University of Southern Queensland

Faculty of Engineering & Surveying

Improving Wireless TCP/IP Performance Using the Median Filter Algorithm

A thesis submitted by

Auc Fai Chan

in fulfilment of the requirements of

Master of Engineering Research

Submitted: March, 2010

Abstract

The estimation of Retransmission Timeout (RTO) in Transmission Control Protocol (TCP) affects the throughput of the transmission link. If RTO is just a little larger than Round Trip Time (RTT), retransmissions will occur too often, and this increases congestion in the transmission link. If RTO is much larger than RTT, the response to retransmit when a packet is lost will be too slow, and this will decrease the throughput in the transmission link (Comer 2006*a*).

Currently, the Jacobson/Karels Algorithm is widely used for the estimation of RTO in TCP implementations. The algorithm uses an Exponential Weighted Moving Average (EWMA) filter to estimate RTT and then determines the RTO from this. The EWMA filter is good if the RTT follows a Gaussian distribution. In reality, traffic in the Internet is bursty and tends to follow a heavy-tailed distribution. Using an EWMA approach to estimate heavy-tailed distribution is inadequate.

The median filter has been recognized as a useful non-linear filter due to its edge preserving and impulse suppressing characteristics, so it is effective in removing impulsive noise (Nodes & Gallagher Jr. 1982). The median filter has been applied to many areas of signal processing, particularly in image processing to remove positive and negative impulsive noise. Thus, it can perform well for heavy-tailed distributions.

In this project, the median filter is applied to estimate the RTT over TCP links, under bursty traffic conditions. Experiments and simulations are conducted to determine if the median filter performs better than the EWMA filter. In the experiments, consistent RTT and a small RTO are obtained, which are desirable factors for high connection throughput. In the simulations, the median filter not only delivers higher throughput, but also drops fewer packets during transmission than EMWA filter does. Both the experiments and simulations show that the median filter outperforms the EWMA filter under bursty traffic conditions.

Associated Publications

The following publications were produced during the period of candidature:

A. F. Chan and J. Leis, "Comparison of Weighted-Average and Median Filters for Wireless Retransmission Timeout Estimation", *Signal Processing and Communication Systems, 2008. ICSPCS 2008. 2nd International Conference on*, 2008, pp 1-6.

A. F. Chan and J. Leis, "Median Filtering Simulation of Bursty Traffic", (in draft) for *Signal Processing and Communication Systems, 2010. ICSPCS 2010. 4th International Conference on*, 2010, pp 1-5.

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

AUC FAI CHAN

0031138976

____

Signature

____

Date

Acknowledgments

First of all, I would like to thank my Principal Supervisor Dr. John Leis for providing me with useful suggestions about the overall structure and contents of my thesis and for helping me with several simulation experiments.

Next, I would like to thank my co-supervisor, Dr. Alexander Kist, whose ideas and suggestions always motivated me.

I would also take this opportunity to thank the staff at the University of Southern Queensland for their encouragement and assistance.

Auc Fai Chan

University of Southern Queensland
March 2010

Contents

Abstract i

Associated Publications ii

Acknowledgments iv

List of Figures xiii

List of Tables xviii

Acronyms & Abbreviations xx

Chapter 1 Introduction 1

1.1 Motivation and Objectives 1

1.2 Organization of this dissertation 3

1.3 Contributions of this Research Project 5

1.4 Further Works of this Research Project 6

Chapter 2 Transmission Control Protocol (TCP) 7

2.1 Introduction 7

2.2 Slow Start 8

2.3 Congestion Avoidance 9

2.4 Fast Retransmit 10

2.5 Fast Recovery 12

2.6 Illustration of Tahoe and Reno 12

2.7 New Reno TCP 13

2.8 Sack TCP 13

2.9 Vegas TCP 14

2.10 BIC and CUBIC TCP 14

2.11 Westwood TCP 15

2.12 TCP Evolution 16

2.13 Summary 16

Chapter 3 Round Trip Time (RTT) and Retransmission Timeout (RTO) 18

3.1 Introduction 18

3.2 Historical Development of RTO 19

3.3 Difficulties in Choosing RTO 20

3.4 Original Algorithm 21

3.5 Karn/Partridge Algorithm 22

3.6 Jacobson/Karels Algorithm 24

3.7 More about Jacobson/Karels Algorithm 25

3.8 Problems with RTT Estimation 27

3.9 Exponential Distribution and Pareto Distribution 28

3.10 Summary 31

Chapter 4 The Median Filter and its Modifications 32

4.1 Introduction 32

4.2 Median Filter 33

4.3 Ranked-order Filters 35

4.4 Recursive Median Filters 36

4.4.1 Properties of Recursive Median Filter 37

4.5 Weighted Median Filters 37

4.6 Recursive Weighted Median Filters (RWMF) 39

4.7 Symmetric Weighted Median Filters 40

4.8 Centre Weighted Median Filters (CWMF) 41

4.9 Adaptive Weighted Median Filters 41

4.10 Summary 43

Chapter 5 Experimental Evaluation of Retransmission Characteristics 44

5.1 Introduction 44

5.2 Experimental Environment 44

5.3 Analysis of Experimental Results 45

5.4 Comparison of Different Round Trip Times 49

5.5 Comparison of Different Retransmission Timeouts 50

5.6 Histograms of Round Trip Times 55

5.7 Mode and Outliers 60

5.8 Computation and Implementation Consideration 61

5.8.1 Example on Memory Access 61

5.9 Summary 63

Chapter 6 A Fast Sorting Algorithm for Median Filtering 64

6.1 Introduction 64

6.2 Comparison of three sorting algorithms 65

6.2.1 The Big-O Notation 65

6.2.2 Application of Big-O Notation 65

6.3 Shellsort 66

6.4 Quicksort 68

6.4.1 Quicksort Algorithm 69

6.4.2 Partition Function 70

6.4.3 More about Quicksort 71

6.5 Fast 2D Median Filtering Algorithm 71

6.5.1 Fast 1D Median Filtering Algorithm 72

6.5.2 Huang's Algorithm in 2D Median Filtering 74

6.6 Sorting Algorithm Performance 76

6.6.1 Shellsort Performance 76

6.6.2 Quicksort Performance 77

6.6.3 Fast 1D Median Filtering Algorithm Performance 78

6.6.4 Comparison of Experimental Results 80

6.7 Analyses of Memory Access in Algorithms 80

6.7.1 Analysis of Memory Access in Shellsort 81

6.7.2 Analysis of Memory Access in Quicksort 82

6.7.3 Analysis of Memory Access in Fast 1D Median Filtering 83

6.8 Summary 86

Chapter 7 Simulations and Analysis of Results 87

7.1 Introduction 87

7.2 The Network Simulator NS2 87

7.3 Median Filter Algorithm in TCP Tahoe 89

7.3.1 Median Filters of sizes 7 and 9 89

7.4 Computation of Retransmission Timeout (RTO) 90

7.5 The Simulation Script 90

7.5.1 Throughput and Goodput 91

7.6 Experiment 1: Exponential Traffic Simulation 92

7.6.1 Settings and Experimental Analysis 93

7.7 Experiment 2: Pareto Traffic Simulation 99

7.7.1 Settings and Experimental Analysis 101

7.8 Error Model for Pareto Traffic 106

7.8.1 Overview of Error Model 106

7.8.2 Error Model Applied to TCP Tahoe 106

7.8.3 Error Model Applied to TCP Median 107

7.8.4 Comparison of TCP Tahoe and TCP Median 108

7.9 Simulation of Two Wireless Nodes 109

7.9.1 TCP Connection over a Wireless Link 110

7.9.2 Error Model Imposed on the Link 110

7.10 Mixed Wireless/Wired Simulation 111

7.11 Performance of TCP Median in Wireless Scenarios 113

7.12 Summary 114

Chapter 8 Conclusions and Further Work 115

8.1 Introduction 115

8.2 Project Overview 115

8.2.1 Project Motivation 116

8.2.2 Project Objective 116

8.2.3 Summary of Literature Review 117

8.3 Project Achievements 119

8.3.1 Development of a New One-Dimensional Median Filtering 119

8.3.2 Practical Experiment to achieve better RTO 120

8.3.3 Median Filter Outperforms Weighted-Average in Simulations . . 121

8.4 Publications arising from this Research 122

8.5 Further Work 122

8.5.1 Median Filter in Reno TCP 122

8.5.2 Median Filter in Full TCP 123

8.5.3 The Fast One-Dimensional Median Filtering Algorithm 124

Bibliography **125**

Appendix A Papers Published in Connection with this Research **133**

Appendix B Partition Function Example **145**

Appendix C Quicksort Algorithm Example **149**

Appendix D Shellsort Source Code **152**

Appendix E Quicksort Source Code **156**

Appendix F Fast 1D Median Filtering Source Code **160**

Appendix G Bubble Sort Source Code **163**

Appendix H The Median Filter Algorithm in TCP Tahoe 165

Appendix I Simulation Script testing.tcl 168

Appendix J Simulation Script wirelessa.tcl 188

Appendix K Simulation Script wirelessb.tcl 196

List of Figures

2.1 Packets in transit during slow start 9

2.2 Fast Retransmit based on duplicate ACKs 11

2.3 TCP's congestion window (Tahoe and Reno) 12

3.1 ACK for retransmission 22

3.2 ACK for original transmission 23

3.3 RTO computed by RFC 793 rules (Jacobson 1988) 25

3.4 RTO computed by Jacobson/Karels Algorithm (Jacobson 1988) 26

3.5 Exponential and Pareto probability density functions (Hei 2001) 30

4.1 Median Filters suppress impulse and at the same time preserve edge . . 34

5.1 A point-to-point wireless connection 45

5.2 Unfiltered RTT from a wireless connection. Note that one of the RTT timing is as long as 2.4 seconds. 46

5.3 Median-filtered RTT with window size of three. The fluctuation in amplitude of RTT is reduced. 46

5.4 Median-filtered RTT with window size of five. Only two large RTT's are not filtered out. 47

5.5 Median-filtered RTT with window size of seven. There is no significant improvement when the window size is larger than five. 47

5.6 Median-filtered RTT with window size of nine. There is no significant improvement when the window size is larger than five. 48

