window from $W(1 - p)$ to W is $RTT * W * p$ (i.e. since the amount of the last cut is $W * p$ packets, then we need $W * p$ round trip time to return to original window size before the cut).

Figure 6.25 shows the typical behaviour of the congestion window when the

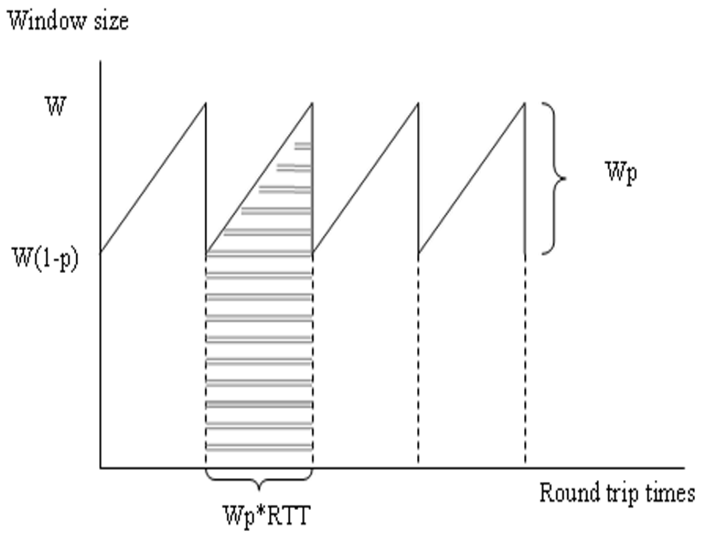

Figure 6.25: Congestion window increase/decrease behaviour

cut factor is $W * p$. From the figure 6.25, we can compute the area under the congestion window increase slope by computing the area of the trapezoid created by the congestion window increase/decrease behaviour. This area will give us the number of packets sent before the drop occurred, as follows:

$$\frac{W * p(W + W * (1 - p))}{2} \tag{6.10}$$

$$= \frac{W * p(2W - W * p))}{2} \tag{6.11}$$

$$= \frac{W * p(W(2 - p))}{2} \tag{6.12}$$

$$= \frac{W^2 * p(2 - p)}{2} \tag{6.13}$$

The equation 6.13 will give us the number of sent packets during the period from $W * (1 - p)$ to W, which is equal to $1/p$ as we said before. So we can compute the window size W, as follows:

$$\frac{W^2 * p(2 - p)}{2} = 1/p \tag{6.14}$$

$$\frac{W^2}{2} = \frac{1/p}{p(2 - p)} \tag{6.15}$$

$$W = \sqrt{\frac{2/p}{p(2 - p)}} \tag{6.16}$$

$$W = \sqrt{\frac{2}{p^2(2 - p)}} \tag{6.17}$$

As we said before, $1/p$ is the total number of packets transmitted before errors occur. Also the time needed to transmit these packets is $W * p * RTT$. So from this,

the TCP-RTT sending rate (SR) can be computed using the following equation :

$$SR = \frac{1/p}{RTT * W * p} = \frac{1}{RTT * W * p^2} \tag{6.18}$$

By substituting W from equation 6.17 we get:

$$SR = \frac{1}{RTT * \sqrt{\frac{2}{p^2(2-p)}} * p^2} = \frac{1}{RTT * p}\sqrt{\frac{2-p}{2}} \tag{6.19}$$

6.5.1 A More General Model

The model described above captures the congestion window behaviour in TCP-RTT (i.e CWA) when there are only transmission drops. However, there are other factors that affect the congestion window size namely drops caused by congestion and timeout events. Each congestion drop will reduce the congestion widow by 50%, and each timeout event will reduce the congestion window to minimum size (usually one segment). We will generalize the model in equation 6.19 to include these effects.

First we will define the effective cut rate (ECR) of the congestion window. In ECR we try to combine the effect of the CWA (transmission drops), the congestion drops and the timeout events. To compute the ECR we need the number of packets diagnosed as transmission drops Td, the number of packets diagnosed as congestion drops Cd and the number of timeout events Te and then we compute the average cut rate as follows:

$$ECR = \frac{p * Td + 0.5 * Cd + 1 * Te}{Td + Cd + Te} \tag{6.20}$$

Where $p * Td$ represents the cut rate of the CWA, $0.5 * Cd$ represents the cut rate of the congestion drops and $1 * Te$ represents the cut rate of timeout events.

And we divide it by the total number of cuts ($Td + Cd + Te$). This will give us an estimation of the congestion window average cut rate during the connection life time.

Now we follow the same reasoning that we used to compute equation 6.19 but this time we use ECR instead of p in $f(W, p)$ so that the the congestion window cut factor is now $f(W, ECR)$.

From equation 6.14 we use ECR as follows:

$$\frac{W^2 * ECR(2 - ECR)}{2} = 1/p \tag{6.21}$$

$$\frac{W^2}{2} = \frac{1/p}{ECR(2 - ECR)} \tag{6.22}$$

$$W = \sqrt{\frac{2/p}{ECR(2 - ECR)}} \tag{6.23}$$

$$W = \sqrt{\frac{2}{p * ECR(2 - ECR)}} \tag{6.24}$$

As we can see, we still use $1/p$ to calculate the total number of packets transmitted before an error occurs. Now the TCP-RTT sending rate (SR) can be computed using the following equation:

$$SR = \frac{1/p}{RTT * W * ECR} = \frac{1}{RTT * W * p * ECR} \tag{6.25}$$

By substituting W from equation 6.24 we get:

$$SR = \frac{1}{RTT * \sqrt{\frac{2}{p*ECR(2-ECR)}} * p * ECR} = \frac{1}{RTT}\sqrt{\frac{2 - ECR}{2 * p * ECR}} \tag{6.26}$$

Equation 6.26 is an approximation of the sending rate of TCP-RTT and it is

not supposed to give an exact estimation but to give an average estimation of the sending rate.

In figures 6.26, 6.27 and 6.28 we compare the results from using the equation 6.26 with average runs from simulations of TCP-RTT with increasing RTT and transmission error rates of 1%, 5% and 10% and with no congestion in the path (although the effect of congestion drops is still included because the error discriminator mismatches some transmission errors as congestion drops). As we can see, the model gives a good approximation of the average sending rate of TCP-RTT with different error rates and different RTT values.

However, in some cases the analytical and experimental results differ slightly in the figures. This is due to the fact that, although we tried to include many aspects of TCP-RTT performance in the model, we did not include them all.

For example, at high error rates such as 10% (figure 6.28) the proposed model seems to underestimate TCP-RTT performance. We think the reason for this is that, with the increase in the error rate, the effect of the multiple drops action MDA and the retransmission timeout action RTA increases. This in turn is because of the increase in the number of packets dropped and the increased chance that the same packet will be dropped more than once, and hence increasing the back-off level of TCP. However, these extra effects have not been included in the proposed model and are left for future work.

6.6 Summary

In this chapter our aim was to add the proposed transmission window actions (TWA) to an error discriminator and to test its performance. We presented a new error discriminator called TCP-RTT and we added TWA to implement its action in the

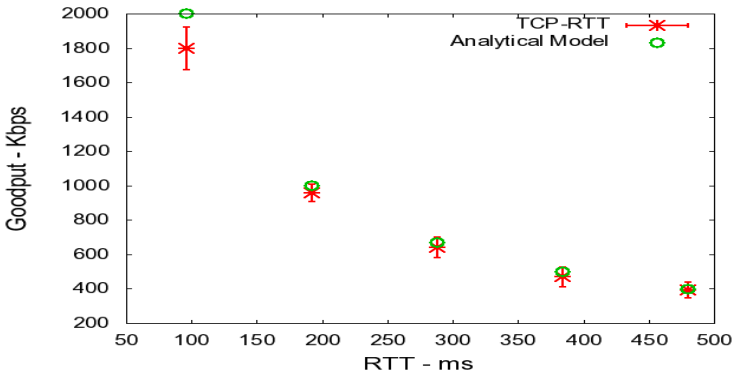

Figure 6.26: Goodput - increasing RTT - 1% error rate - TCP-RTT and Analytical model

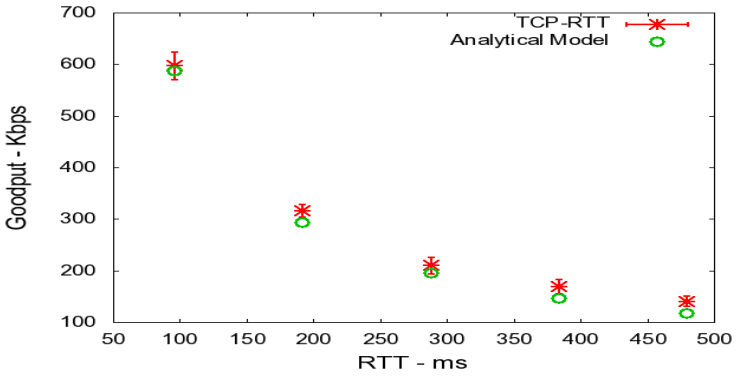

Figure 6.27: Goodput - increasing RTT - 5% error rate - TCP-RTT and Analytical model

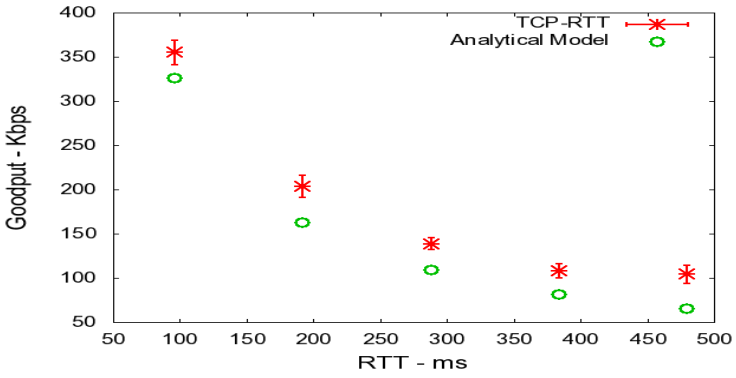

Figure 6.28: Goodput - increasing RTT - 10% error rate - TCP-RTT and Analytical model

case of transmission errors. In the case of congestion errors TCP-RTT acts like normal TCP.

Throughout the chapter we tested the performance of TCP-RTT to see if it can achieve two aims. First to see if it outperforms TCP in terms of goodput which is the aim of any error discriminator, and second to see if it will reduce any increase in the network congestion loss rate compared to existing error discriminators.

By comparing TCP-RTT with standard TCP we showed that TCP-RTT actually is able to gain higher goodput than TCP with different transmission error rates. Also by comparing TCP-RTT with the Spike error discriminator, it is clear that TCP-RTT has reduced the congestion loss rate noticeably. So it improves the TCP performance with less effect on the network.

The addition of TWA to the error discriminator plays an important role in creating traffic with less variability compared to current error discriminators, which jump between two extremes: cut congestion window or avoid cutting at all. On the other hand, TWA will make TCP-RTT cut the congestion window whenever drops occur at a rate which depends on the error type. We believe that this technique has produced more stable flows, which has a positive effect on the network.

Also, we showed that TCP-RTT has a high fairness index in sharing bottleneck bandwidth in our topology with an increased numbers of flows.

One other result from this chapter is that the error patterns (bursty or non-bursty) can affect TCP-RTT performance noticeably. We saw that the frequency of dropping events can affect TCP-RTT performance more than the number of drops per event.

Finally, in an effort to gain more understanding of how TCP-RTT works, we proposed an analytical model to approximate the TCP-RTT sending rate under different error rates and round trip times. The results from the analytical model

have been compared with the experimental results and they show consistency and similarity with each other.

Chapter 7

Allowing Multiple Drops Action for Congestion Losses

7.1 Introduction

In general, error discriminators use different sets of actions (contradictory to some extent) when they diagnose transmission drops different from actions used in the case of congestion losses. Because of this, in the previous experiments TCP-RTT used the CWA, MDA and RTA actions only when the errors were diagnosed as transmission errors.

The MDA action is concerned with resending multiple transmission drops from the same window. However, authors like [103] indicated that due to the bursty nature of TCP traffic, caused by reasons like slow start and delay acknowledgments which lead to injecting the network with sudden bursts of data, multiple packet drops per window can be caused also by congestion in the network.

Because of this, we suggest here the use of our multiple drops action MDA in the case of both congestion and transmission drops; we expect it to give TCP-RTT

more gain in terms of performance (goodput). However, we will study the impact of this change on the network and fairness toward other flows.

7.2 Results

In the following experiments the error discriminator uses MDA whether the error is considered congestion or transmission. The CWA and the RTA actions are used only for transmission errors. The experiment settings and assumptions are the same as in section 6.3.1 with a minimum round trip delay of 12ms. We call the new approach TCP-RTTM. In the following experiment TCP-RTTM will be subjected to both congestion and transmission errors.