5.7 Smoothed RTT by using equation (5.1). The pattern in Figure 5.7 is the same as that in Figure 5.2, except that the amplitude in Figure 5.7 is reduced. 50

5.8 Unfiltered RTO from a wireless connection. Figure 5.8 will be compared with other RTO's in Figure 5.11. 52

5.9 Smoothed RTO using (5.2) and (5.3). Figure 5.9 will be compared with other RTO's in Figure 5.11. 52

5.10 Median-filtered RTO with window size of three. Figure 5.10 will be compared with other RTO's in Figure 5.11. 53

5.11 Different RTOs. Median-filtered RTO has amplitude less than those of SRTO and unfiltered RTO. 53

5.12 Unfiltered RTO and median-filtered RTO. Median-filtered RTO has amplitude less than that of unfiltered RTO. 54

5.13 Unfiltered RTO and median-filtered RTO (Zoom-in view). Figure 5.13 is the enlargement of Figure 5.12, for the portion near time equal to zero. 54

5.14 Histogram of unfiltered RTT from a wireless connection. There are RTT's of long duration lying on the right of the mode. 55

5.15 Histogram of smoothed RTT by using equation (5.1). The RTT's are distributed in a similar pattern as that in Figure 5.14, except that the duration is shorter. 56

5.16 Histogram of median-filtered RTT with window size of three. For window size of 3, the RTT's are still distributed on the right side of the mode. It will be seen in later figures that the distribution pattern will change. 56

5.17 Histogram of median-filtered RTT with window size of five. Some RTT's begin to exist on the left side of the mode. 57

5.18 Histogram of median-filtered RTT with window size of seven. The distribution of RTT's approaches a normal distribution. 57

5.19 Histogram of median-filtered RTT with window size of nine. There are no significant changes after window size of seven. 58

5.20 Histogram of median-filtered RTT with window size of eleven. There are no significant changes after window size of seven. 58

5.21 Histogram of median-filtered RTT with window size of thirteen. There are no significant changes after window size of seven. 59

6.1 Comparison of growth n, $n\log n$ and n^2. It can be seen that the growth rate of n^2 is the largest. 66

6.2 Experimental results of Shellsort 77

6.3 Experimental results of Quicksort 78

6.4 Experimental results of Fast 1D Median Filtering Algorithm 79

6.5 Comparison of the three sorting algorithms 80

7.1 Simulation Network of Experiment 1, s1 sends packets to d1; s2 to d2 and so on. Links from sources to RO and from R1 to destinations are 10 Mb. ROR1 is a bottleneck of 1 Mb. 93

7.2 Comparison of accumulative goodput between median filter and weighted-average filter. This figure shows that a median filter size 5 can deliver higher goodput. 94

7.3 RTT and RTO obtained from weighted-average. The RTT curve and RTO curve are farther apart. The response to retransmit when a packet is lost will be too slow. 96

7.4 RTT and RTO obtained from median filter of size 5. The RTT curve and RTO curve are closer to each other. This will probably increase the throughput. 97

7.5 RTT and RTO obtained from median filter of size 7. Since median filter of size 7 delivers fewer goodput than size 5 does, size 7 is not the best choice. 98

7.6 RTT and RTO obtained from median filter of size 9. Since median filter of size 9 delivers fewer goodput than size 5 does, size 9 is not the best choice. 98

7.7 Simulation Network of Experiment 2, s1 sends packets to d1; s2 to d2 and so on. Links from sources to RO and from R1 to destinations are 10 Mb. R0R1 is a bottleneck of 1 Mb. 100

7.8 Comparison of accumulative goodput between median filter and weighted-average filter. This figure shows median filter size 5 can deliver more goodput. 101

7.9 RTT and RTO obtained from weighted-average. The RTT curve and RTO curve are farther apart. The response to retransmit when a packet is lost will be too slow. 103

7.10 RTT and RTO obtained from median filter of size 5. . The RTT curve and RTO curve are closer to each other. This will probably increase the throughput. 103

7.11 RTT and RTO obtained from median filter of size 7. Since median filter of size 7 delivers fewer goodput than size 5 does, size 7 is not the best choice. 104

7.12 RTT and RTO obtained from median filter of size 9. Since median filter of size 9 delivers fewer goodput than size 5 does, size 9 is not the best choice. 104

7.13 TCP Tahoe throughput versus lossrate. As the lossrate increases, the throughput decreases. 107

7.14 TCP Median throughput versus lossrate. As the lossrate increases, the throughput decreases. 108

7.15 Throughput of TCP Median decreases to a lesser extent. At high loss rates, there is little to differentiate the two. 109

7.16 Throughputs of TCP Tahoe and TCP Median versus packet loss rate. At packet loss rate of 1%, TCP Median outperforms TCP Tahoe by 10%. 111

7.17 Mixed Wireless/Wired Simulation. Packets are sent from S to D via BS. 112

7.18 Throughputs of TCP Tahoe and TCP Median versus packet loss rate. At packet loss rate of 1%, TCP Median outperforms TCP Tahoe by 12%. 112

List of Tables

5.1 Effect on different window sizes 60

6.1 Contributions of n, $n\log n$ and $2n^2$ to the function y. It can be seen that $2n^2$ contributes most to y. 65

6.2 Memory Access Analysis on Shellsort main loop 81

6.3 Memory Access Analysis on insertion function 82

6.4 Memory Access Analysis on Quicksort main loop 82

6.5 Memory Access Analysis on partition function 83

6.6 Memory Access Analysis on bubble sort function 85

6.7 Memory Access Analysis on fast median main loop 85

7.1 Exponential traffic and analysis for median filter size 5 and weighted-average filter 95

7.2 Comparison of packet drop percentages. It is found that all packet drop percentages of median filters are lower than that of the weighted-average filter which is 1.66%. 99

7.3 Pareto traffic and analysis for median filter size 5 and weighted-average filter 102

7.4 Packet drop percentages of median filters with size 5, 7 and 9. It is found that all packet drop percentages from median filters are lower than the packet drop percentage from the weighted-average, which is 1.93%. . . . 105

Acronyms & Abbreviations

1D	one dimensional
2D	two dimensional
ACK	Acknowledgement
BIC	Binary Increase Congestion Control
CBR	Constant Bit Rate
cwnd	Congestion window
EWMA	Exponential Weighted Moving Average, also called Weighted Average
FIR	Finite Impulse Response
FTP	File Transfer Protocol
IIR	Infinite Impulse Response
LAN	Local Area Network
Mbps	Megabits per second
MF	Median Filter
NS2	Network Simulator, version 2
pdf	probability density function
RF	Radio Frequency
RFC	Requests For Comments
RTO	Retransmission Timeout
RTT	Round Trip Time
ssthresh	slow start threshold
tcl	Tool Command Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network

Chapter 1

Introduction

1.1 Motivation and Objectives

In wireless networks, electromagnetic waves, instead of cables, are used to connect telecommunication devices such as cellular telephones, communication satellites and wireless local area networks (Goldsmith 2005). The implementation of connecting wireless networks, take place at the physical layers of the networks (Patil & et al. 2003).

Wireless Local Area Networks use radio frequency (RF) waves to transmit data among computers in a small area such as offices and educational institutions (Goldsmith 2005). Wireless networks are quick-fix methods to connect remote regions, where there is a lack of telecommunications infrastructure, to the Internet. On the other hand, if wired networks are used, long cables are needed to connect these remote regions to the service providers.

Wireless networks are becoming more and more popular. They allow users to access the Internet with their laptop computers at any locations inside their offices or homes, so wireless networks can provide convenience and mobility to their users (Goldsmith 2005).

In the Government Accountability Office (GAO) report, John Poirier stated that the number of wireless subscribers in the United States had increased from about 3.5 million in 1989, to about 270 million in 2008 (Poirier 2009). Wireless networks are also installed

in airports, hotels, cafes and shopping malls to provide wireless access to the Internet for their customers.

In the Internet, reliable data transfer service is provided by Transmission Control Protocol (TCP) (Kurose & Ross 2005). TCP was designed for used in wired network and it does not work so well in wireless networks (Balakrishnan & et al. 1997), because wireless networks have high bit-error rates due to channel fading, noise and interference.

TCP was first proposed for wired networks where packet losses were always due to congestion in the link. Packet losses caused by high bit-error rates in wireless networks, are wrongly interpreted by TCP congestion avoidance algorithm as congestion in the link (Balakrishnan & et al. 1997).

Since TCP was designed originally for wired networks, the motivation of this research project is to investigate the performance of TCP over wireless networks. The main objective of this research project is to enhance TCP performance in managing congestion over wireless networks, by using median filters. Median filters will be used to estimate the Round Trip Time (RTT). Then, the RTT is used to determine the Retransmission Timeout (RTO).

When a host computer sends out a packet, it starts a timer. If the host receives an acknowledgement (ACK) before the timer expires, it sends a new packet and restarts the timer. The time between the stating of the timer and the receiving of ACK is called the Round Trip Time (RTT) (Comer 2006*b*, Peterson & Davie 2000).

If the timer expires before the host receives an ACK, the host will retransmit the same packet. The time between the starting of the timer and the expiring of the timer is called the Retransmission Timeout (RTO) (Kurose & Ross 2005, Comer 2006*b*, Peterson & Davie 2000).

The median filter has been recognized as a useful non-linear filter due to its edge preserving and impulse suppressing characteristics, so it is efficient in removing impulsive noise (Nodes & Gallagher Jr. 1982). The median filter has been applied to many areas of signal processing, particularly in image processing to remove positive and negative impulsive noise which causes white dots and black dots on photographs. The median

filter is expected to have a better RTT estimation in such a way that the throughput under bursty traffic increases.

In this research, the median filter is applied to network simulations for retransmission timeout estimation, in the presence of bursty traffic flows. Our simulation results show that, the median filter can perform better than the EWMA filter, in terms of throughput and packet drop percentage, i.e. the median filter delivers more throughput and loses less packets.

The objectives of this research project are:

1. To review the literature to understand how TCP works.
2. To review the literature to understand median filters.
3. To develop a fast algorithm for median filtering and apply it to the problem of RTT estimation (and hence determination of RTO).
4. To conduct a practical experiment to show that median filter can produce better RTO.
5. Using network simulator version 2 (NS2), to investigate how median filters can improve the retransmission timeout (RTO) of TCP and to use Exponential and Pareto traffic sources in the investigation.
6. To analyse the data obtained in the simulations and to determine whether the median filter can deliver more goodput than weighted-average filter.
7. To present the literature review, practical experiments, NS2 simulations, and analysis of experimental results, in a dissertation.

1.2 Organization of this dissertation

This dissertation is organized as follows:

Chapter 1 introduces the motivation, objectives and organization of this dissertation. TCP was designed for wired networks, and it does not work so well in wireless networks. Wireless networks have high bit-error rates due to channel fading, noise and interference. Packet losses caused by high bit-error rates are wrongly interpreted by

TCP as congestion in the transmission link. How can we improve TCP performance in wireless networks? This question motivated this research project. The objectives of this research project, the dissertation organization, the basic contributions of this research project, and publications generated from this research project are introduced in Chapter 1.

Chapter 2 is a literature review on TCP. The major components of TCP, such as slow start, congestion avoidance, fast retransmit and fast recovery are presented here.

Chapter 3 is a literature review on Round Trip Time and Retransmission Timeout (RTO). The evolution of RTO from its original algorithm to Karn/Partridge algorithm, and then to Jacobson/Karels algorithm are described in this chapter.

Chapter 4 describes the class of median filter algorithms. This chapter includes median filters and some of their modifications. These modifications include ranked-order filters recursive median filters.

In Chapter 5, a practical experiment is presented. There are two approaches to the testing of our median filter algorithm. One approach is by means of practical experiment, which is presented in Chapter 5. Another approach is to use software such as Network Simulator, version 2 (NS2) (ISI 2008*a*) to conduct simulations, which is presented in Chapter 7. In this experiment, a simple point-to-point wireless connection is established between a laptop computer and a desktop computer. Results from the experimental analysis show that median filter algorithm performs better than the weighted-average.

In Chapter 6, the Big-O notation is applied to analyze the speed of the three sorting algorithms. Then, literature reviews on Shellsort and Quicksort are presented. Finally, a fast one-dimensional median filtering algorithm was formulated for TCP, based on the work of T.S. Huang *et al.*.

Chapter 7 is about the modification of tcp.cc program, simulations and analysis of results. tcp.cc is a module inside NS2 and it simulates Transmission Control Protocol. Our goal of conducting the simulations is to show that, under bursty traffic flows (approximated by Exponential and Pareto traffic sources), median filter can perform

better than weighted-average filter.

Analysis of simulation results show that median filter delivers more goodput than weighted-average filter does, where goodput is defined as throughput minus retransmission. In addition, the median filter has a lower packet drop percentage, meaning that (retransmission/throughput)×100% is lower.

Chapter 8 provides some conclusions on the overall project, and proposes further research based on the findings. The primary conclusion is that both practical experiment and simulation show that median filter performs better than EWMA.

1.3 Contributions of this Research Project

The contributions of this research project are briefly presented below:

- The current TCP uses a method called exponential weighted moving average (EWMA) to estimate the round trip time (RTT). The median filter can perform better than the EWMA. Consistent RTT and smaller retransmission timeout (RTO) are obtained by using median filter. The contribution of this project is that, the above results are desirable factors for higher link throughput.

- A 'fast' one-dimensional median filtering algorithm is developed in this project. Shellsort, Quicksort and the 'fast' one-dimensional median filtering algorithms are experimentally used to sort the same set of data, and then the timings, spent by the three methods to sort the same set of data, are compared. Experiments show that the fast one-dimensional median filtering algorithm is faster than Quicksort and much faster than Shellsort. The fast one-dimensional median filtering algorithm could satisfy our need of a fast sorting algorithm.

- Simulation results using NS2, with Exponential and Pareto traffics, show that a median filter of size 5 delivers more throughput than EWMA filter does. Moreover, median filter of size 5 has lower packet drop percentage than EWMA has. These results are consistent with the experimental results.

1.4 Further Works of this Research Project

The first further development is to improve the computational efficiency of median filtering, which involves sorting an array into ascending order and selecting the element at the middle. A decrease in the number of steps in sorting can improve the computational efficiency. For the practical experiment, it should be easy to implement the fast one-dimensional median filtering algorithm developed in this project. For simulation with NS2, it will be a complicated task to modify the program tcp.cc using the fast one-dimensional medium filtering algorithm.

The second further development is to apply median filter in Full TCP. The Tahoe TCP is a one-way agent, which only sends packets from sources to destinations. Full TCP is a two-way agent, which performs bidirectional data transfer. A test can be set up to investigate whether the Full TCP with median filter in it, can deliver more goodput.

The third further development is to apply median filter to Reno TCP. The modified TCP used in this project is Tahoe TCP, which contains slow start, congestion avoidance and fast retransmit, but does not contain fast recovery. Reno TCP contains fast recovery. A test can be set up to investigate whether the Reno TCP with median filter in it, can deliver more goodput.

Chapter 2

Transmission Control Protocol (TCP)

2.1 Introduction

Two researchers, Vinton G. Cerf and Robert E. Kahn, proposed an inter-network protocol called TCP/IP. They published their paper, "A Protocol for Packet Network Intercommunication", in May 1974 in IEEE Transactions on Communications (Cerf & Kahn 1974).

Current implementations of TCP include four interwoven algorithms as the Internet standards: slow start, congestion avoidance, fast retransmit, and fast recovery. Van Jacobson (Jacobson 1988, Jacobson 1990) provided details on these algorithms. W. Stevens (Stevens 1994) provided examples of the algorithms in action. G. Wright (Stevens & Wright 1995) provided the source codes for the implementation. RFC 1122 requires that a TCP had to implement slow start and congestion avoidance (Braden 1989*a*), but fast retransmit and fast recovery were implemented after RFC 1122.

This chapter describes the basic elements of TCP, namely, slow start, congestion avoidance, fast retransmit and fast recovery. Several variations of TCP, such as Vegas TCP, Westwood TCP are introduced. This chapter ends with a section on evolution of TCP.

2.2 Slow Start

Slow start is one of the algorithms that TCP uses to control congestion inside the network. It is also known as the exponential growth phase. TCP starts a connection with the sender injecting multiple packets into the network, up to the window size advertised by the receiver. There is no problem when the two hosts are on the same LAN, but if there are routers and slower links between the sender and the receiver, problems can arise. Some intermediate routers must put the packets in queue, and it is possible for these routers to run out of space. The algorithm to avoid this is called slow start. It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgments (ACKs) are returned by the other end (Jacobson 1988).

Slow start adds another window to the sender's TCP: the congestion window, abbreviated to "cwnd". When a new connection is established by a host on a network, the congestion window is initialized to one packet (i.e., the packet size announced by the other end). Each time an ACK is received, the congestion window is increased by one packet. The sender can transmit up to the minimum of the congestion window and the advertised window. That is, if cwnd is smaller, transmit up to cwnd. If advertised window is smaller, transmit up to advertised window. The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver.

The sender starts by transmitting one packet and waits for its ACK. When that ACK is received, the congestion window is increased from one to two, and two packets can be transmitted.

When each of these two packets is acknowledged, the congestion window is increased to four. This provides an exponential growth. At some point the capacity of the Internet will be reached, and an intermediate router will start dropping packets. This tells the sender that there is network congestion and it needs to take steps to reduce the load on the network (Jacobson 1988).

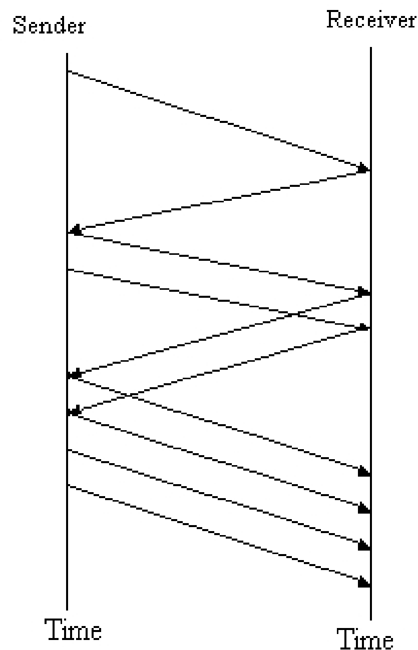

Figure 2.1: Packets in transit during slow start

Figure 2.1 shows the growth in the number of packets in transit during slow start, which increases the congestion window exponentially, rather than linearly (Peterson & Davie 2000).

2.3 Congestion Avoidance

Congestion can occur when data arrives on a big pipe (a fast LAN) and it is sent out to a smaller pipe (a slower WAN). Congestion can also occur when multiple input streams arrive at a router whose output capacity is less than the sum of the inputs. Congestion avoidance is a way to deal with lost packets (Braden 1989*a*).

There are two indications for packet loss: a timeout occurring and the receipt of duplicate ACKs. Congestion avoidance and slow start are independent algorithms with different objectives. In practice they are implemented together (Jacobson 1988).

Congestion avoidance and slow start require two variables to be maintained for each connection: a congestion window, cwnd, and a slow start threshold, ssthresh. The combined algorithm operates as follows:

1. Initialization for a given connection sets a slow start threshold, ssthresh, for example, to eight packets, and congestion window, cwnd to one packet.
2. The TCP output routine never sends more than the minimum of cwnd and the receiver's advertised window.
3. When congestion occurs (indicated by a timeout or the reception of duplicate ACKs), ssthresh is set to one-half of the current window size (the minimum of cwnd and the receiver's advertised window, but at least two packets). In addition, if the congestion is indicated by a timeout, cwnd is set to one packet (i.e., slow start).
4. When new data is acknowledged by the other end, cwnd is increased, but the way it is increased depends on whether TCP is performing slow start or congestion avoidance.

If cwnd is less than or equal to ssthresh, TCP is in slow start; otherwise TCP is performing congestion avoidance. Slow start continues until TCP reaches ssthresh, and then congestion avoidance takes over.

Slow start has cwnd beginning at one packet, and has cwnd increased by one packet every time an ACK is received. As mentioned earlier, this increases the window exponentially: TCP sends one packet, then two, then four, and so on.

Congestion avoidance is a linear growth of cwnd, compared to slow start's exponential growth. In section 2.6, Illustration of Tahoe and Reno, a graph will be shown to clarify the relationship between slow start and congestion avoidance.