As we can see in figure 7.1, MDA action to congestion drops has improved

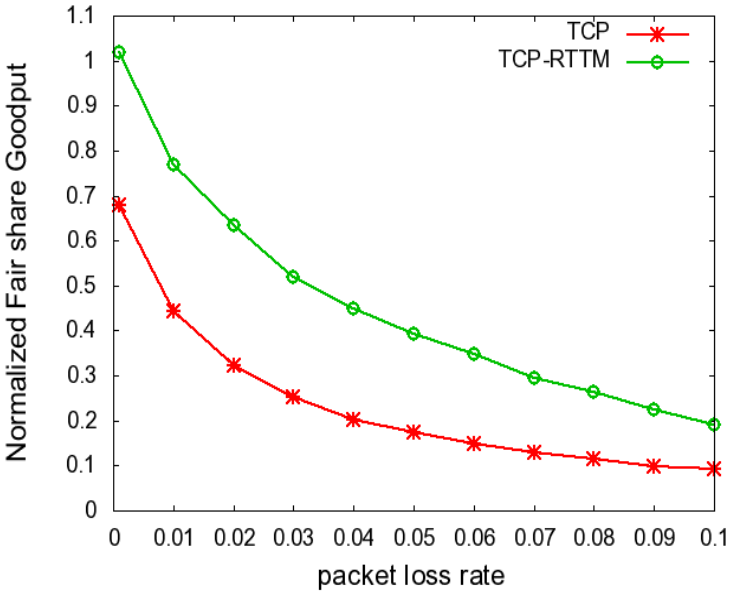

Figure 7.1: TCP vs. TCP-RTTM normalized fair share goodput

the performance of TCP-RTTM noticeably over standard TCP. Under small error rates TCP-RTTM seems to take slightly more than its fair share goodput. This raises question about TCP-RTTM fairness which we will study later in this chapter.

However, as we can see in figure 7.1, the increase in the fair share is small (the increase is only 0.02)

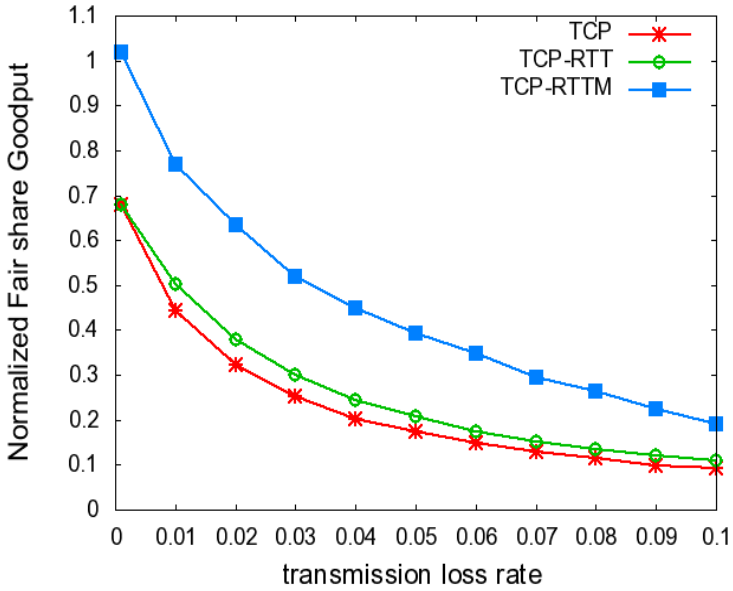

Figure 7.2: TCP vs. TCP-RTT vs. TCP-RTTM normalized fair share goodput

Also, figure 7.2 shows a comparison between TCP-RTT and TCP-RTTM, and as we can see that TCP-RTTM improves the performance under low error rates (unlike TCP-RTT). This is because under low error rates the main factor that affects the performance is the congestion drops, and TCP-RTTM can handle them better than TCP-RTT.

Using MDA will reduce the chance of falling into timeout because TCP-RTTM will try to resend all drops packets, not only for transmission drops but also for congestion drops. This is the main factor that has led to the higher performance of TCP-RTTM. This improvement continues with the increase in the transmission error rate.

Another factor that has made adding MDA invaluable is related to the fact that TCP-RTT suffers from slightly higher congestion drop rates compared to standard

TCP. This is because of the increase in the retransmission rate and the increase in the congestion window (sending rate) caused by using CWA and MDA in the case of transmission errors. So because of this increase in the congestion adding MDA, which retransmits all lost packets from the same window, has reduced the effect of this increase on TCP-RTTM performance.

In this experiment the congestion error rate is around 0.01.

7.3 Impact of TCP-RTTM on the Network

Here we will discuss the effect of TCP-RTTM on the network performance (Congestion drop rate, end-to-end delay and bottleneck average queue size), and on other flows that share the bottleneck.

7.3.1 Impact on Network Congestion Loss Rate

One aim of this study is to create a transmission drop action that will prevent or reduce the effect of actions taken by current error discriminators on the network congestion drop rate. For this reason, in this section, we will compare the effect of TCP-RTTM and the SpikeNR error discriminator on the congestion loss rate. The SpikeNR is an end-to-end error discriminator based on the NewReno version of TCP. The reason for comparing with SpikeNR is that it is capable of handling multiple packet drops in case of congestion just like TCP-RTTM.

7.3.1.1 Single Flow Case

Here a single TCP-RTTM connection will run concurrently with a mixed cross traffic of TCP and UDP connections. We compare the congestion loss rate of TCP-RTTM and the SpikeNR [1] error discriminator.

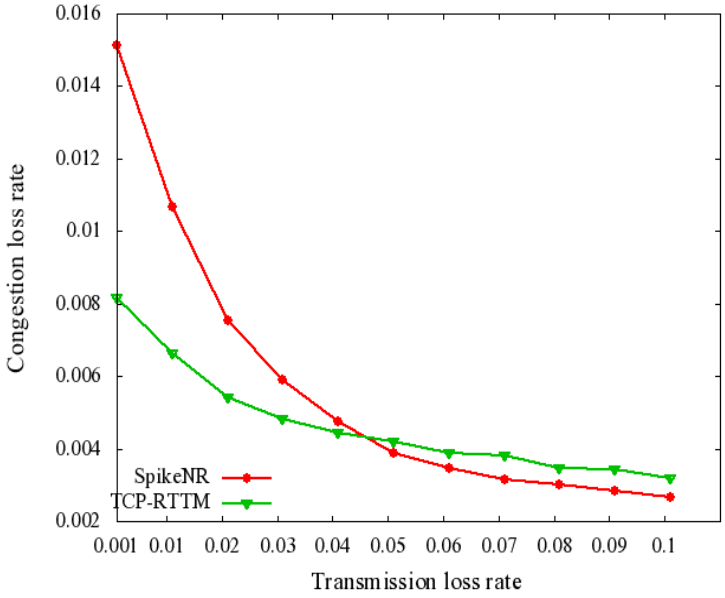

Figure 7.3: Network congestion loss rate (TCP-RTT and Spike)

Figure 7.3 shows that TCP-RTTM has a lower congestion loss rate compared to SpikeNR, especially under transmission error rates from 0.001 to 0.04.

However, the congestion loss rate of SpikeNR becomes less than TCP-RTTM when the transmission loss rate exceeds 4%. We think this is caused by SpikeNR falling into longer timeout events caused by high transmission loss rate and hence less packets are being sent. This is supported by the fact that we found SpikeNR has a lower sending rate than TCP-RTTM under such error rates.

Next we will show how multiple flows from the same protocols (i.e. TCP-RTTM and SpikeNR) will affect the congestion loss rate.

7.3.1.2 Multiple Flows Case

Here we will test the aggregate effect of multiple instants of the same flow (i.e. TCP-RTTM or SpikeNR) on the bottleneck congestion rate. This will give us an indication how TCP-RTTM works when it is used on a wider scale.

In this scenario we will not increase the bottleneck bandwidth with the increase in the number of flows in order to increase in the congestion rate with the increase in the number of flows.

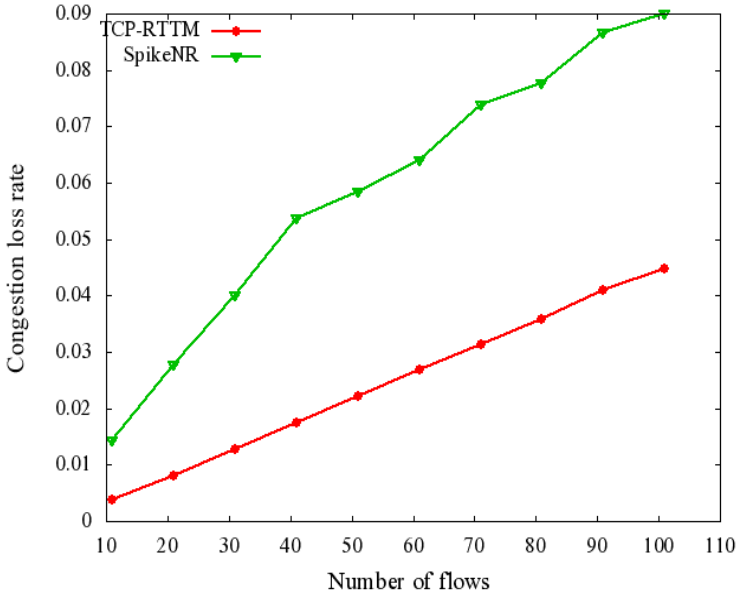

Figure 7.4: Network congestion loss rate for multiple flows

Figure 7.4 shows the congestion loss rate caused by increasing the number of flows. As we can see, TCP-RTTM has reduced the congestion loss rate compared to SpikeNR. The increase in the congestion loss rate with the increase in the number of flows is inevitable due to the fact that TCP-RTTM will miss some congestion drops as transmission drops and will not cut the congestion window as aggressively as TCP.

However, compared to SpikeNR it is clear that TCP-RTTM has reduced the congestion loss rate noticeably. The transmission loss rate used in this experiment is 1%. Similar results were observed with different transmission error rates so we report only the results with 1% transmission error rates.

7.3.2 End-to-End Delay

Here we show the end-to-end delay and bottleneck queue size when we use TCP-RTTM and SpikeNR.

In figure 7.5 we show the average queue size. As we can see in the figures TCP-

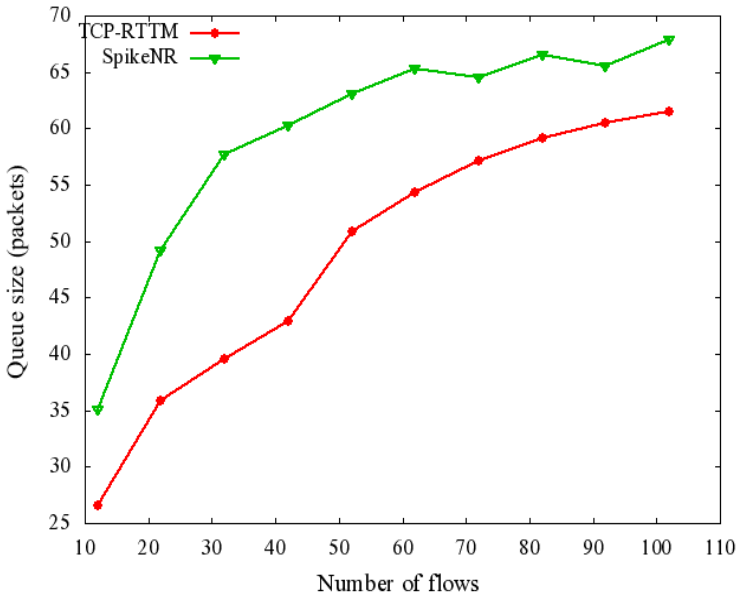

Figure 7.5: Average queue size

RTTM was able to reduce the queue size, which has resulted in a reduction in the end-to-end delay compared to SpikeNR as we can see in figure 7.6. In figure 7.6 we use the forward path end-to-end delay because the congestion takes place in the forward bath (from sender to receiver).

7.3.3 TCP-RTTM Fairness

As we did in TCP-RTT, for TCP-RTTM we will use the Jain fairness index [34] to compare the fairness of TCP-RTTM and SpikeNR.