2.4 Fast Retransmit

Modifications to the congestion avoidance algorithm were proposed in 1990 (Jacobson 1990). Researchers realized that TCP generates an immediate acknowledgment (a duplicate ACK) when an out-of-order packet is received. The purpose of this duplicate ACK is to let the other end know that a packet was received out-of-order, and to tell the other end what sequence number is expected (Braden 1989*a*).

Since TCP does not know whether a duplicate ACK is caused by a lost packet or just a reordering of packets, it waits for a small number of duplicate ACKs to be received.

It is assumed that if there is just a reordering of the packets, there will be only one or two duplicate ACKs before the reordered packet is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a packet has been lost. TCP then performs a retransmission of what appears to be the missing packet, without waiting for a retransmission timer to expire (Stevens 1997). This is called fast retransmit.

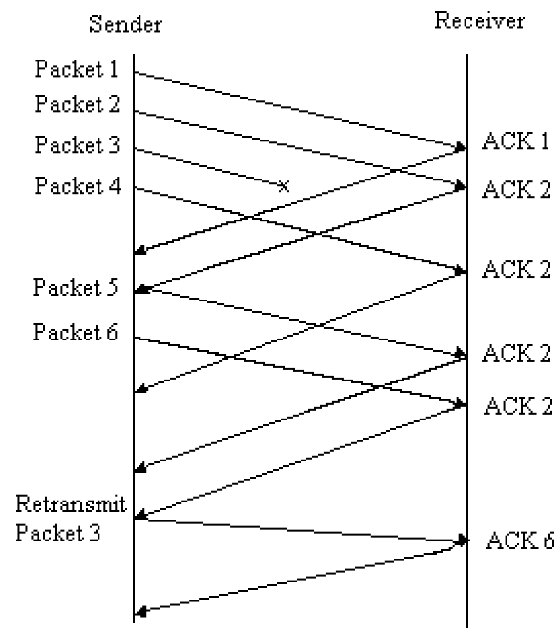

Figure 2.2: Fast Retransmit based on duplicate ACKs

Figure 2.2 illustrates how duplicate ACKs lead to a fast retransmit. In this example, the receiver receives packets 1 and 2, but packet 3 is lost in the network. Thus, the receiver will send a duplicate ACK for packet 2 when packet 4 arrives, again when packet 5 arrives, and so on. When the sender receives the third duplicate ACK for packet 2, the sender retransmits packet 3. When the retransmitted copy of packet 3 arrives at the destination, the receiver then sends a cumulative ACK for everything, up to and including packet 6, back to the sender (Peterson & Davie 2000).

2.5 Fast Recovery

When the third duplicate ACK in a row is received, ssthresh is set to one-half the congestion window, the congestion window at the time the third duplicate ACK is received. Then the missing packet is retransmitted (Jacobson 1988).

After fast retransmit sends what appears to be the missing packet, congestion avoidance, but not slow start is performed. This is called the fast recovery (Jacobson 1990, Stevens 1994).

In TCP Reno, the fast recovery is implemented but not in TCP Tahoe.

2.6 Illustration of Tahoe and Reno

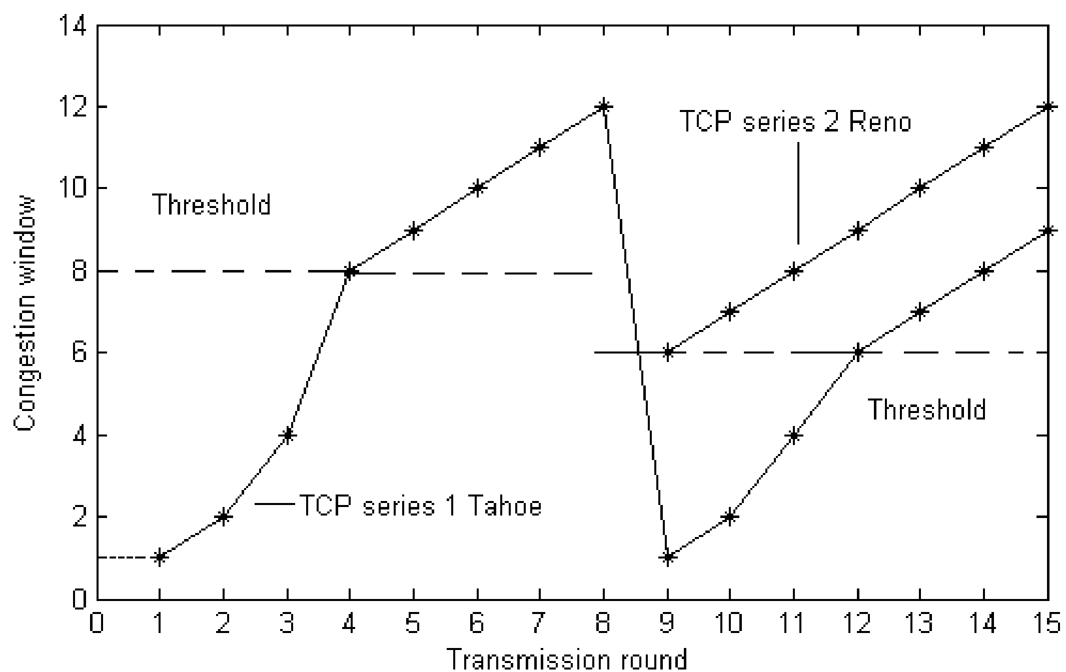

Figure 2.3: TCP's congestion window (Tahoe and Reno)

(Kurose & Ross 2005)

Figure 2.3 shows TCP's congestion window for both Tahoe and Reno. In this figure, the threshold is initially equal to 8 packets. The congestion window climbs exponentially during slow start and hits the threshold at the fourth round of transmission. After

hitting the threshold, TCP performs congestion avoidance. The congestion window then climbs linearly until a triple duplicate ACK occurs, just after transmission round 8. The congestion window is 12 packets when this loss event occurs. The new threshold is then set to half congestion window, that is 0.5*12= 6 packets. Under TCP Reno, the congestion window is set to 6 packets (new threshold) and then grows linearly. However, in TCP Tahoe, the congestion window is set to 1 packet and grows exponentially i.e. slow start (Kurose & Ross 2005, Jacobson 1988).

The fast retransmit algorithm first appeared in Tahoe release, and it was followed by slow start. The fast recovery algorithm appeared in Reno release (Stevens 1997, Allman & et al. 1999, Floyd & Henderson 1999).

To sum up, when timeout occurs, both TCP Reno and TCP Tahoe go back to slow start. When three duplicate ACK's are received, TCP Tahoe performs slow start, but TCP Reno performs congestion avoidance.

2.7 New Reno TCP

New Reno TCP (Floyd & Henderson 1999, Braden & et al. 1998) modifies the action taken when receiving new ACKs. In order to exit fast recovery, the sender must receive an ACK for the highest sequence number sent. Thus, new partial ACKs (those which represent new ACKs but do not represent an ACK of all outstanding data) do not deflate the usable window back to the size of the congestion window.

2.8 Sack TCP

The TCP may experience poor performance when multiple packets are lost from one window of data. With the limited information available from cumulative acknowledgments, a TCP sender can only learn about a single lost packet per round trip time. An aggressive sender could choose to retransmit packets early, but such retransmitted packets may have already been successfully received. A selective acknowledgment (SACK) (Floyd 1996) mechanism, combined with a selective repeat retransmission policy, can

help to overcome these limitations. The receiving TCP sends back SACK packets to the sender informing the sender of the data that has been received. The sender can then retransmit only the missing data packets.

2.9 Vegas TCP

The Vegas extends the Reno's retransmission mechanisms. At first, Vegas reads and records the system timestamp each time a packet is sent. This is used for following situations (Reddy & Rao 2006):

- When a duplicate acknowledgment is received, Vegas checks to see if the difference between the current time and the timestamp recorded for the relevant packet is greater than the timeout value. If it is true, then Vegas retransmit the packet without having to wait for duplicate acknowledgments.
- When a non duplicate acknowledgement is received and it is the first or second one after a retransmission, Vegas checks again to see whether the time interval since the packet is sent, is larger than the timeout value. If it is true that the time interval is larger than the timeout value, then Vegas retransmits the packet.

2.10 BIC and CUBIC TCP

The common concern is that TCP underperforms in those areas of application where there is a high bandwidth-delay system. The problem is that the additive window inflation algorithm used by TCP can be very inefficient in long-delay, high-speed environments like satellite transmission service.

Binary Increase Congestion Control (BIC) (Xu & et al. 2004) takes a different view, by assuming that TCP is actively searching for a packet sending rate that is on the threshold of triggering packet loss, and uses a binary search algorithm to achieve this efficiently.

BIC can be too aggressive in low RTT networks and in slower speed situations, thus leading to a refinement of BIC, namely CUBIC (Rhee & et al. 2008). CUBIC uses a third-order polynomial function to govern the window inflation algorithm, rather than the exponential function used by BIC. CUBIC can produce fairer outcomes in a situation of multiple flows with different RTTs. CUBIC also limits the window adjustment in any single RTT interval to a maximum value.

2.11 Westwood TCP

BIC and CUBIC concentrate on the rate increase function, attempting to provide for greater stability for TCP sessions as they converge to a long-term available sending rate. The other perspective is to examine the multiplicative decrease function, to see if there is further information that a TCP session can use to modify this rate decrease function (Huston 2006).

The approach taken by Westwood (Gerla & et al. 2001) is to concentrate on the halving by TCP of its congestion window in response to packet loss (as signaled by three duplicate ACK packets). The conventional TCP algorithm of halving the congestion window can be refined by the observation that the stream of return ACK packets actually provides an indication of the current bottleneck capacity of the network path, as well as an ongoing refinement of the minimum RTT of the network path. The Westwood algorithm maintains a bandwidth estimate by tracking the TCP acknowledgement value and the inter-arrival time between ACK packets in order to estimate the current network path bottleneck bandwidth. In the case of the Westwood approach, the bandwidth estimate is based on the receiving ACK rate, and is used to set the congestion window. The Westwood sender keeps track of the minimum RTT interval, as well as a bandwidth estimate based on the return ACK stream. In response to a packet loss event, Westwood does not halve the congestion window, but instead sets it to the bandwidth estimate times the minimum RTT value.