As we can see in figure 7.7, TCP-RTTM has a constant high fairness index. On the other hand, SpikeNR tends to have decreasing fairness with the increase in the

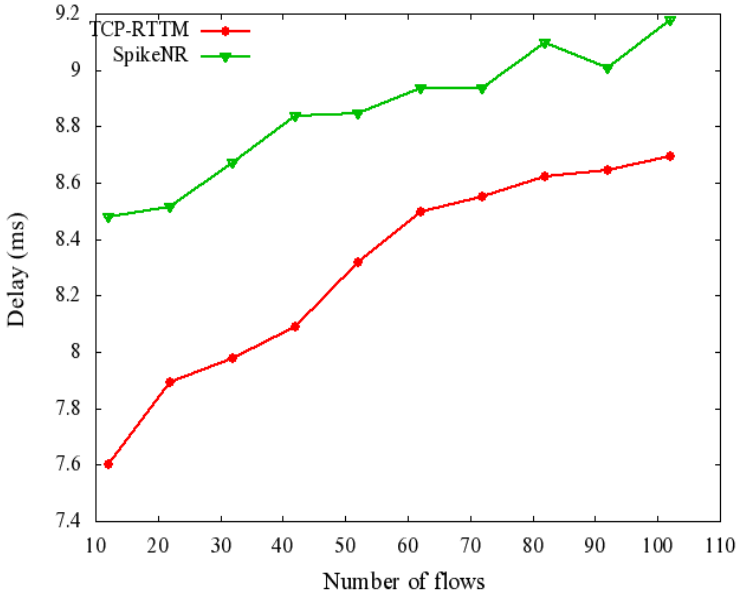

Figure 7.6: end-to-end delay

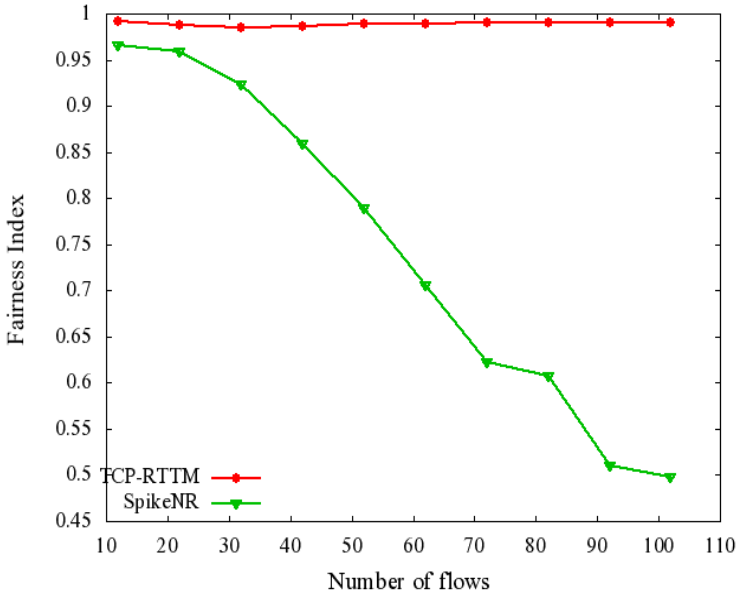

Figure 7.7: Fairness index - TCP-RTTM and SpikeNR

number of flows.

TCP-RTTM has been able to achieve high fairness because it responds to all error types (congestion and transmission errors) while SpikeNR only responds to what appears to be congestion errors. However, SpikeNR may miss some congestion errors, which will prevent it from reducing its sending rate in a genuine congestion situation. Also when a new flow starts sending, SpikeNR may not give it the opportunity to take a fair share from the bottleneck because it may miss the signals being sent from the network indicating new flows (these signals are packet drops).

This is a good indication that the improvement gained by TCP-RTTM has no negative effect on the network throughput or other individual traffic goodput. Also, this indicates that any increase in the network congestion level caused by the throughput increase of TCP-RTTM is actually a result of increased network resource utilization rather than the greediness of TCP-RTTM. This is confirmed by the fairness results of TCP-RTTM.

7.3.4 Cedge Configuration - Varying *midalpha*

Both TCP-RTT and TCP-RTTM use the equation 6.3 to calculate the value of *Cedge* and then use this value to determine the error type (congestion/transmission error) based on whether the current RTT value is higher or lower than *Cedge*.

As we explained in chapter 6, Cedge is a value between minRTT and maxRTT and the value of midalpha will determine how close Cedge is to the minRTT or maxRTT (see equation 6.3). When midalpha increases the Cedge in equation 6.3 will move toward the maxRTT and when midalpha decreases Cedge will move toward minRTT. This feature has an important role in the discriminator function because any increase in the Cedge value will tend to make the discriminator classify more errors as transmission errors and any decrease in Cedge will make the discriminator

classify more errors as congestion.

In this section we will see the effect of different *midalpha* values. We will test the performance of TCP-RTTM when *midalpha* takes the values: 0.15, 0.25 and 0.50. We choose these values because 0.15 has been found from previous experiments to give minimum impact on the network. The 0.25 and 0.5 values represent the quarter and half way points between minRTT and maxRTT.

Figures 7.8, 7.9 and 7.10 show the performance improvement gained by TCP-RTTM over standard TCP, and at the same time they show the impact on the network in terms of the increase in the congestion drop rate and the increase in the average queue size at the bottleneck. As we can see, above 1% transmission error rate the improvement in performance is more than the increase in the congestion drop rate and the increase in the queue size. This indicates that under most error rates TCP-RTTM adds improvement to TCP performance over and above the negative impact on the network in terms of the increase of the congestion drop rate and the increase in queue size. This is true for different values of *midalpha* (0.15, 0.25 and 0.5).

At very low error rates (0.001) the goodput improvement is less than the increase in congestion drop rate, especially with higher *midalpha* (for example *midalpha* = 0.5 in figure 7.10). The reason is that under low transmission error rates the main factor affecting the performance is the congestion drop rate, which is in this case around 1%. With *midalpha* = 0.50, TCP-RTTM tends to classify more packets as transmission errors and so the chance of misclassifying congestion drop as transmission drops will increase, and this will increase the congestion in the bottleneck. This explains why we see a higher congestion drop rate in figure 7.10.

However, even with high increases in the congestion drop rate in the case of *midalpha* = 0.50 (reached 90%), the actual congestion drop rate value is still not

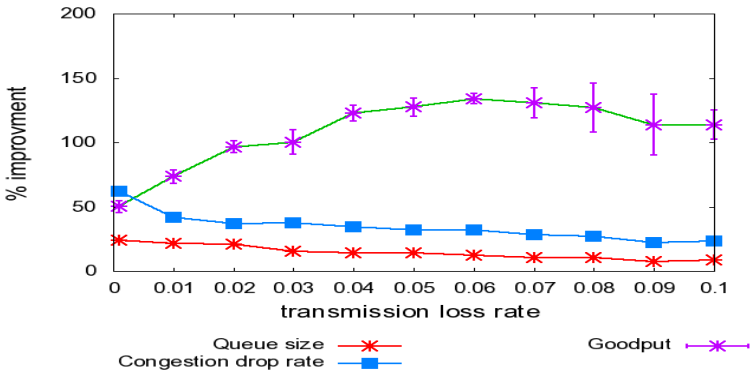

Figure 7.8: Improvement in goodput and impact on the network-$midalpha = 0.15$

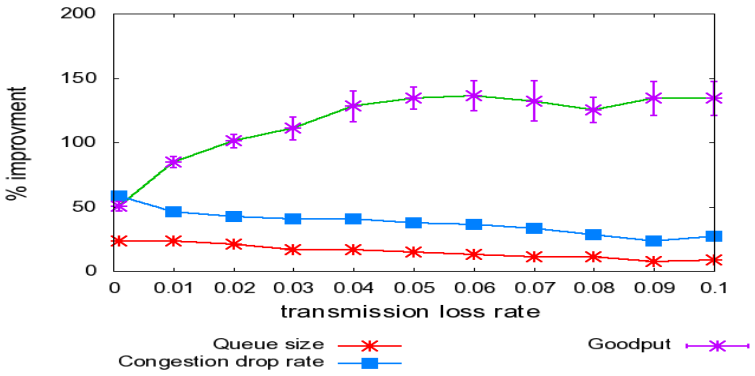

Figure 7.9: Improvement in goodput and impact on the network-$midalpha = 0.25$

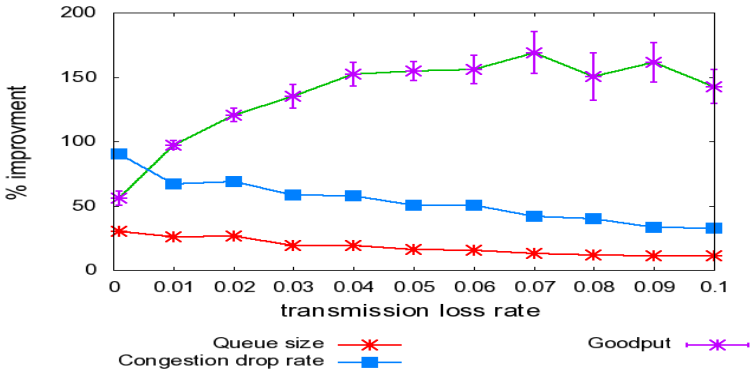

Figure 7.10: Improvement in goodput and impact on the network-$midalpha = 0.50$

more than 3.3% (the congestion drop rate for TCP is 1.7%), and also it decreases with the increase in the transmission error rate. Figure 7.11 shows the actual values

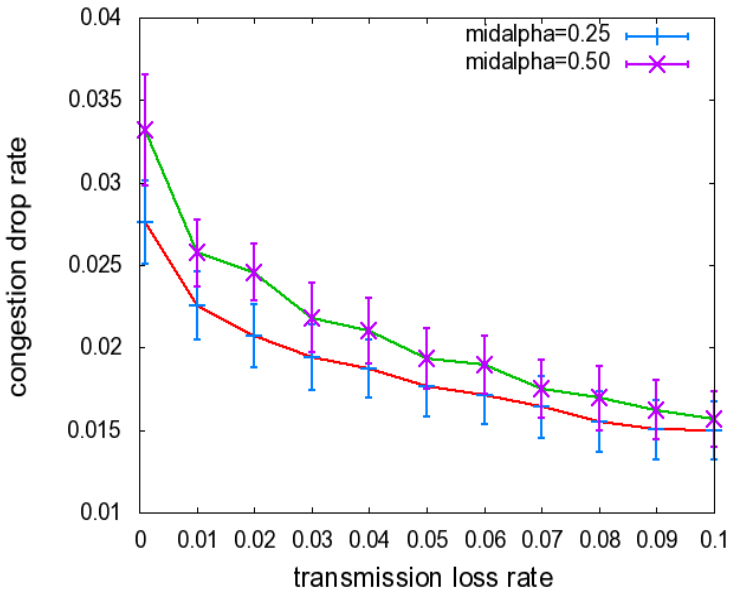

Figure 7.11: Congestion drop rate

for congestion drop rates for $midalpha = 0.25$ and $midalpha = 0.5$ (congestion drop rates in the case of $midalpha = 0.15$ are very close to $midalpha = 0.25$ so we did not include them in the graph).

When we increase *midalpha* the increase in the queue size is small in all cases and so is the packets delay at the bottleneck.

7.4 Different RTT and Bandwidth Values

TCP-RTTM uses a delay based error discriminator that uses the RTT increase/decrease to decide the error type. Also the RTA mechanism in TCP-RTT estimates the available bandwidth in order to compute the back-off level during the timeout events. So in this section we will vary the RTT and the bandwidth values to see how this affects the performance of TCP-RTTM.

7.4.1 Different RTT Values

Figure 7.12 shows the performance of TCP and TCP-RTTM with different RTT

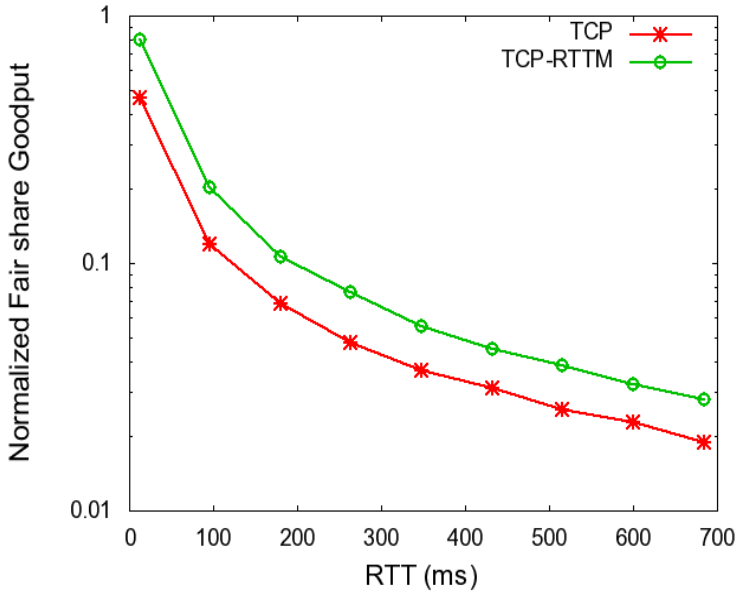

Figure 7.12: Performance with different RTT values - semi-log scale normalized fair share goodput

values. The performance of TCP and TCP-RTTM depends heavily on the RTT value because with the increase in RTT, TCP and TCP-RTTM need more time to increase congestion window size and hence errors will have a larger effect in higher RTT values. However, these results show that TCP-RTTM still outperforms TCP for different RTT values. In figure 7.12 both TCP and TCP-RTTM suffers from congestion error rates of around 0.005 and also from a transmission drop rate of 0.01.