2.12 TCP Evolution

The evolution of TCP is to seek a point of delicate balance between self-optimization and cooperative behavior. Self-optimization is to optimize the use of bandwidth for one of the users in the link, while cooperative behavior is to consider the need of other users in the link and to give a fair share of the bandwidth of the link to other users.

The evolution of TCP must avoid making radical changes that may stress the deployed network into congestion collapse, and also must avoid a congestion control among competing protocols (Gerla & et al. 2001). The Internet architecture to date has been able to achieve new benchmarks of network efficiency. Much of the credit for this must go to the operation of TCP, which manages to work at that point of delicate balance between self-optimization and cooperative behavior.

Widespread deployment of transmission protocols that take a more aggressive position on self-optimization will ultimately lead to situations of congestion collapse. On the other hand, widespread deployment of conservative transmission protocols may lead to lower jitter and lower packet retransmission rates, but at a cost of considerably lower network efficiency (Floyd 2000).

The challenges faced with the evolution of TCP is to maintain a coherent control architecture that has consistent behavior within the network, consistent interaction with instances of data flows that use the same control architecture, but adequately flexible to adapt to differing network characteristics and different application profiles (Huston 2000).

2.13 Summary

Tahoe TCP utilizes the slow start, congestion avoidance and fast retransmit algorithms. After a connection is established, slow start is invoked. The congestion window increases exponentially. When the congestion window reaches the slow start threshold, ssthresh, congestion avoidance takes over. The congestion window then increases linearly.

Upon receiving three duplicate ACK's, Tahoe TCP retransmits the packet that appears to be lost, without waiting for the retransmission timer to expire. This is the fast retransmit algorithm.

Reno TCP includes fast recovery. In addition, Reno TCP does not return to slow start during fast retransmit, but reduces the congestion window to half its current value.

Each variation of TCP is designed to solve a special problem using a specific method, and is suitable for a particular application. For instance, Binary Increase Congestion Control TCP (BIC TCP) searches for a packet sending rate that is on the threshold of triggering packet loss, and uses a binary search algorithm to achieve this. BIC TCP is designed for use in long-delay, high speed environments like satellite transmission service.

In the next chapter, Round Trip Time (RTT) and Retransmission Timeout (RTO) and Jacobson/Karels Algorithm will be introduced. All these topics are closely related to TCP.

Exponential distribution and Pareto distribution will also be introduced in the next chapter, because in Chapter 7 simulations, Exponential and Pareto traffic sources will be used.

Chapter 3

Round Trip Time (RTT) and Retransmission Timeout (RTO)

3.1 Introduction

When timeout occurs, both TCP Reno and TCP Tahoe return to slow start, as mentioned in Section 2.6. At slow start, TCP begins transmission with one packet. It is obvious that the value of timeout affects the throughput of the transmission link. This chapter presents an in-depth review of retransmission timeout.

The reliability of delivery of data is guaranteed in Transmission Control Protocol (TCP) by the following procedure. When a host computer sends a packet, it starts a timer. If the host receives an acknowledgement (ACK) before the timer expires, it sends a new packet and restarts the timer. The time between the starting of the timer and the receiving of ACK is called the Round Trip Time (RTT) (Comer 2006*a*, Peterson & Davie 2000).

If the timer expires before the host receives an ACK, the host will retransmit the same packet again. The time between the starting of the timer and the expiring of the timer is called Retransmission Timeout (RTO) (Comer 2006*a*, Kurose & Ross 2005, Peterson & Davie 2000).

It is obvious that the RTO should be larger than the RTT; but how much larger should RTO be (Comer 2006*a*)? The answer to this question is the focus of this project.

This chapter starts with highlighting the difficulties in choosing RTO. Three algorithms are presented for the estimation of RTO, namely: the Original Algorithm specified by RFC 793 in 1981 (Postel 1981), the Karn/Partridge Algorithm specified by the publication, "Improving Round-Trip-Time Estimates in Reliable Transport Protocols", in 1987 (Karn & Partridge 1995); and the Jacobson/Karels Algorithm specified by the paper "Congestion Avoidance and Control" published in 1988 (Jacobson 1988). Then there is a section on the historical development of RTO. Finally, Exponential distribution and Pareto distribution are introduced.

3.2 Historical Development of RTO

Retransmission timeout (RTO) estimation is important for data transmission in the Internet so it is worthwhile to investigate the historical development of RTO and the problems involved in RTT estimation.

The TCP specification that specifies the first original algorithm (Section 3.4), is RFC 793, which was published in 1981 (Postel 1981).

The first original algorithm, after several years of implementation in the Internet, was found to have ambiguity in the measurement of RTTs that were retransmitted (Section 3.5).

In 1987, Phil Karn of Bell Communication Research, and Craig Partridge of Harvard University, found that TCP was suffering from a problem they called retransmission ambiguity.

They presented a novel and effective method to clarify this retransmission ambiguity problem, in their paper, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", (Karn & Partridge 1995). Their method could mitigate congestion in the Internet, but, it was Jacobson and Karels, who later addressed congestion in the

Internet more effectively.

At the time RFC 793 and clarification of retransmission ambiguity were implemented in the Internet, TCP used window based flow control, as a means for the receiver to restrict the amount of data sent by the sender. This flow control was used to prevent overflow of the receiver's data buffer, available for that connection. TCP assumes that packets could be lost due to either errors or congestion, but did not implement any dynamic mechanism to adjust the flow control window in response to congestion.

In October, 1986, Van Jacobson reported the first Internet congestion collapse. The data throughput from Lawrence Berkeley Laboratory to University of California Berkeley, which are 400 yards (1 yard = 0.92 meter) apart, dropped from 32 Kbps to 40 bps. Van Jacobson and Mike Karels started to investigate why the drop was so big (Jacobson 1988).

In 1988, Van Jacobson published his paper, "Congestion Avoidance and Control", (Jacobson 1988). Although the paper was published under the name of Van Jacobson, Jacobson acknowledged in his paper that the algorithms and ideas described in his paper were developed in collaboration with Mike Karels of University of California Berkeley, Computer System Research Group.

In his paper, Jacobson presented the Jacobson/Karels Algorithm which was described in Section 3.6, in addition to slow-start, congestion avoidance, and fast retransmit.

Slow-start, congestion avoidance and fast retransmit were described in Chapter 2, and Jacobson/Karels Algorithm was described in Chapter 3. These descriptions give the impression that Chapter 2 and Chapter 3 are separate issues. It should be pointed out that Chapter 2 and Chapter 3 are related issues. They are presented in two separate chapters for the reason of easy understanding.

3.3 Difficulties in Choosing RTO

Some of the difficulties in choosing an appropriate RTO are stated below:

- The range of variation in RTTs between two host computers is great (Peterson & Davie 2000);
- If RTO is just a little larger than RTT, retransmission will be too often, and that increases congestion in the transmission path (Comer 2006*a*);
- If RTO is much larger than RTT, the response to retransmit when a packet is lost, will be too slow; and this will decrease the throughput of the transmission path (Comer 2006*a*).

The RTO now implemented in TCP was proposed by Jacobson in 1988 (Comer 2006*a*, Peterson & Davie 2000); but before this Jacobson/Karels algorithm, there were the original algorithm and Karn/Partridge algorithm. These three algorithms are described in the following sections in chronological order.

3.4 Original Algorithm

This section presents the first original algorithm prescribed in the TCP specification (Peterson & Davie 2000). When TCP sends a packet, it starts a timer. When an ACK of that packet is received, TCP stops the timer and reads the time, which is taken to be the Sample RTT. A new Estimated RTT is the weighted average of the old Estimated RTT and the Sample RTT. Expressing the above idea in formula:

$$A_i = (1 - G_1)A_{i-1} + G_1 M_{i-1} \tag{3.1}$$

where A_i is the new Estimated RTT;
A_{i-1} is the old Estimated RTT;
M_{i-1} is the Sample RTT.
G_1 is a constant, which is typically equal to 0.125. (0.125=1/8).

Therefore,
New Estimated RTT = (0.875)(Old Estimated RTT) + (0.125)(Sample RTT)
The RTO is then calculated by the simple formula

$$RTO = (\beta)(NewEstimatedRTT) \tag{3.2}$$

The first original specification recommended $\beta = 2$.

References for the above formulas are from RFC 793.

3.5 Karn/Partridge Algorithm

The first original algorithm, after several years of implementation in the Internet, was found to be inadequate. Problems arose when there was retransmission.

When a packet was retransmitted and then an ACK was received, it was impossible to know whether this ACK was associated with the original transmission or the retransmission.

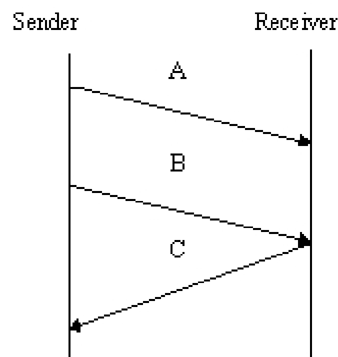

Figure 3.1: ACK for retransmission

In Figure 3.1, **A** represents original transmission; **B** represents retransmission; **C** represents ACK. If it was assumed that the ACK was for the original transmission, but it was actually for the retransmission as shown in Figure 3.1, then the calculated Sample RTT would be too large (Kurose & Ross 2005, Peterson & Davie 2000).

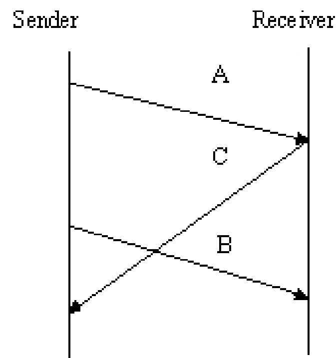

Figure 3.2: ACK for original transmission

In Figure 3.2, **A** represents original transmission; **B** represents retransmission; **C** represents ACK. If it was assumed that the ACK was for retransmission, but it was actually for the original transmission as shown in Figure 3.2, then the calculated Sample RTT would be too small (Kurose & Ross 2005, Peterson & Davie 2000).