7.4.2 Different Bandwidth Values

In this experiment the bandwidth takes three different values according to three widely used standards T1(1.5Mbps), T2(6.5Mbps) and T3(45Mbps) carriers [2]. The

transmission error rate is 0.01 and the congestion error rate is around 0.005. The RTT is around 124ms, which is the average presented in figure 5.2 in section 5.2.3. Figure 7.13 shows that the improvement increases with the increase in the available

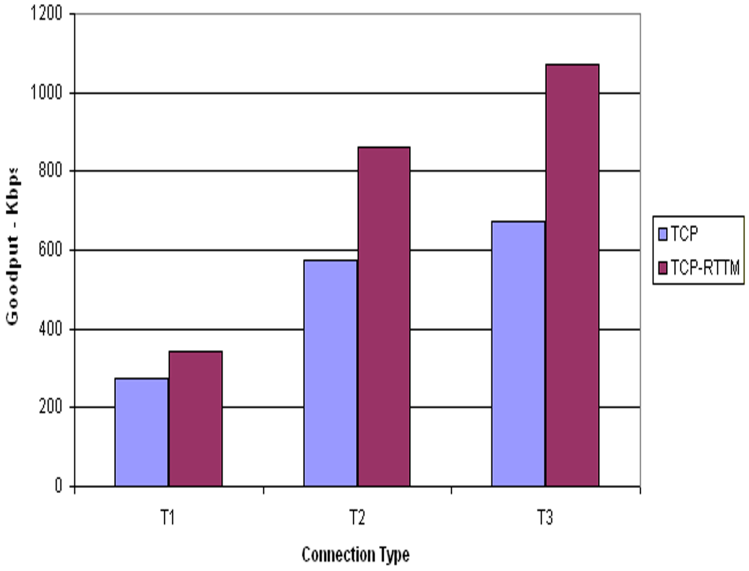

Figure 7.13: Performance with different bandwidths

bandwidth since there is more room for retransmission. Also, the increase in the bandwidth will give the window more room to increase, however the drop rate will limit this increase and because of this both TCP and TCP-RTTM could not fully utilize the available bandwidth.

7.5 The Effect of Error Burst Size

TCP-RTTM showed performance improvement over standard TCP. However, in this section we will compare TCP-RTTM with some of the protocols presented in chapter 3.

By this comparison we will see how TCP-RTTM uses the MDA algorithm to respond to different transmission error burst sizes. This is important because with

the increase in the burst size it will be harder for TCP to recover from drops because TCP cannot recover from multiple drops from the same window and will timeout after the first drop.

However, using the MDA technique, TCP-RTTM is able to improve TCP performance with increasing error burst sizes and also with existing congestion (congestion drops around 1%).

Figure 7.14 shows a comparison between TCP-NewReno [37] (NReno in the

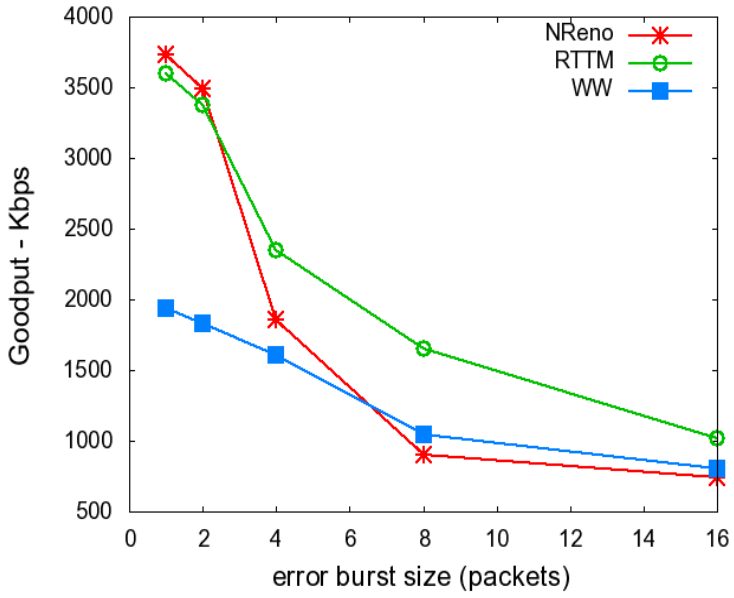

Figure 7.14: Performance with different transmission error burst size

figure) , TCP-Westwood [53] (WW in the figure) and TCP-RTTM (RTTM in the figure). As we explained in chapter 3 TCP-NewReno is able to recover from multiple congestion errors from the same window of data, whereas TCP-Westwood is aimed at improving TCP performance, especially for transmission errors.

As we can see in figure 7.14 when the error burst size is small (1 and 2 packets per burst of errors) the performance of TCP-RTTM and TCP-NewReno are close. However it appears that the TCP-NewReno multiple drops retransmission technique

has slightly higher performance over MDA with small burst size (1 and 2 packets). On the other hand, we can see that when the burst size increases (4,8 and 16 packets) TCP-NewReno cannot cope and TCP-RTTM gives better performance. We think this is because TCP-NewReno only resends one packet per RTT, however, when multiple packets are dropped, the timeout mechanism will work before TCP-NewReno is able to resend all dropped packets.

Also in figure 7.14 we include the performance of TCP-Westwood as it aims to improve TCP performance for transmission errors. We can see TCP-Westwood starts with lower performance than TCP-NewReno and TCP-RTTM. This is because of the way Westwood computes the TCP sending window, as we explained in chapter 3. TCP-Westwood uses the Bandwidth-Delay product to compute the sender window size, and because in our experiments we simulate congestion as well as transmission errors, TCP-Westwood will estimate a low available bandwidth (because of the congestion) and hence smaller window size. On the other hand TCP-RTTM and TCP-NewReno uses the AIMD (additive increase/multiplicative decrease) mechanism which increases the window size until error occurs. This results in higher window sizes, especially with low burst sizes (note: when we run the same experiment with higher delay values this big difference between Westwood and the other protocols became very small. This indicates that delays have a big impact on the AIMD mechanism when errors occur because with higher delays AIMD has difficulty recovering from errors). However, Westwood seems to respond in a better way, compared to NewReno, to the increase of burst size since the increase is only in the transmission errors (which Westwood can handle better) and not in the congestion errors that are fixed.

The conclusion from figure 7.14 is that RTTM can handle higher transmission error bursts sizes when compared to TCP-NewReno and TCP-Westwood.

Another comparison we conducted here is between TCP-RTTM and another protocol called TCP-Westwood-nr [104] which is actually the result of combining NewReno and Westwood into one protocol in order to gain the benefits of both multiple drops retransmission and the ability to deal with transmission errors.

Figure 7.15 shows the comparison between TCP-RTTM and TCP-Westwood-

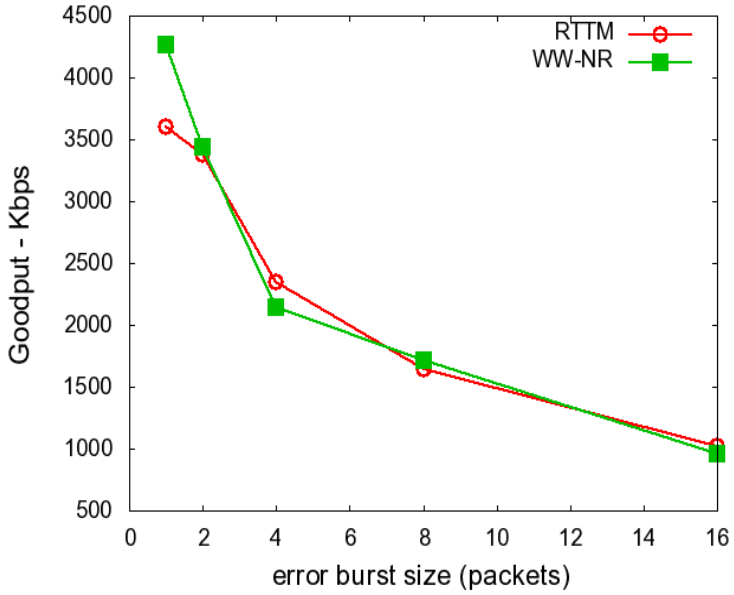

Figure 7.15: Performance with different transmission error burst size 2

nr (WW-NR). The results shows a high degree of similarity between RTTM and Westwood-nr performance (except under low burst sizes where NewReno performs better as we mentioned before). However, RTTM has an advantage in that it responds to transmission errors by cutting the congestion window using the CWA which can prevent further congestions in the network, in the case of congestion errors being mismatched as transmission errors. This shows that RTTM is able to achieve a performance close to Westwood-nr even if it cuts the congestion window in case of transmission errors.

Finally, we want to note that in these experiments we used packets as error

units instead of time (i.e. instead of using time to measure good and bad states in the error model, we used packets). This was done because we found it easier to apply specific burst sizes this way. However a side effect is that the reduction in performance was bigger than in the case when the error burst is measured in time where average drop rate was slightly lower. This is because when the bad state is measured in time, sometimes the system can be in the bad state but there are no packets to drop. However, when the bad state is measured in packets then it always drops the required number of packets (i.e. the system stays in the bad state until the required number of packets are dropped).

Also, we want to note that since TCP-Westwood and TCP-Westwood-nr are not included in the ns2 version that we used in our work (ns2.29), we added them to ns2 implementation and the source code was obtained from the TCP-Westwood official homepage [104].

7.6 System Limitations

The performance of TCP-RTT and TCP-RTTM will depend heavily on the computation of Cedge as presented in section 7.3.4. Cedge depends on the computed RTT and a fixed value for *midalpha*. From extensive simulations we found that $midalpha = 0.15$ gives the best results in terms of performance and in diagnosing of congestion errors, and hence in the reduction of the congestion loss rate.

However, even when using the same network conditions, TCP-RTT and TCP-RTTM congestion loss rates were slightly higher than standard TCP. This is due to the fact that the retransmission rates of TCP-RTT and TCP-RTTM were higher than TCP. Moreover, due to the increase in the performance gained by using TCP-RTT, the bottleneck utilization increases, and hence the congestion drops may in-

creases. Authors such as [72] have noticed the same phenomena when using the TCP-Casablanca [72] error discriminator and TCP-NewReno [37] under similar network conditions.

However, TCP-RTT and TCP-RTTM produced less congestion loss rates when compared to similar error discriminators (SpikeR and SpikeNR).

7.7 Summary

In this chapter we proposed an extension to TCP-RTT (proposed in chapter 6) where we use multiple drops action (MDA) not only for transmission errors but also for congestion errors. We call it TCP-RTTM.

We tested the improvement gained by TCP-RTTM and the effect of the error discriminator on the network. TCP-RTTM was found to produce less congestion loss rates when compared to a similar error discriminator with no transmission drop actions (TWA). Also TCP-RTTM gave high fairness index values no matter how many flows were competing in the bottleneck.

The potential effect of different *midalpha* values on the congestion loss rate was discussed. Then we applied different bandwidth and round trip values to see how they affected the performance. The round trip time values were chosen based on real Internet traces. In all cases TCP-RTTM was able to outperform standard TCP in terms of goodput.

Also we compared the effect of the error burst size on the performance of TCP-RTTM and we compared it with the same effect on the performance of two other protocols that are designed to tolerate error bursts. The results showed that TCP-RTTM was able to match the performance of these protocols.

Finally, we discussed some limitations and constraints on the design and perfor-

mance of TCP-RTT and TCP-RTTM.

Chapter 8

Conclusion

8.1 Conclusions

Many solutions have been proposed to overcome the problem of TCP performance degradation in heterogeneous networks where congestion and transmission errors may coexist. One class of solutions are the end-to-end error discriminators which can be added to TCP to improve its congestion control mechanism and so improve the performance. The benefit of using such a technique is that it requires changes to the end point only so no changes are required in the network. Also if it is applied to the sender side then the changes may be minimal, as the number of servers is usually much lower than the number of clients.

However, current error discriminators simply suppress TCP and prevent it from cutting the congestion window in the case of transmission errors. This has resulted in increasing the network congestion loss rate when using such mechanisms. In the following we summarize the main findings and contributions of this thesis:

- Through presenting different error discriminators it has been identified that current transmission actions used in error discriminators are not sufficient,

and might lead to increases in the network congestion loss rate as presented in many studies.

- Three areas where TCP interacts with drops have been identified : congestion window mechanism, retransmission mechanism and timeout mechanism.

- A new congestion window action (CWA) to deal with transmission errors has been proposed. This mechanism will be used by the error discriminator in the case of transmission errors to cut the congestion window according to number of dropped packets. This technique reduces the variability in the sending rate found in current error discriminators, which is caused by moving the congestion window, and hence the sending rate, between two extremes: one is to cut the congestion window in the case of congestion drops, and the other is to avoid cutting at all in the case of transmission errors.

- A new multiple drops action (MDA) has been proposed to help TCP to recover from multiple drops from the same window of data, and hence to reduce the number of timeout events.

- A proposal to add bandwidth estimation to the error discriminator in order to calculate the retransmission timeout back-off. This technique helps the error discriminator to reduce the idle time by reducing the back-off time if the bandwidth is available and the drops are actually transmission drops and not a result of congestion.

- The three algorithms, CWA, MDA and RTA are then added to a new end-to-end error discriminator, called TCP-RTT. Simulation results show that the proposed error discriminator with the new transmission window actions has achieved higher goodput than TCP. Moreover, TCP-RTT has much lower

congestion loss rates, less end-to-end delay and created smaller queue size than Spike error discriminator [1]. Also the fairness of TCP-RTT is compared with the fairness of Spike and the results show that TCP-RTT has a better sharing of bottleneck bandwidth than Spike. The proposed error discriminator, TCP-RTT, is a sender side solution (server) so there is no need to change the network or the clients' side.

- An analytical model to approximate TCP performance has been extended to approximate TCP-RTT performance. The new model considers the case where the congestion window cut factor is variable and is based on the error rate. The analytical model has been tested and the results show that it can approximate TCP-RTT performance.

- In the last chapter we proposed an extension to TCP-RTT to allow it to use the MDA action in the case of congestion drops as well as transmission errors. The new technique (TCP-RTTM) has been found to improve TCP-RTT performance in our experiments. Also the impact on the network congestion loss rate was compared with similar error discriminator and was found to be better in many cases.

The addition of transmission window actions (TWA) to the error discriminator plays an important role in creating traffic with less variability compared to current error discriminators which cause TCP sending rate to jump between cutting the congestion window size in case of congestion errors and avoiding cutting it at all in the case of transmission errors. TWA makes TCP-RTT to cut the congestion window even for transmission errors at a rate related to the number of dropped packets. This technique has produced a more stable congestion window responses which improves TCP performance and at the same time has a positive effect on the

network compared with existing error discriminators like Spike and SpikeNR.

Moreover, TCP-RTT has reduced the network congestion loss rate and the end-to-end delay; it is anticipated that this will improve the service provided by the network to other flows especially flows that require low delay or low drop rates.

8.2 Proposals for Extensions and Future Work

- Although the new transmission drops actions have improved TCP performance, they come with a number of limitations. For example, in CWA since *tthresh* records the congestion window size when the first drop occurs, and since CWA does not know if the drop is congestion drop or transmission drop, the performance of the algorithm will depend on when transmission errors occur after a timeout. If transmission errors keep occurring early enough, *tthresh* will take a small value and hence many errors will be considered congestion drops so the performance will be like normal TCP. One solution to this problem is that if the network suffers from constantly high transmission errors then we delay assigning *tthresh* to congestion window, so instead of assigning *tthresh* to congestion window after first drop we wait for the second or third drop. This can be implemented as a parameter in CWA which can be configured by the network operator who will evaluate the state of the connection (i.e. noisy or not). The implementation and the test of this solution are left for future work.

- We assumed the forward and backward link have the same speed when computing the bandwidth estimation in RTA which can give a false indication of high bandwidth if the backward path is faster than the forward path and vice versa. Although this is an inherent problem with TCP and not specific to

RTA since TCP increases its sending rate based on acknowledgments arrival rate, future work should study the potential effects of path speed asymmetry on the bandwidth estimation used in RTA.

- So far we added the TWA to end-to-end error discriminators which uses implicit feedback from the network like round trip time. We also want to test the effect of adding TWA to error discriminators that uses explicit feedback from the network like the ECN based error discriminator [105] and test it with our transmission window actions.

- In our topology we used drop tail queue in the bottleneck. However, since our error discriminator uses delay to infer congestion we want to test the effect of AQM (like RED [57]) on the proposed solutions.

- Improve the proposed analytical model to include the effects of MDA and RTA. The current model includes only the effects of CWA, congestion drops and timeout event.

- Congestion and packet loss in the backward path can increase the RTT significantly and hence gives a false indication of congestion in the forward path. Some solutions can be applied like using time stamps to measure the RTT. We want in the future work to apply time stamps to measure the RTT and conduct an analysis of TCP-RTT performance when applying this solutions.

- An improvement to the TCP-RTT error discrimination mechanism can be done by making it monitor whether the congestion increases or decreases. In the future work we plan to use the slope of the round trip time (RTT) readings and see if the slope increase or decrease ratios can be used as an indication of the direction of the congestion.

- The performance of TCP-RTT and TCP-RTTM will depend heavily on the computation of *Cedge* as presented in chapter 6. *Cedge* depends on the computed RTT and a fixed value for *midalpha*. We are interested in the future work to see if it is possible to automatically choose *midalpha* based on correlations found in the round trip time readings.

- In future work we want to investigate more complex scenarios like: different topology settings with multiple bottlenecks, other traffic models and cross traffic with flows that has specific quality of service requirements like video and audio.

- In future work we want to expand our comparisons to cover more error discriminators than the ones used in chapters 6 and 7.

References

[1] S. Cen, P. Cosman, and G. Voelker, "End-to-end differentiation of congestion and wireless losses," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 703–717, 2003.

[2] A. S. Tanenbaum, *Computer Networks*. Prentice Hall, 4th ed., 2005.

[3] M. Hassan and R. Jain, *High Performance TCP/IP Networking Concepts,Issues and Solutions*. Prentice Hall, 2005.

[4] J. Postel, "Transmission control protocol," *RFC 793*, 1981.

[5] H. Elaarag, "Improving TCP performance over mobile networks," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 357–374, 2002.

[6] A. Bhushan, "File transfer protocol," *RFC 114*, 1971.

[7] J. Poste and J. Reynolds, "Telnet protocol specification," *RFC 854*, 1983.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol-HTTP/1.1," *RFC 2616*, 1999.

[9] H. Zimmermann, "OSI reference model– The ISO model of architecture for open systems interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, 1980.

[10] V. Jacobson, "Congestion avoidance and control," in *Symposium proceedings on Communications architectures and protocols*, (Stanford, California, United States), pp. 314–329, ACM Press, 1988.

[11] V. Jacobson, "Modified TCP congestion avoidance algorithm," *email sent to end2end-interest mailing list*, 1990.

[12] G. Xylomenos, G. C. Polyzos, P. Mahonen, and M. Saaranen, "TCP performance issues over wireless links," *IEEE Communications Magazine*, vol. 39, no. 4, pp. 52–58, 2001.

[13] R. Caceres and L. Iftode, "The effects of mobility on reliable transport protocols," *In Proceedings of the 14th International Conference on Distributed Computing Systems*, pp. 12–20, 1994.

[14] H. Balakrishnan, S. Seshan, A. E., and R. H. Katz, "Improving TCP/IP performance over wireless networks," *In Proceedings 1st ACM international conference on Mobile Computing and Networking (Mobicom)*, 1995.

[15] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving tcp performance over wireless links," in *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, (Palo Alto, California, United States), ACM Press, 1996. pages 256-269.

[16] B. S. Bakshi, P. Krishna, N. H. Vaidya, and D. K. Pradhan, "Improving performance of TCP over wireless networks," in *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, p. 365, IEEE Computer Society, 1997.

[17] A. V. Bakre and B. R. Badrinath, "Implementation and performance evaluation of indirect TCP," *IEEE Transactions on Computers*, vol. 46, no. 3, pp. 260–278, 1997.

[18] A. Bakre and B. R. Badrinath, "I-TCP: indirect tcp for mobile hosts," in *15th IEEE International Conference on Distributed Computing Systems (ICDCS'95)*, pp. 136–143, 1995.

[19] R. Caceres and L. Iftode, "Improving the performance of reliable transport protocols in mobile computing environments," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 5, pp. 850–857, 1995.

[20] W. Stevens, "TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," *RFC 2001*, 1997.

[21] F. Cheng Peng and S. C. Liew, "TCP Veno: TCP enhancement for transmission over wireless access networks," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 2, pp. 216–228, 2003.

[22] T. Ye, X. Kai, and N. Ansari, "Tcp in wireless environments: problems and solutions," *IEEE Communications Magazine*, vol. 43, no. 3, pp. S27–S32, 2005.

[23] R. Braden, "Requirements for internet hosts - communication layers," *RFC 1122*, 1989.

[24] M. Allman, V. Paxson, and W. Stevens, "Tcp congestion control," *RFC 2581*, 1999.

[25] B. Sardar and D. Saha, "A survey of tcp enhancements for last-hop wireless networks," *IEEE Communications Surveys*, vol. 8, no. 3, pp. 20–34, 2006.

[26] G. T. Nguyen, B. Noble, R. H. Katz, and M. Satyanarayanan, "A trace-based approach for modeling wireless channel behavior," *Simulation Conference Proceedings*, pp. 597–604, 1996.

[27] V. Cerf and R. Kahn, "A protocol for packet network intercommunication," *IEEE Transactions on Communications*, vol. 22, no. 5, pp. 637–648, 1974.

[28] J. Postel, "User datagram protocol," *RFC 768*, 1980.

[29] L. L. Peterson and B. S. Davie, *Computer Networks a Systems Approach*. Morgan Kaufmann Publishing, Elsevier Science, San Francisco., 3rd ed., 2003.

[30] V. Paxson and M. Allman, "Computing tcp's retransmission timer," *RFC 2988*, 2000.

[31] J. C. Hoe, "Improving the start-up behavior of a congestion control scheme for tcp," in *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, (Palo Alto, California, United States), pp. 270–280, ACM Press, 1996.

[32] R. Jain and K. Ramakrishnan, "Congestion avoidance in computer networks with a connectionless network layer: concepts, goals and methodology," *Proceedings of the Computer Networking Symposium*, pp. 134–143, 1988.

[33] D. E. Comer, *Internetworking with TCP/IP, Principles, protocols and architecture*, vol. 1. Prentice Hall, 1995.

[34] D. M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems*, vol. 17, no. 1, pp. 1–14, 1989.

[35] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated*, vol. 2. Addison Wesely, 1995.

[36] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," *ACM Transactions in Computer Systems*, vol. 9, no. 4, pp. 364–373, 1991.

[37] S. Floyd and T. Henderson, "The NewReno modification to TCP's fast recovery algorithm," *RFC 2582*, 1999.

[38] A. Myles and D. Skellern, "Comparison of mobile host protocols for IP," *Journal of Internetworking Research and Ezperience*, vol. 4, no. 4, 1993.

[39] J. Ioannidis and G. Q. M. Jr, "The design and implementation of a mobile internetworking architecture," *In Proceedings of the USENIX Conference*, 1993.

[40] B. R. Badrinath, A. Bakre, T. Imielinski, and R. Marantz, "Handling mobile clients: A case for indirect interaction," *In Proceedings of the 4th IEEE Workshop on Workstation Operating Systems*, 1993.

[41] C. Parsa and J. J. Garcia-Luna-Aceves, "Improving tcp performance over wireless networks at the link layer," *Mobile Networks and Applications*, vol. 5, no. 1, pp. 57–71, 2000.

[42] E. Ayanoglu, S. Paul, T. F. LaPorta, K. K. Sabnani, and R. D. Gitlin, "Airmail: a link-layer protocol for wireless networks," *Wireless Networks, Kluwer Academic Publishers*, vol. 1, no. 1, pp. 47–60, 1995.

[43] K. Brown and S. Singh, "M-TCP: TCP for mobile cellular networks," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 5, pp. 19–43, 1997.

[44] V. Jacobson and R. T. Braden, "TCP extensions for long-delay paths," *RFC 1072*, 1988.

[45] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "Tcp selective acknowledgement options," *RFC 2018*, 1996.

[46] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno and SACK TCP," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 3, pp. 5–21, 1996.

[47] R. Jain, "A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 5, pp. 56–71, 1989.

[48] Z. Wang and J. Crowcroft, "A new congestion control scheme: slow start and search (tri-s)," *ACM SIGCOMM Computer Communication Review*, vol. 21, no. 1, pp. 32–43, 1991.

[49] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: new techniques for congestion detection and avoidance," in *Proceedings of the conference on communications architectures, protocols and applications*, (London, United Kingdom), pp. 24–35, ACM Press, 1994.

[50] L. S. Brakmo and L. L. Peterson, "Tcp vegas: end to end congestion avoidance on a global internet," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1465–1480, 1995.

[51] U. Hengartner, J. Bolliger, and T. Gross, "Tcp vegas revisited," in *Proceedings of Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2000*, vol. 3, pp. 1546–1555, 2000.

[52] S. Biaz and N. Vaidya, "Distinguishing congestion losses from wireless transmission losses: a negative result," in *Proceedings of the Seventh International Conference on Computer Communications and Networks*, pp. 722–731, 1998.

[53] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proceedings of the 7th annual international conference on Mobile computing and networking*, (Rome, Italy), pp. 287–297, ACM Press, 2001.

[54] M. Gerla, G. Pau, M. Y. Sanadidi, R. Wang, S. Mascolo, C. Casetti, and S. Lee, "TCP westwood: Enhanced congestion control for large leaky pipes," in *NASA Workshop*, 25 June 2001.

[55] L. Grieco and S. Mascolo, "Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 25–38, 2004.

[56] S. Biaz and N. Vaidya, "De-randomizing congestion losses to improve TCP performance over wired-wireless networks," *IEEE/ACM Transaction in Networking*, vol. 13, no. 3, pp. 596–608, 2005.