In 1987, Karn and Partridge proposed the Karn/Partridge algorithm, which is described below:

- TCP calculated Sample RTT only for packets that were sent once, and did not calculate Sample RTT for packets that were sent twice, that is, for packets that were retransmitted (Kurose & Ross 2005, Peterson & Davie 2000).
- Whenever there was retransmission, TCP used the following formula to calculate RTO:

$$R_i = (2)(R_{i-1}) \tag{3.3}$$

R_i Next retransmission timeout

R_{i-1} Last retransmission timeout

It was found that Karn/Partridge algorithm could mitigate congestion in the Internet (Kurose & Ross 2005, Peterson & Davie 2000).

3.6 Jacobson/Karels Algorithm

In 1988, two researchers, Jacobson and Karels, proposed their Jacobson / Karels Algorithm (Comer 2006*a*, Peterson & Davie 2000), which is described by the following formulas (3.4), (3.5), (3.6), (3.7):
Firstly, find the difference between the Sample RTT and the old Estimated RTT.

$$\text{Difference} = M_{i-1} - A_{i-1} \tag{3.4}$$

where M_{i-1} is the Sample RTT (Kurose & Ross 2005, Peterson & Davie 2000).
Secondly, find the new Estimated RTT by Equation (3.5).

$$\begin{aligned} A_i &= A_{i-1} + (G_1)(\text{Difference}) \\ &= A_{i-1} + (G_1)(M_{i-1} - A_{i-1}) \\ &= A_{i-1} + G_1 M_{i-1} - G_1 A_{i-1} \\ &= (1 - G_1)A_{i-1} + G_1 M_{i-1} \end{aligned} \tag{3.5}$$

G_1 is a constant, which is typically equal to 0.125 (Kurose & Ross 2005, Peterson & Davie 2000, Ma & et al. 2004). (0.125=1/8) The A_i described in this section is the same as that presented in Original Algorithm. Thirdly, find the new deviation from the old deviation by Equation (3.6).

$$\begin{aligned} V_i &= V_{i-1} + G_2(|\text{Difference}| - V_{i-1}) \\ &= V_{i-1} + G_2|\text{Difference}| - G_2 V_{i-1} \\ &= (1 - G_2)V_{i-1} + G_2|\text{Difference}| \end{aligned} \tag{3.6}$$

where V_i is the new deviation;
V_{i-1} is the old deviation;
|Difference| is absolute value of Difference;
G_2 is a constant, which is typically equal to 0.25 . (0.25=1/4)
(Kurose & Ross 2005, Peterson & Davie 2000, Ma & et al. 2004)

Finally, find the next retransmission timeout from the new Estimated RTT and new deviation.

$$R_i = A_i + KV_i \tag{3.7}$$

where R_i is the next retransmission timeout;

V_i is the new deviation

K is a constant, which is typically equal to 4.

(Kurose & Ross 2005, Peterson & Davie 2000, Ma & et al. 2004)

3.7 More about Jacobson/Karels Algorithm

Jacobson performed a practical experiment, and plotted RTT and RTO in the same figure, in order to prove that his Algorithm is superior to RFC 793.

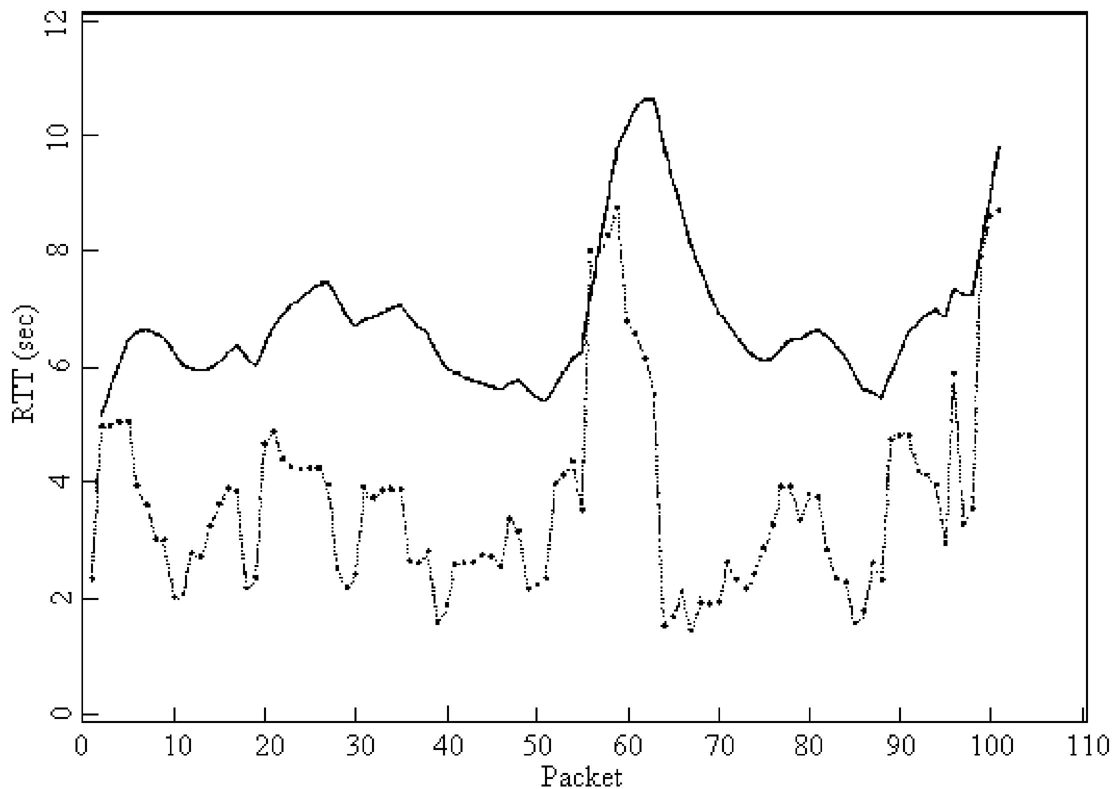

Figure 3.3: RTO computed by RFC 793 rules (Jacobson 1988)

Figure 3.3 shows trace data representing per-packet round trip time on a well-behaved Arpanet connection. The x-axis is the packet number. Packets were numbered sequentially, starting with one. The y-axis is the elapsed time from the send of the packet to the sender's receipt of its ACK, i.e. the RTT. During this portion of the trace, no packets were dropped or retransmitted (Jacobson 1988).

The dotted line shows the RTTs. The solid line shows the RTOs computed according to the rules of RFC 793 (Jacobson 1988).

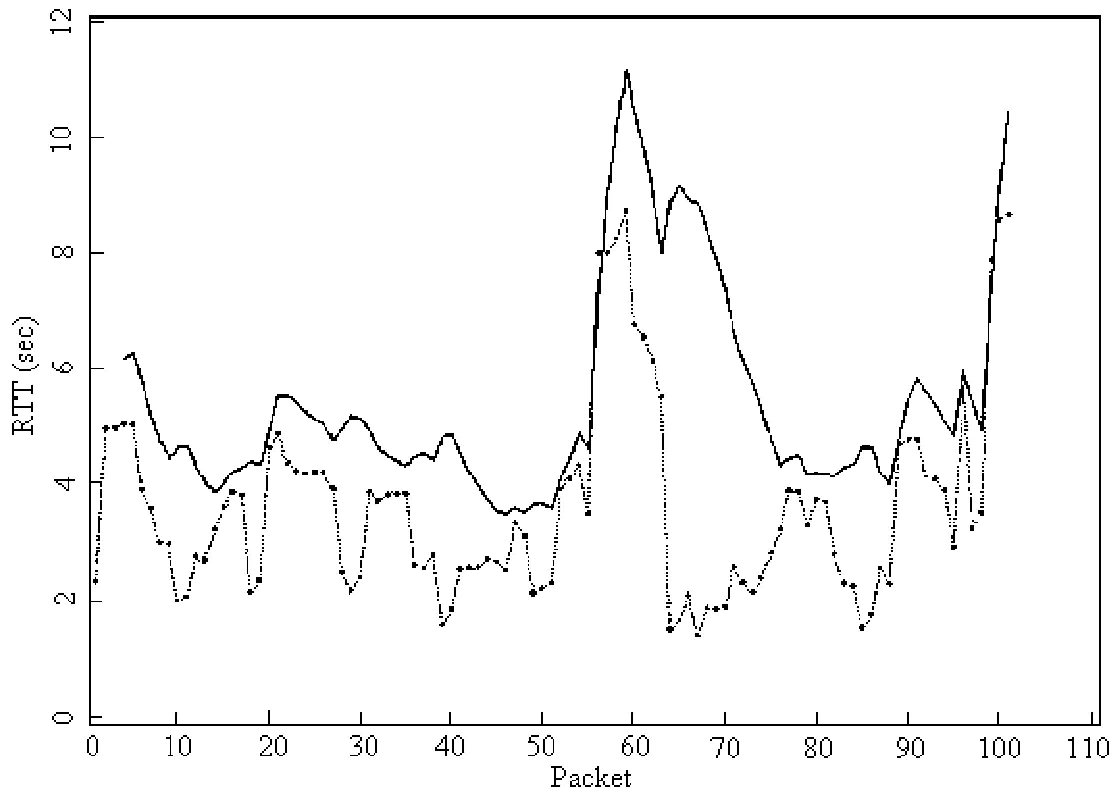

Figure 3.4: RTO computed by Jacobson/Karels Algorithm (Jacobson 1988)

Figure 3.4 shows the same data as above, but the solid line shows RTOs computed according to Jacobson/Karels Algorithm (Jacobson 1988).

When Figure 3.3 and Figure 3.4 are compared, it is found that the shape of the solid line in Figure 3.4 resembles the shape of the dotted line. Also, the solid line of RTOs is closer to the dotted line of RTTs.

In Chapter 7, simulations of TCP using median filter are conducted. By plotting RTTs and RTOs in the same figure, RTTs and RTOs resemblance in shapes, and closeness in lines. Our simulations in Chapter 7 are supported by Figure 3.3 and Figure 3.4.

The improvements made by Jacobson/Karels Algorithm vary from noticeable to dramatic. This Algorithm is especially important on low-speed link, where the deviation in packet sizes causes a large deviation in RTT. [Here, the deviation is:

RTO = (new Estimated RTT) + (4)(new deviation)]

In a 9.6 kb link, the link utilization could go from 10% to 90% as a result of implementing Jacobson/Karels Algorithm (Braden 1989*b*).