[57] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transaction on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[58] S. Floyd, "Tcp and explicit congestion notification," *ACM Computer Communication Review*, vol. 24, no. 5, pp. 10–23, 1994.

[59] S. Dawkins, D. Glover, J. Griner, D. Tran, T. Henderson, J. Heidemann, J. Touch, H. Kruse, S. Ostermann, K. Scott, and J. Semke, "Ongoing TCP research related to satellites," *RFC 2760*, 2000.

[60] S. Biaz and X. Wang, "Can ECN be used to differentiate congestion losses from wireless losses?," *Technical Report CSSE04-04, Auburn University*, 2004.

[61] K. Xu, Y. Tian, and N. Ansari, "Improving TCP performance in integrated wireless communications networks," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 2005.

[62] V. Jacobson, R. Braden, and D. Borman, "Tcp extensions for high performance," *RFC 1323*, 1992.

[63] S. Biaz and N. Vaidya, "Is the round-trip time correlated with the number of packets in flight ?," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement, IMC03*, 2003.

[64] S. Biaz and N. Vaidya, "Discriminating congestion losses from wireless losses using inter-arrival times at the receiver," in *Proceedings of Application-Specific Systems and Software Engineering and Technology Conference*, pp. 10–17, 1999.

[65] P. Sinha, T. Nandagopal, N. Venkitaraman, R. Sivakumar, and V. Bharghavan, "WTCP: a reliable transport protocol for wireless wide-area networks," *Wireless Networks*, vol. 8, no. 2/3, pp. 301–316, 2002.

[66] C. Zhang and V. Tsaoussidis, "TCP-real: improving real-time capabilities of tcp over heterogeneous networks," in *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, (Port Jefferson, New York, United States), pp. 189–198, ACM Press, 2001.

[67] X. Li, J. Wu, S. Cheng, and J. Ma, "Performance enhancement of transmission control protocol (TCP) for wireless network applications," *United States Patent 6757248*, 2004.

[68] Y. Tobe, Y. Tamura, A. Molano, S. Ghosh, and H. Tokuda, "Achieving moderate fairness for udp flows by path-status classification," in *Proceedings of the 25th Annual IEEE Conference on Local Computer Networks*, pp. 252–261, 2000.

[69] S. Biaz, M. Mehta, S. West, and N. Vaidya, "TCP over wireless networks using multiple acknowledgments," tech. rep., Texas A & M University, 1997. Report No. : 97-001.

[70] K. Tae-eun, L. Songwu, and V. Bharghavan, "Improving congestion control performance through loss differentiation," in *Proceedings of the Eight International Conference on computer Communications and Networks*, pp. 412–418, 1999.

[71] N. K. G. Samaraweera, "Non-congestion packet loss detection for TCP error recovery using wireless links," *IEE Proceedings in Communications*, vol. 146, no. 4, pp. 222–230, 1999.

[72] S. Biaz and N. Vaidya, "Differentiated services: A new direction for distinguishing congestion losses from wireless losses," tech. rep., University of Auburn, 2003. Report No. : CSSE03-02.

[73] S. Wang and H. Kung, "Use of TCP decoupling in improving tcp performance over wireless networks," *Wireless Networks*, vol. 7, no. 3, pp. 221–236, 2001.

[74] C. Lim, "An adaptive End-to-End loss differentiation scheme for TCP over wired/wireless networks," *IJCSNS,International Journal of Computer Science and Network Security*, vol. 7, no. 3, p. 72, 2007.

[75] S. Biaz and N. Vaidya, *Heterogeneous data networks: congestion or corruption?* PhD thesis, Taxes A&M University, 1999.

[76] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose, "Modeling tcp reno performance: a simple model and its empirical validation," *IEEE/ACM Trans. Netw.*, vol. 8, no. 2, pp. 133–145, 2000.

[77] G. Yang, R. Wang, M. Gerla, and M. Sanadidi, "TCP bulk repeat," *Computer Communications*, vol. 28, no. 5, pp. 507–518, 2005.

[78] J. C.-S. Wu, "A real time transport scheme for wireless multimedia communications," *Mobile Networks and Applications*, vol. 6, no. 6, pp. 535–546, 2001.

[79] G. Yang, R. Wang, M. Y. Sanadidi, and M. Gerla, "Performance of TCPW BR in next generation wireless and satellite networks," in *Proceedings of IEEE International Conference on Communications, ICC 2003*, vol. 1, pp. 674–678, 2003.

[80] L. Rizzo, "TCP retransmissions on very lossy networks," 1996, `http://info.iet.unipi.it/~luigi/tcp.ps` (accessed October 2007).

[81] ns2, "The network simulator," 2006. `http://www.isi.edu/nsnam/ns.` (accessed September 2008).

[82] G. Holland and N. Vaidya, "Analysis of TCP performance over mobile ad hoc networks," *Wireless Networks, Kluwer Academic Publishers*, vol. 8, no. 2/3, pp. 275–288, 2002.

[83] H. Hung-Yun and R. Sivakumar, "Performance comparison of cellular and multi-hop wireless networks: a quantitative study," in *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Cambridge, Massachusetts, United States, ACM Press, 2001.

[84] Wikipedia, "Berkeley software distribution," `http://en.wikipedia.org/wiki/Berkeley_Software_Distribution` (accessed Feb 2008).

[85] G. Wang and Y. Xia, "An ns2 tcp evaluation tool," *Internet Engineering Task Force*, 2007.

[86] W. R. Stevens, *TCP/IP illustrated the protocols*. Addison-Wesley professional computing series, Addison-Wesley Pub. Co., 1994.

[87] V. Paxson and S. Floyd, "Wide area traffic: the failure of poisson modeling," *IEEE/ACM Transactions on Networking*, vol. 3, no. 3, pp. 226–244, 1995.

[88] L. Muscariello, M. Meillia, M. Meo, M. A. Marsan, R. L. Cigno, and D. di Elettronica, "An MMPP-based hierarchical model of internet traffic," in *Proceedings of IEEE International Conference on Communications*, vol. 4, 2004.

[89] M. Efnuseva, "mmpp traffic generator for ns," *`http://www.tlc-networks.polito.it/muscariello/mmpp_tg/` (accessed 2007).*

[90] B. Paolo, *Packet Loss in Terrestrial Wireless and Hybrid Networks*. PhD thesis, University of Pisa, 2007.

[91] W. M. Eddy, S. Ostermann, and M. Allman, "New techniques for making transport protocols robust to corruption-based loss," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 5, pp. 75–88, 2004.

[92] C. D. Vleeschouwer and P. Frossard, "Explicit window-based control in lossy packet networks," tech. rep., Swiss Federal Institute of Technology, Signal Processing Institute, 2004.

[93] W. Matthews and L. Cottrell, "The pinger project: active internet performance monitoring for the HENP community," *IEEE Communications Magazine*, vol. 38, no. 5, pp. 130–136, 2000.

[94] T. P. p. IEPM, "Internet end-to-end performance monitoring," *http://www-iepm.slac.stanford.edu/pinger/, Accessed (March 2007).*

[95] S. Kurkowski, T. Camp, and M. Colagrosso, "MANET simulation studies: the incredibles," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 4, pp. 50–61, 2005.

[96] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: a simple model and its empirical validation," in *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*, vol. 28, pp. 303–314, 1998.

[97] "Matlab, the mathwork inc.," 1994-2006.

[98] "Office excel, microsoft corporation.," 1985-2003.

[99] H. D. Jeong, *Modelling of self-similar teletraffic for simulation.* PhD thesis, University of Canterbury, New Zealand, 2002.

[100] H. D. J. Jeong, J. S. R. Lee, D. McNickle, and K. Pawlikowski, "Distributed steady-state simulation of telecommunication networks with self-similar teletraffic," *Simulation Modelling Practice and Theory*, vol. 13, no. 3, pp. 233–256, 2005.

[101] M. Chen and A. Zakhor, "Rate control for streaming video over wireless," *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM04*, vol. 2, 2004.

[102] M. Matthew, S. Jeffrey, and M. Jamshid, "The macroscopic behavior of the TCP congestion avoidance algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997.

[103] M. Sooriyabandara and G. Fairhurst, "Performance limitations due to TCP burstiness in GEO satellite networks with limited buffering," in *London Communications Symposium*, pp. 127–130, 2007.

[104] U. C. S. Department, "TCP westwood home page." `http://www.cs.ucla.edu/NRL/hpi/tcpw/` (accessed September ,2008).

[105] R. Ramani and A. Karandikar, "Explicit congestion notification (ecn) in tcp over wireless network," 2000.

[106] M. S. Taqqu, V. Teverovsky, and W. Willinger, "Estimators for long-range dependence: an empirical study," *Fractals*, vol. 3, no. 4, pp. 785–798, 1995.

[107] T. Karagiannis, M. Faloutsos, and M. Molle, "A user-friendly self-similarity analysis tool," *Special Section on Tools and Technologies for Networking Research and Education, ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 81–93, 2003. 957004.

Appendix A

Standard TCP Reaction to Drops

Following algorithm shows the standard TCP reaction to drops as described in [20]:

```
1: if (AckSeqNo == last_ack) then reported = reported+1
2:     if reported == 3 then                                   ▷ Packet drop
3:         resend packet with seqNo=AckSeqNo+1
4:         cwnd = cwnd / 2
5:         ssthresh = cwnd
6:     end if
7: end if
8: if timeout==true then
9:     ssthresh = cwnd/2                    ▷ Actually ssthresh = min(2,cwnd/2)
10:    cwnd = 1
11: end if
12: recalculate RTO
```

Variables:
AckSeqNo: Sequence number of the received acknowledgment.
last_ack : Sequence number of the last acknowledged packet (last new acknowledgment).
reported : Variable that keeps track of how many duplicate acknowledgment TCP received so far. This variable is set to 0 whenever a new acknowledgment is received.
cwnd: Congestion window size.
ssthresh: Slow Start threshold
RTO: Retransmission timeout timer. Calculated based in the average RTT.

Figure A.1: Standard TCP reaction to drops

Appendix B

MDA - The Case of Non Sequence Errors

MDA is designed to recovers multiple drops per window when they happen in sequence. However, we present here an idea for the case when error are not in sequence. In this case we have two options, either to follow TCP-Newreno [37] and to resend the dropped packets one per round trip time which will take long time if we have too many dropped packets but it will prevent multiple retransmission of already received packets. Other method is to resend all the window as Bulk Repeat [77] which is good if we have too many drops per window but if the error rate is low it will cause a lot of unnecessary retransmission of packets.

Our proposal is to combine the two options. We will use number of dropped packets to estimate dropping rate per window. Then based on that we choose which action to choose (i.e. one packet per RTT or resend all the window).

We will define α which represent the error rate, so that if drops rate per window is lower than α then we use slower but more conservative method of resending packet per RTT since the number of errors is low. However, if the number of errors per

window exceeded α then TCP can resends all window at once. The algorithm is the same as MDA with following additions:

- We count number of partial acknowledgments and after receiving the second partial acknowledgment which indicates that the first retransmission did not cover all lost packets because they were not consecutive we estimate the error rate roughly based on number of dropped packets per window as following:

$$error\_rate = \frac{num\_drops}{window\_size}$$

- If error_rate $\geq \alpha$ then there are many errors and it is OK to resend the whole window starting from the `prev_ack+1` until `last_sent_3Dack`.

- However if the error_rate $< \alpha$ then we resend one packet per round trip time since we have only small drops.

The flowchart in figure B.1 Shows the updated MDA algorithm.

Choosing α depends on the network operator needs and the state of the lossy link. If the connection has high rate of errors then it is better to choose lower values for alpha to allow TCP to do more retransmission. However, if the error rate is low then a higher value for α is preferable to prevent TCP from retransmitting unnecessary packets.

As we said the computation of error_rate is a rough computation and we do not claim it represent the actual drop rate in the network. However, the use of this method will be as a final resort and after MDA fail to recover all drops.