For a new connection, the initializations should be:
RTT = 0 second
RTO = 3 seconds

The lower bound for RTO should be in fractions of a second (to accommodate high speed LANs), and the upper bound should be 240 seconds (Braden 1989*b*).

Jacobson and Karels made a great contribution to prevent congestion collapse in today's Internet.

3.8 Problems with RTT Estimation

The first problem with RTT estimation is described as follows:
RFC 793 suggested finding:
New Estimated RTT = (7/8)(Old Estimated RTT) + (1/8)(Sample RTT)
and then calculated
RTO = (β)*(New Estimated RTT).*
The suggested β=2 could adapt to loads of at most 30% (Jacobson 1988). Above this point, a connection would respond to load increases by retransmitting packets that had only been delayed in transit (Jacobson 1988). This would cause a more serious congestion in the network.

There is still another problem to be discussed below:

$A_i = (1 - G_1)A_{i-1} + G_1 M_{i-1}$

M_{i-1} Sample RTT

G_1 Constant, typically equal to 1/8

$A_i = A_{i-1} - G_1 A_{i-1} + G_1 M_{i-1}$

$A_i = A_{i-1} + G_1(M_{i-1} - A_{i-1})$

The last expression above states that a new prediction (A_i) is made, based on the old prediction (A_{i-1}), plus a fraction (G_1=1/8) of the prediction error ($M_{i-1} - A_{i-1}$).

The prediction error is the sum of two components:
1. Error due to noise in the measurement, which is random and unpredictable, such as fluctuations in competing traffic. This part is denoted by E_r.
2. Error due to a poor choice of A_{i-1}. This part is denoted by E_e.

Then $A_i = A_{i-1} + G_1 E_r + G_1 E_e$
The $G_1 E_e$ term moves A_i in the correct direction while $G_1 E_r$ moves A_i in a random direction (Jacobson 1988).

If A_i follows a Gaussian distribution (also called normal distribution), $G_1 E_r$ will cancel one another after a number of samples, and A_i will converge to the correct value. Unfortunately, A_i does not follow a Gaussian distribution. Instead, A_i follows a heavy-tailed distribution.

3.9 Exponential Distribution and Pareto Distribution

In 1995, two researchers Paxson and Floyd found that FTP data connections had bursty arrival rates. In addition, the distribution of the number of bytes in each burst has a heavy right tail (Paxson & Floyd 1994).

In statistics, heavy-tailed distributions are probability distributions whose tails are not exponentially bounded (Asmussen 2003). That is, they have heavier tails than the exponential distribution.

One of the simplest heavy-tailed distributions is the Pareto distribution. The Pareto distribution is named after its creator, the Italian economist Vifredo Pareto (Reed 2003).

Since exponential and Pareto traffics will be used in Chapter 7 simulations, it is necessary to understand their pdf's.

Exponential distribution (Drossman & Veirs 2002) is defined as:

$$N(x) = \begin{cases} \frac{N_0}{e^{\alpha x}} & \text{for } x \geqslant 0 \\ \\ 0 & \text{elsewhere} \end{cases} \tag{3.8}$$

In order to see the similarity between Exponential distribution and Pareto distribution, *N(x)* is simplified as follows:

Let $\alpha = 1$.

Then, $N(x) = \frac{N_0}{2.7^x}$ where N_0, the value of N(x) at time x = 0, is a constant. Later, it will be shown that one example of Pareto distribution is : $f(x) = \frac{Constant}{x^{2.7}}$

Now Pareto distribution will be introduced. The definition of Pareto distribution (Freund & Walpole 1987) is:

$$f(x) = \begin{cases} \frac{\alpha k^{\alpha}}{x^{\alpha+1}} & \text{for } x > k \\ \\ 0 & \text{elsewhere} \end{cases} \tag{3.9}$$

Let us consider the simple case where k = 1.

$$f(x) = \begin{cases} \frac{\alpha}{x^{\alpha+1}} & \text{for } x > 1 \\ \\ 0 & \text{elsewhere} \end{cases} \tag{3.10}$$

Let $\alpha = 1.7$
$f(x) = \frac{1.7}{x^{2.7}}$

There is similarity between exponential distribution and Pareto distribution. The parameter k specifies the minimum value of x. The parameter α determines the shape of f(x). In the simulations of Chapter 7, when the shape of Pareto is set, it is actually setting the value of α.

Exponential distribution and Pareto distribution in Figure 3.5 are plotted for comparison. In Figure 3.5, Pareto distribution decreases much more slowly than exponential

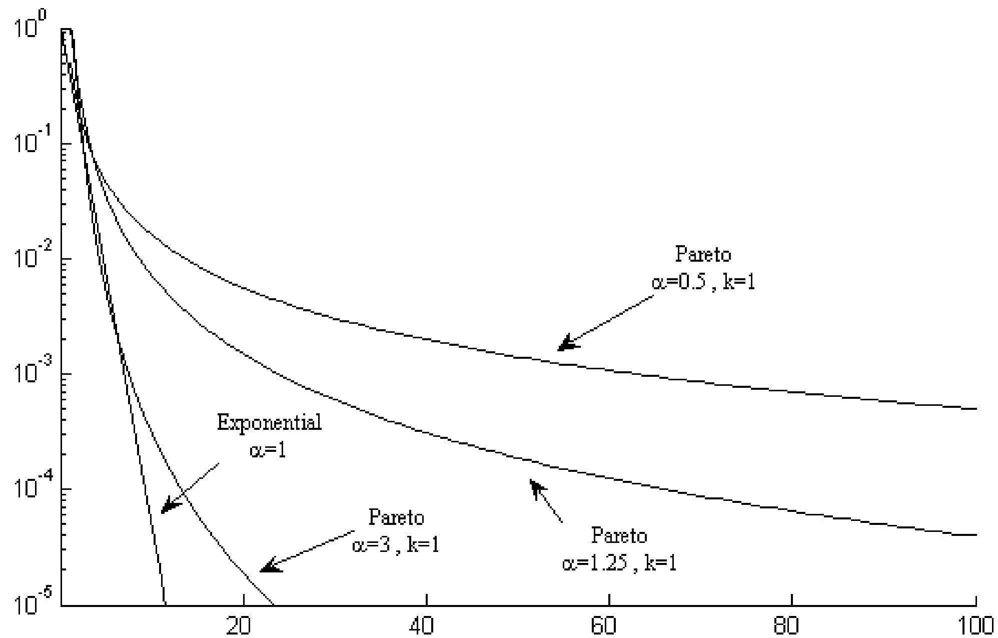

Figure 3.5: Exponential and Pareto probability density functions (Hei 2001)

distribution. In other words, Pareto distribution has a heavier tail than exponential distribution.

In 1997, Willinger *et al.* (Willinger & et al. 1997) generated Ethernet traffic by superposition of many Pareto traffic sources, setting them ON and OFF. During ON period, the source transmits a burst of packets. During OFF period, the source is in idle and no packets are transmitted.

In Chapter 7, the ON period, OFF period and shape of the Pareto traffic are set in our simulations.

This sub-section is to support our objective to use median filter to replace weighted-average in estimating RTT.

3.10 Summary

This chapter builds up theoretical foundations to support our network simulations in Chapter 7. In Section 3.4, the original algorithm states that a new Estimated RTT is the weighted average of the old Estimated RTT and the Sample RTT. The RTO is then calculated as two times new Estimated RTT. Our proposal in Chapter 7 is that, a new Estimated RTT is the median of the latest five Sample RTT's; and the RTO is then calculated as 1.25 times new Estimated RTT. What will be done in Chapter 7, is to replace weighted-average filter with median filter.

Jacobson had performed a practical experiment, and plotted RTT and RTO in the same figure to show that his algorithm is superior to RFC 793 (Refer to Section 3.2). This idea will be used to plot RTT and RTO in the same figure, to show that our proposal of using median filter, is better than weighted-average filter in Chapter 7.

In Section 3.7, problem with RTT Estimation is based on Jacobson's paper, "Congestion Avoidance and Control" (Jacobson 1988). The analyses of the problems give supportive reason to our objective to use median filter to replace weighted-average in estimating RTT.

Chapter 4

The Median Filter and its Modifications

4.1 Introduction

Since one of the objectives of this project is to apply the median filter to find a better round trip time (RTT), it is appropriate to review some literature on these topics. A literature review of these topics is presented below. Linear filter does not preserve sharp edges of signals well (Yin & et al. 1996), and it cannot attenuate impulsive noise totally. The nonlinear median filter can overcome these two problems; it can preserve sharp edges and at the same time, attenuate impulsive noise very well (Nodes & Gallagher Jr. 1982).

Topics presented in the following sections are:

- Median Filters
- Ranked-order Filters
- Recursive Median Filters
- Weighted Median Filters
- Recursive Weighted Median Filters

- Symmetric Weighted Median Filters
- Centre Weighted Median Filters
- Adaptive Weighted Median Filters

4.2 Median Filter

Median Filter was proposed by J. Tukey in 1974 (Tukey 1974). It is a nonlinear filter and the principle of superposition that applies to linear filter does not apply to it (Yin & et al. 1996). An array is input to the median filter. The input array is sorted either in ascending or descending order. The output of the median filter is the middle element of the sorted input array.

Properties of Median Filters

Signals invariant to further median filtering is called the root signal (Arce & Gallagher Jr. 1982).

A finite number of repeated median filtering of a signal of finite length will yield a root signal (Yin & et al. 1996). This property of median filters is called the convergence property.

A root signal remains unchanged after further median filtering (Burian & Kuosmanen 2002). A root signal of a median filter of length *2k+1*, is also a root signal of median filter of length less than *2k+1* (Nodes & Gallagher Jr. 1982).

For a median filter of length *2k+1*,

$$X_{(n)} = median\left[X_{(n-k)}, \cdots, X_n, \cdots, X_{(n+k)}\right] \tag{4.1}$$

$X_{(n)}$ is called the root signal of that particular median filter, if the above condition is satisfied for all n (Yin & et al. 1996, Mitra & Kaiser 1993).

The two important properties of the median filter are:

(a) edge preservation,

(b) impulse suppression.

Since impulse has high frequency content, and median filter attenuates it, median filter is a low-pass filter (Arce & Stevenson 1987).