Also an improvement to the way to compute error_rate is to use a weighted average of error_rate measurement from different windows instead of a single measurement as we did above which will prevent affected by sudden changes in the

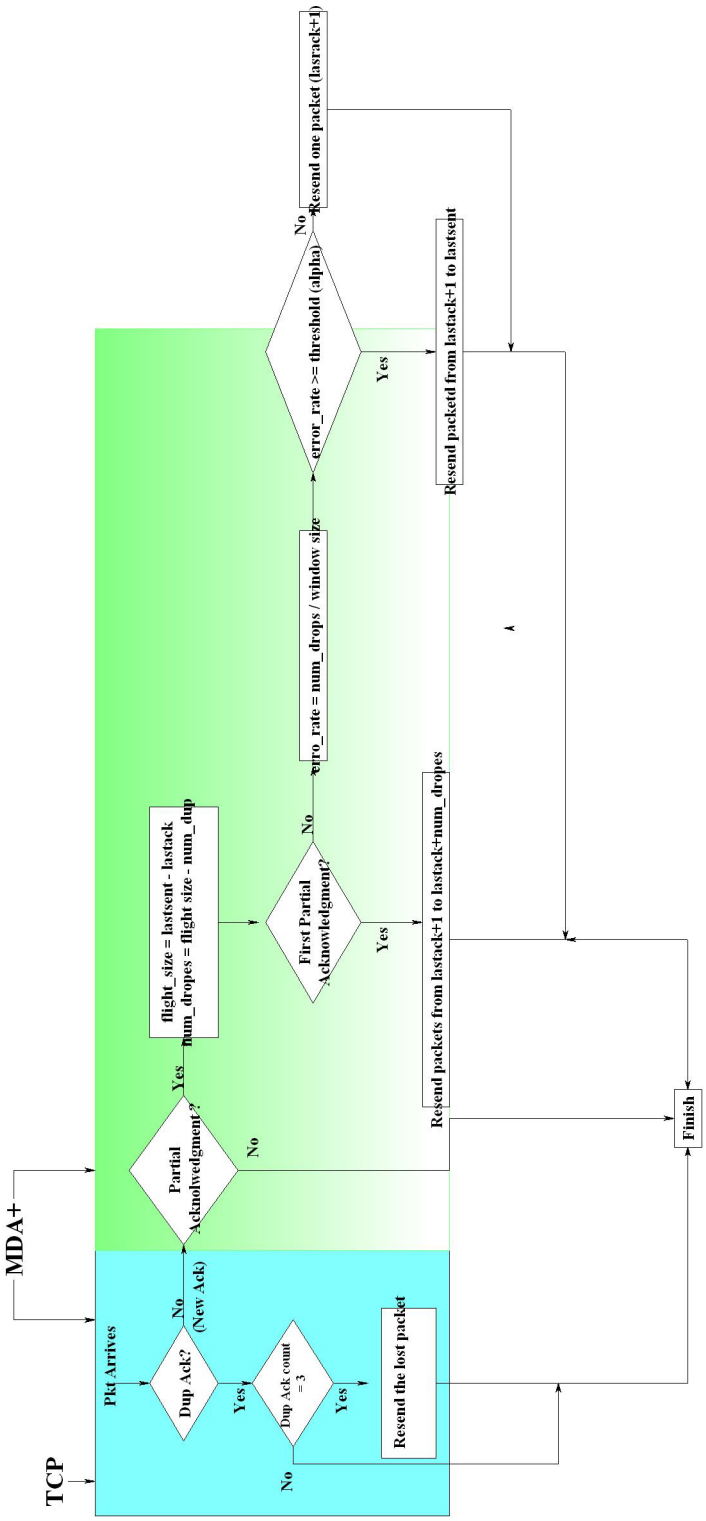

Figure B.1: Multiple drops action + error rate estimation flow chart

error_rate .

Finally, here we presented the idea and we plan to add it to MDA and test it in the future work.

Appendix C

Traffic Generation

Here we will show that MMPP traffic generator [88] which we used in our simulator has the ability to approximate burstiness and correlation over large time scales found in Internet traffic. The correlation in Internet traffic is widely known as long range dependence (LRD).

We use a simple topology presented in figure C.1 with one sender that uses an MMPP traffic generator. Then we count number of arrival packets to the first router R1 during time T where T takes different values namely 1, 0.1 and 0.01 seconds.

The resulted traffic is then fed to an external tool we developed using MATLAB and uses the Aggregated Variance Method as described in [106] to calculate the Hurts parameter which is used to measure the level of LRD in the traffic [106] . As a reference we use also a ready made tool called SELFIS [107] (SELF similarity analysIS) which uses several methods to to calculate the Hurts parameter (If all methods indicates that the traffic is LRD then we report results using Aggregated Variance method only).

Table C.1 shows the Hurst parameter for different intervals using our tool (called AV) and SELFIS. A Hurst parameter > 0.5 indicates presence of LRD in the traf-

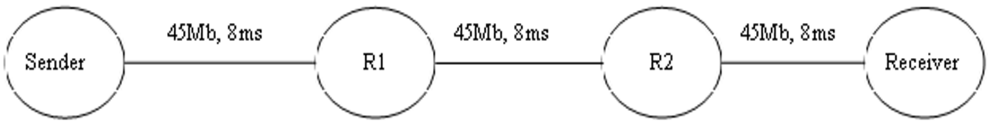

Figure C.1: Simple Topology

Time Interval (seconds)	Hurst Parameter	
	AV estimator	SELFIS
1	0.68	0.69
0.1	0.76	0.78
0.01	0.77	0.79

Table C.1: MMPP traffic generator Hurst parameter for different intervals

fic [106]. As we can see all results indicates that MMPP traffic generator can generate LRD traffic. The small difference between the results obtained using our Av estimator and SELFIS is probably due to different methods of implementing mathematical and statistical functions and the differences in programming language (MATLAB uses C/C++ and SELFIS uses Java).

Following we include the MATLAB code for the AV tool, the ideas in this MatLab code is based on the documentation of Aggregated variance method in [106]:

AV estimator for Long Range Dependence

```
% M Alnuem 2007
  echo off;
  clear all;
  clc;
  [traffic] = textread('traffic0.01s-100sRun.tr','%f'); %Input file
  [i,j] = size(traffic);
  n=i;
  mlimit=n/2 %maximum size m can take
  count1 = 0;
  i=2;
%C is the constant that determains next m sizee
%so m(i+1)=mi*C (see Taqqu et al.,1995)
  C=2;
%Here we caculate different sizes of m
  while (i <= mlimit)
      count1 = count1+1;
      mvec(count1)=i;
      i=i*C;
 end
 %mvec
 [t,mvecsize] = size(mvec);
 for bk=1:mvecsize % for bigk=1 to number of m sizes
     m=mvec(bk); % current m
      nm = n/m;
      Xm=1:nm;
%for k=1,2....N/m Calculate Xm(k).
%see (Taqqu et al.,1995) equation 3.1
      for k=1:nm
          count2 =0;
          for i=(k-1)*m+1:k*m
              count2=count2+1;
              X(count2)=traffic(i);
          end
          Xm(k)=mean(X);
      end
%Calculate sample variance for block m.
```

```
%see ( Taqqu et al.,1995) equation 3.2
      variance(bk)= var(Xm);

 end

% Following we calculate the log-log scale for he variance and m
% Then we calclate the slop of the best fit
%line and use this slop to calculate H
          logm = log10(mvec);
          logvar = log10(variance);
          sumX=sum(logm);
          sumY=sum(logvar);
          squareX=power(logm,2);
     ssquareX=sum(squareX);
     XY=logm.*logvar;
          sumXY=sum(XY);
     [m,n] = size(mvec);
     slop=(sumXY - (sumX*sumY/n))/(ssquareX-(power(sumX,2)/n))% Calculating the Slop
     H =(slop/2)+1; % Calculating H according to taqqu95 slop = 2H -2
     H
     %Followin we draw the best fit line in a log-log chart for the variance and m
     %We have the slop so nest we calculate the intercept point b
     b=(sumY-slop*sumX)/n;
     Xaccess = logm;
     for i=1:n
          Yaccess(i)=slop*logm(i)+b;
     end

     plot(logm,logvar,'*',Xaccess,Yaccess,'r.-');
     title('Aggregated Variance Method');
     xlabel('log10(m)');
     ylabel('log10(variance)');
```

Appendix D

TCP-RTT Source Code

Following a subset of TCP-RTT source code.

```
1  /*
2  TCP–RTT. M alnuem 2007
3
4  Here I will implement the actions toward transmission errors + the RTT error discriminator.
5  The actions implements here are
6  –CWA : congestion window action
7  –MDA: Multiple drops action
8  –RTA: Retransmission timeout action
9
10 Note that some functions used here are inherited from the class TcpAgent in TCP.cc and
11 from Tcp–Reno.cc in ns2
12
13 */
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <sys/types.h>
18 #include "ip.h"
19 #include "tcp.h"
20 #include "flags.h"
21 #include "random.h"
22 #include "basetrace.h"
23 #include "hdr_qs.h"
```

```
24
25 static class edtaTcpClass : public TclClass {
26 public:
27         edtaTcpClass() : TclClass("Agent/TCP/Reno/edta") {}
28         TclObject* create(int, const char*const*) {
29                 return (new edtaTcpAgent());
30         }
31 } class_TA;
32
33 edtaTcpAgent::edtaTcpAgent() : RenoTcpAgent(),
34 window_edge(0),
35 window_edge_time(0.0),
36 num_dupack(0),
37 dup_flag(0),
38 dup_flight(0),
39 first_dup(1),
40 lastsent(-1),lastack(-1),tmpls(-1),
41 num_backoffs(0),
42 idel_time(0.0),
43 prev_packet_time(0.0),
44 current_pacekt_time(0.0),
45 avg_idel_time(0.0),
46 pkt_count(0.0),
47 tthresh_(0.0),
48 first_drop(1),
49 ack_time(0),
50 avg_rate(0),
51 maxavg_rate(0),
52 nt_backoff_(0),
53 builk_repeat(0),
54 single_cut(0),
55 congdp_(0),
56 wiredp_(0),
57 AvgRTT_(0),
58 alpha_(0.9),
59 maxp_(0.9),
60 ncongd_(0),
61 nwired_(0),
62 vRTT_(0),
```

```
63 method_(0),
64 midalpha_(0.15),
65 last_dupack_(0),
66 last_recoreded_(-1),
67 p_avg_delay(0),
68 p_delay(0),
69 MDA_congestion(0),
70 tthresh_enabled(1)
71 {
72                     bind("idel_time", &idel_time);
73                     //bind("avg_idel_time", &avg_idel_time);
74                     bind("num_backoffs", &num_backoffs);
75                     bind("CWA_enabled", &CWA_enabled);
76                     bind("MDA_enabled", &MDA_enabled);
77                     bind("RTA_enabled", &RTA_enabled);
78                     bind("single_cut", &single_cut);
79                     bind("tthresh_", &tthresh_);
80                     bind("nt_backoff_",&nt_backoff_);
81                     bind("t_backoff_",&t_backoff_);
82                     bind("avg_rate",&avg_rate);
83                     bind("maxavg_rate",&maxavg_rate);
84                     bind("AvgRTT_", &AvgRTT_);
85                     bind("congdp_", &congdp_);
86                     bind("wiredp_", &wiredp_);
87                     bind("ncongd_", &ncongd_);
88                     bind("nwired_", &nwired_);
89                     bind("alpha_", &alpha_);
90                     bind("maxp_", &maxp_);
91                     bind("CeilRTT_", &CeilRTT_);
92                     bind("FloorRTT_", &FloorRTT_);
93                     bind("method_", &method_);
94                     bind("midalpha_", &midalpha_);
95                     bind ("sentpackets",&sentpackets);
96                     bind("p_avg_delay",&p_avg_delay);
97                     bind("MDA_congestion",&MDA_congestion);
98                     bind("tthresh_enabled",&tthresh_enabled);
99                     outf = fopen("out.tr","w");
100                    cdropf = fopen("congout.tr","w");
101                    wdropsf = fopen("wireout.tr","w");
```

```
102                     cdropf2 = fopen("congout2.tr","w");
103                     wdropsf2 = fopen("wireout2.tr","w");
104                     alldrops = fopen("alldrops.tr","w");
105                     CeilRTT_ = 0.0;
106                     FloorRTT_ =10000;
107                     Waction_=0;
108
109 }
110
111  void edtaTcpAgent::tcpprint ( char* toprint)
112 {
113
114           fprintf(outf,toprint);
115 }
116
117
118 void edtaTcpAgent::MDA(Packet *pkt)
119 {
120         hdr_tcp *tcphdr = hdr_tcp::access(pkt);
121          char s[50];
122     sprintf(s,"Partial Ack :,%d,%d\n",tcphdr->seqno(),lastsent);
123         tcpprint(s);
124         int inc = 0;
125         if (CWA_enabled) // last ack+1 has been sent
126         inc=1;
127         else
128         inc=0;
129                 if (first_partial)
130 // This will run only when the first
131 //partial acknowledgment arrives
132         {
133         double flight_size =  lastsent-lastack;
134    if (flight_size > 0)
135    {
136    double percent_droped =double( Tphase_window/flight_size);
137         char d[50];
138    sprintf(d,"in 1, percent_droped:%f\n",percent_droped);
139    tcpprint(d);
140
```

```
141         if (percent_droped >= 0.05)
142         {
143         sprintf(d,"in 2\n");
144         tcpprint(d);
145         builk_repeat=1;}
146         else {builk_repeat=0;
147         sprintf(d,"in 3\n");
148         tcpprint(d);}
149         }
150         builk_repeat=0;
151         if (!builk_repeat)
152         for (int i=1+inc;i<=(Tphase_window+1);i++)
153         resned(lastack+i);
154         char f[350];
155         sprintf(f,"flight1:%f,Tphase_window:%d,dupAck:%d,builkrepeat:%d,T/F:%f\n",
156 flight_size,Tphase_window,dup_count,builk_repeat,
157 Tphase_window/flight_size);
158         tcpprint(f);
159         output (t_seqno_++,0);
160         }
161         else// resend at least one packet
162         { builk_repeat=0;
163                 if (!builk_repeat)
164                 {
165         resned(lastack+1+inc);
166         output (t_seqno_++,0);
167         }
168 }
169
170         builk_repeat=0;
171         reset_rtx_timer(1,0);
172         if (builk_repeat)
173         {
174         t_seqno_ = lastack+1+inc;
175         }
176
177 }
178
179 void edtaTcpAgent::initial_checks(Packet *pkt)
```

```
180 {
181         hdr_tcp *tcphdr = hdr_tcp::access(pkt);
182         ++nackpack_;
183         ts_peer_ = tcphdr->ts();
184         recv_helper(pkt);
185         recv_frto_helper(pkt);
186         if (tcphdr->seqno() == lastack) // Duplicate Acknowledgment
187         {
188         ++num_dupack;
189     ++dup_count;
190         if (dup_count==3)
191         {
192         dup_flight = maxseq_ -  tcphdr->seqno() ;
193         dup_flag = 1;
194         Tphase = 1;//Entering trnsmission phase
195         first_partial = 1;
196         if (first_dup)
197         {
198                 if (CWA_enabled)
199                 fast_retrans(0);
200 //CWA - resend and delay the cut
201 //untill we recieve first partial
202 //acknowledgment
203                 else
204                 fast_retrans(1);
205 // Reno case- cut after first 3 duplicate acknowledgment
206                 lastsent = maxseq_;
207         }
208         if (CWA_enabled && first_drop)
209 // first drop in this
210 //in this connection or
211 //after a timeout
212         {
213                 tthresh_ = cwnd_;
214                 first_drop = 0;
215         }
216                 }
217         if (dup_flag)
218         {
```

```
219         --dup_flight;
220     if (dup_flight == 0)
221     {
222         char s[50];
223      sprintf(s,"end_of_Dups\n");
224         //tcpprint(s);
225         dup_flag= 0;
226            }
227         }
228     }
229 }
230
231 int edtaTcpAgent::Twindow()
232 {
233         if (first_partial)
234 // This will run only when
235 //the first partial acknowledgment arrives
236         {
237         int flight_size =  lastsent-lastack;
238         int window = flight_size-dup_count; // Number of dropped packets
239         if (window>0)
240         return(flight_size-dup_count);
241         else
242         return (0);
243 }
244 else
245 return(Tphase_window);
246 }
247
248 void edtaTcpAgent::bw_est()
249   { //Computing the bandwidth for RTO back-off
250         double oack_time =ack_time ;
251         ack_time = Scheduler::instance().clock();
252         double rate =((size_*8)/(ack_time-oack_time))/1024;
253         double alpha = 0.9;
254         avg_rate = (avg_rate*alpha) + (rate*(1-alpha));
255         if ( maxavg_rate < avg_rate)
256         maxavg_rate = avg_rate ;
257 }
```

```
258
259 void edtaTcpAgent::packet_delay(Packet *pkt)
260 {
261   hdr_tcp *tcpheader = hdr_tcp::access(pkt);
262   if (!(tcpheader->seqno() == lastack))
263   {
264   double now = Scheduler::instance().clock();
265   double Ackts = tcpheader->ts();
266   double Go = Ackts - tcpheader->ts_echo(); // forward path Delay
267   p_delay+=Go;
268   p_avg_delay= (p_delay/++pkt_count)*1000;
269 // multiply *1000  to Convert from seconds to ms
270   }
271 }
272
273 // This function should be called from rtt_timeout() in TCP.cc in ns2
274 void edtaTcpAgent::RTA(){
275         if (maxavg_rate>0)
276         {
277         nback_offs = log2(t_backoff_);
278         double pavg_rate = 1-(avg_rate / maxavg_rate);
279         nback_offs = nback_offs * pavg_rate;
280         nt_backoff_ = pow(2,nback_offs);
281 }
282 else {nt_backoff_ = t_backoff_;}
283 }
284
285 //RTT-Error Discrimination function
286 void  edtaTcpAgent::ED(Packet *pkt)
287 {
288         double now = Scheduler::instance().clock();
289         hdr_tcp *tcpheader = hdr_tcp::access(pkt);
290   // First compute the AvgRTT
291   oldRTT = newRTT;
292   newRTT= now - tcpheader->ts_echo() ;
293   double Ackts = tcpheader->ts();
294   //newRTT = newRTT/tcp_tick_;
295   newRTT = newRTT*1000;//Convert  from secondes to ms
296   double newGo = Ackts - tcpheader->ts_echo(); // forward path Delay
```

```
297    //newGo = newGo/tcp_tick_;
298    newGo = newGo*1000;
299    oldAvg_=AvgRTT_;
300    if (tcpheader->seqno() != last_ack_)
301    AvgRTT_ = alpha_*AvgRTT_+(1-alpha_)*newRTT;
302    vRTT_ = (vRTT_*(alpha_)) + (fabs(newRTT-AvgRTT_)*((1-alpha_)));
303    double vari = newRTT-AvgRTT_;
304    if ((FloorRTT_ > newRTT) && (newRTT>0) )
305    FloorRTT_ = newRTT;
306    if (CeilRTT_< newRTT)
307    if (tcpheader->seqno() != last_ack_)
308    CeilRTT_ = newRTT;
309    double mid = FloorRTT_ + midalpha_*(CeilRTT_-FloorRTT_); // Cedge
310    if (AvgRTT>=mid)
311    congdp_ = 1;
312    else
313    congdp_=0;
314    wiredp_ = 1-congdp_;
315   if (tcpheader->seqno() == last_ack_)
316   if (dupacks_ == 0)
317   {
318   //fprintf(outf,", First DupAck");
319   dupRTT1_ = oldRTT;
320   dupRTT2_ = newRTT;
321 }
322
323   if (congdp_)
324   {
325   CWA_enabled=0;
326   MDA_enabled=MDA_congestion;
327 // enable/disable MDA for congestion losses as well
328   RTA_enabled=0;
329 }
330 else
331 {
332   CWA_enabled=1;
333   MDA_enabled=1;
334   RTA_enabled=1;
335 }
```

```
336 }
337
338 void edtaTcpAgent::recv(Packet *pkt, Handler*)
339 {
340         hdr_tcp *tcphdr = hdr_tcp::access(pkt);
341         double now = Scheduler::instance().clock();
342         current_pacekt_time = now;
343         packet_delay(pkt);// compute the forward path Delay  for trancing.
344         ED(pkt);//Call the error discriminator
345     initial_checks(pkt);
346     if (RTA_enabled)
347     bw_est();// bandwidth estimation
348     else //Reno case
349     avg_rate = 0;
350 // This way TCP will use exponential back off as in Reno
351         //////Following New Acknowledgment Block////////
352         if (tcphdr->seqno() > lastack) {
353 // New Acknowldgment - or Partial acknowledgment
354     prev_packet_time = current_pacekt_time ;
355     lastack = tcphdr->seqno();
356     recv_newack_helper(pkt);
357 // This function is inherited from TCP.cc in ns2
358     dupwnd_ = 0;// Exit fast recovery
359         if (lastsent > tcphdr->seqno())
360 // partial acknowledgment - Entering transmission errors phase
361         {
362         Tphase_window=Twindow();
363 // compute Tphase window which
364 //depends on number of droppes packets
365 //Tphase_window will be used by MDA to resend
366 //number of packets equla to Tphase_window
367 // and will be used by CWA to cut cwnd acoording to Tphase_window
368         if (CWA_enabled)
369 // trnamsission errors congestion window action
370         fast_retrans(2);
371         if (MDA_enabled) // Multiple drop action
372         MDA(pkt);
373         first_dup = 0;
374 } else {// A new Acknowledgment that acknowledge all outstanding data
```

```
375 first_dup=1;
376 Tphase = 0;
377 dup_count=0;
378 dup_flag =0;
379 dup_flight=0;
380 }
381 //       dup_flag =0;
382 //dup_flight=0;
383 //       dup_count=0;
384   first_partial = 0;
385   Packet::free(pkt);
386   send_much(0, 0, maxburst_);
387 // send as much as min(cwwnd, reciever window) allows.
388 }
389
390 void
391 edtaTcpAgent::win_cut(int method,double amount)
392 {
393         ++ncwndcuts_;
394     if (method == 1 )
395     {
396     cwnd_ =  amount;
397     // In case of transmission error we keep the ssthresh as it is because
398     //Equilibrium point probably has not changed
399     //ssthresh_ = (int) amount;
400 }
401         else if (method ==2)
402         ssthresh_ = (int) amount;
403         else if (method ==3)
404         {
405         cwnd_ =  amount;
406         ssthresh_ = (int) amount;
407         }
408         if (cwnd_ < 1)
409                 cwnd_ = 1;
410         if (ssthresh_ < 2)
411                 ssthresh_ = 2;
412    if ((method ==1) ||( method ==3))
413                 cong_action_ = TRUE;
```

```
414 }
415
416 void edtaTcpAgent::reduce_cwnd(int reduction_type)
417 {
418         int same_window = (recover_ < lastack);
419 // are we in the same window
420         if (!single_cut)
421         same_window=1;
422         //if (recover_ > lastack)
423         //{
424         if ((reduction_type == 1) && (same_window)) //congestion drop
425         {
426         recover_ = maxseq_;
427 // recover is lastseen but
428 //we keep it for compatibility with other tcp ns2 codes.
429         tmpls = maxseq_;
430         last_cwnd_action_ = CWND_ACTION_DUPACK;
431 // this variable is
432 //required for compatability withe file tcp.cc in ns2
433         int old_win = windowd();
434         double new_win = windowd()/2;
435 //        char s[50];
436 //      sprintf(s,"Wincutbefore1:,%d,%d\n",int(cwnd_),int(ssthresh_));
437 //       tcpprint(s);
438         win_cut(3,new_win);
439 //        sprintf(s,"Wincutafter1:,%d,%d\n",int(cwnd_),int(ssthresh_));
440 //                tcpprint(s);
441         }
442
443         if ((reduction_type == 2) && (same_window)) // Transmission drop
444         {
445         recover_ = maxseq_;
446 // recover is lastseen but
447 //we keep it for compatability with other tcp ns2 codes.
448         last_cwnd_action_ = CWND_ACTION_DUPACK;
449 // this variable is
450 //required for compatability withe file tcp.cc in ns2
451 //                char s[50];
452 //      sprintf(s,"Wincutbefore2:,%d,%d\n",int(cwnd_),int(ssthresh_));
```

```
453 //        tcpprint(s);
454           if ((cwnd_ < tthresh_) || !(tthresh_enabled) )
455 //and the error is transmission error according to the ED
456           {
457           int new_win = cwnd_-Tphase_window;
458 // other option is to  do:  new_win = windowd()-Tphase_window
459           if (new_win<1)
460           new_win=1;
461           double now = Scheduler::instance().clock();
462           win_cut(1,new_win);
463       }
464           else
465           //reduce_cwnd(1);
466           {
467           //win_cut(3,cwnd_/2);
468           int old_win = windowd();
469           double new_win = windowd()/2;
470           win_cut(1,new_win);
471 // We cut the congestion window  (and not not the ssthres)
472 //only since there is a chance the error is transmission error.
473 }
474           }
475 //}
476 }
477
478 void edtaTcpAgent::resned(int seqno)
479 {
480           char s[50];
481       double now = Scheduler::instance().clock();
482           sprintf(s,"Resending at:,%f,%d\n",now,seqno);
483           tcpprint(s);
484           output(seqno, TCP_REASON_DUPACK);
485 }
486
487 void  edtaTcpAgent::fast_recovery()
488 {
489           dupwnd_ = numdupacks_;
490 }
491
```

```
492 void
493 edtaTcpAgent::fast_retrans(int reduction_type )
494 //Transmission action Fast Retransmission
495 {
496
497                 //int reduction_type= 1;
498                 // 1: reduce cwnd to half, ssthresh to half
499                 reduce_cwnd(reduction_type);
500                 resned(lastack+1);
501                 fast_recovery();
502                 reset_rtx_timer(1,0);
503                 first_dup=0;
504                 if (lastack+1> last_recoreded_)
505                 {
506                 if (congdp_ || ((cwnd_ >= tthresh_) && (tthresh_enabled)) )
507             fprintf(cdropf,"%d\n",last_ack_ + 1);
508             else
509             fprintf(wdropsf,"%d\n",last_ack_ + 1);
510             fprintf(alldrops,"%d\n",last_ack_ + 1);
511             // Following to record all drops considered
512         //congestion by the error discriminator
513         //(i.e. without using tthreash)
514             if (congdp_ )
515                fprintf(cdropf2,"%d\n",last_ack_ + 1);
516             else
517               fprintf(wdropsf2,"%d\n",last_ack_ + 1);
518     }
519    last_recoreded_ =lastack+1 ;
520         return;
521         }
```