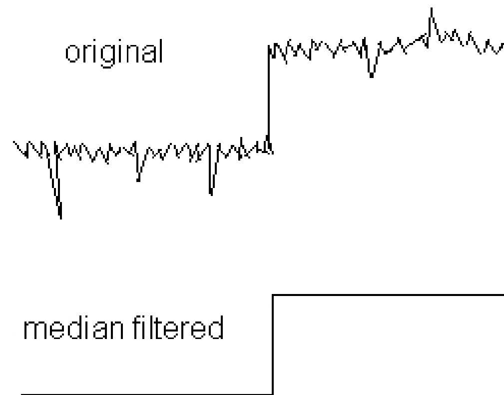

Figure 4.1: Median Filters suppress impulse and at the same time preserve edge

(Mitra & Kaiser 1993)

In digital image processing, positive and negative impulses appear in the photograph as white and black dots (or black and white dots). This impulsive noise is given the name **salt and pepper noise** because salt is white and pepper is black in colour. Salt and pepper noise can be removed by using median filter.

Example for Median Filter (Pratt 1978):

Apply median filter with window size of 3 to the array *x(n)*, where *x(n) = [2 80 6 3]*.

Solution

y[1] = Median [2 2 80] = 2

y[2] = Median [2 80 6] = Median [2 6 80] = 6

y[3] = Median [80 6 3] = Median [3 6 80] = 6

y[4] = Median [6 3 3] = Median [3 3 6] = 3

The original signal *x(n) = [2 80 6 3]*, after being filtered by median filter, becomes *z(n) = [2 6 6 3]*. In finding *y[1]*, the beginning point is appended, which is *2*. In finding *y[4]*, the ending point is appended, which is *3*. The appending of the beginning and end points is to account for the start and end effects (Clarkson & Stark 1995, Arce & Stevenson 1987).

Some concepts presented by Gallapher and Wise (Gallagher Jr. & Wise 1981) related to root signals are summarized below:

For a median filter of length *2k+1*,

- a constant neighbourhood is a region of at least *k+1* consecutive identically valued points (Gallagher Jr. & Wise 1981);
- an edge is a monotonically rising or falling set of points surrounded on both sides by constant neighbourhoods of different values (Gallagher Jr. & Wise 1981);
- an impulse is a set of at least one but less than *k+1* points (in other words, an impulse is a set of at least one and at most k points), whose values are different from the surrounding regions, which are identically valued constant neighbourhoods (Gallagher Jr. & Wise 1981);
- an oscillation is any signal structure that is not part of a constant neighbourhood, an edge or an impulse (Gallagher Jr. & Wise 1981).

4.3 Ranked-order Filters

The ranked-order filter was first introduced by Arce and Stevenson in 1987. It takes the *i-th* largest sample from the input data set, as its output. $Y_{(n)}$ = *i-th* largest value of the set $[X_{(n-k)}, ..., X_{(n)}, ..., X_{(n+k)}]$ where $Y_{(n)}$ is the output, and $X_{(n)}$'s are the inputs (Yin & et al. 1996).

Maximum filter and minimum filter are two special cases of ranked-order filter. Maximum filter and minimum filter take the maximum sample and the minimum sample from the input data set, as their outputs respectively (Yin & et al. 1996).

Chronologically, median filter was proposed before ranked-order filter; but median filter can be treated as a member of the ranked-order filter family (Arce & Stevenson 1987). In other words, it can be said that ranked-order filter was developed first, and then developed a special case, the median filter.

Example for Ranked-order Filter (Poularikas 1999):

Given input data set $\boldsymbol{X}$ = *[4,14,18,40,10]*

Find the outputs of the ranked-order filter.

Solution Sort $\boldsymbol{X}$ and store in $\boldsymbol{X}$-ordered.

$\boldsymbol{X}$*-ordered = [4, 10, 14, 18, 40]*

For *rank r = 1, the output = 4.*

For *rank r = 2, the output = 10.*

For *rank r = 3, the output = 14 = median*

For *rank r = 4, the output = 18.*

For *rank r = 5, the output = 40.*

4.4 Recursive Median Filters

Recursive median filter is a modification of median filter. It uses the previously derived output samples to replace some of the input samples. A recursive median filter of length *2k+1* is defined by the following equation (Yin & et al. 1996, Mitra & Kaiser 1993):

$$Y_{(n)} = median\left[Y_{(n-k)}, \cdots, Y_{(n-1)}, X_n, X_{(n+1)}, \cdots, X_{(n+k)}\right] \tag{4.2}$$

where Y_n is output;

$Y_{(n-k)}, \cdots, Y_{(n-1)}$ are the previous output samples;

$X_{(n)}, X_{(n+1)}, \cdots, X_{(n+k)}$ are the input samples.

Recursive median filtering is carried out in two steps.

- The previously derived output is used to replace the old input.
- Then the window of the recursive filter is moved to the next position (Yin & et al. 1996, Mitra & Kaiser 1993).

4.4.1 Properties of Recursive Median Filter

For the same number of filtering operations, the recursive median filter provides better smoothing effect than the original median filter, but the input signal is distorted more after recursive median filtering (Yin & et al. 1996, Burian & Kuosmanen 2002).

If a median filter can produce a root signal after one filtering, the median filter is said to be idempotent (Haavisto & et al. 1991). One recursive median filtering operation will cause an input signal to become the root signal (Mitra & Kaiser 1993). The recursive median filter is described as idempotent for possessing this ability (Yin & et al. 1996, Burian & Kuosmanen 2002). Recursive median filter is idempotent when filtering one-dimensional signals, but this is not generally true for two-dimensional signals.

Repeated non-recursive median filtering operations also cause an input signal to become the root signal. However, the root signals produced by non-recursive and recursive operations are generally not the same (Yin & et al. 1996, Mitra & Kaiser 1993, Burian & Kuosmanen 2002).

4.5 Weighted Median Filters

When a particular element in the input array is more important than the other elements, it is duplicated in the input array. The number of times for which it is duplicated is called the weight of that particular element. For example, in the input array of [1, 2, 3, 4, 5], the element 2 is more important. It is duplicated three times to reflect its importance. The array then becomes [1, 2, 2, 2, 3, 4, 5]. The weight of 2 is said to be three.

The weighted median filter is a modification of the standard median filter. The structure of weighted median filter is similar to that of Finite Impulse Response (FIR) filter, but FIR is linear (Yin & et al. 1996, Mitra & Kaiser 1993). Recall that weighted median filter is non-linear. For real-valued signals, weighted median filter can be defined in two different ways, which give the same output (Yin & et al. 1996, Mitra & Kaiser 1993).

The first definition of weighted median filter is stated below (Yin & et al. 1996, Mitra & Kaiser 1993):

Let $\boldsymbol{X}$ = $[X_1,X_2,\cdots,X_N]$ be a discrete-time continuous-valued input vector,

$\boldsymbol{W}$ = $[W_1,W_2,\cdots,W_N]$ be the integer weights. Then the output

$$\boldsymbol{Y} = median[W_1 \Diamond X_1, W_2 \Diamond X_2, \cdots, W_N \Diamond X_N] \tag{4.3}$$

where the $\Diamond$ denotes duplication, that is,

$$A \Diamond X = \underbrace{X, \cdots, X}_{\text{A times}} \tag{4.4}$$

The second definition of weighted median of X is stated as follows:

The weighted median is β that minimizes the sum

$$\sum_{i=1}^{N} W_i |X_i - \beta| \tag{4.5}$$

where X_i and W_i are defined as in the first definition (Yin & et al. 1996, Mitra & Kaiser 1993). β is chosen from $\boldsymbol{X}$.

Example of Weighted Median Filter (Poularikas 1999):

Given the input vector $\boldsymbol{X}$ = *[4, 14, 18, 40, 10]*

and the weight $\boldsymbol{W}$ = *[-0.05, 0.1, 0.15, 0.1, 0.05]*

find the output of the weighted median filter.

Solution

Put $\boldsymbol{X}$ and $\boldsymbol{W}$ in small brackets and store in $\boldsymbol{Y}$.

$\boldsymbol{Y}$ = *[(4, -0.05), (14, 0.1), (18, 0.15), (40, 0.1), (10, 0.05)]*

Move the negative sign in weight to input data.

$\boldsymbol{Y}$ = *[(-4, 0.05), (14, 0.1), (18, 0.15), (40, 0.1), (10, 0.05)]*

Sort $\boldsymbol{Y}$ according to $\boldsymbol{X}$ and store in $\boldsymbol{Z}$.

$\boldsymbol{Z}$ = *[(-4, 0.05), (10, 0.05), (14, 0.1), (18, 0.15), (40, 0.1)]*

Find the sum of weight and then divide it by two.

Halfsum = (0.05 + 0.05 + 0.1 +0.15 +0.1)/2

Halfsum = 0.225

Add the weights 0.05 + 0.05 + 0.1 +0.15 = 0.35 until the sum of the weights is larger than the halfsum. The last weight 0.15 belongs to 18, so 18 is the weighted median filter output.

4.6 Recursive Weighted Median Filters (RWMF)

The recursive weighted median filter is obtained by combining the recursive median filter and weighted median filter.

When filtering one-dimensional signals, RWMF is usually not idempotent (Burian & Kuosmanen 2002). RWMF is better than Infinite Impulse Response (IIR) filter because bandpass RWMF and highpass RWMF can be designed to have good stopband characteristics which IIR cannot attain (Arce & Paredes 2000).

RWMF is similar to IIR filter in structure (Arce & Paredes 2000). The difference equation of IIR filter is stated below:

$$Y(n) = \sum_{l=1}^{N} A_l Y(n-l) + \sum_{k=-p}^{q} B_k X(n-k) \tag{4.6}$$

where $Y(n)$ is the output;
$Y(n-l)$ is the previous output;
$X(n-k)$ is the input;
A_l is the weight called feedback coefficient; and
B_k is the weight called feedforward coefficient.

A total of (N+p+q+1) coefficient is needed to define the recursive difference equation. The following replacements can be used to obtain the RWMF (Arce & Paredes 2000).

- Replace summation sign by median operation.
- Replace multiplication weighting by signed duplication weighting.

The equation of RWMF obtained by these replacements is stated as follows (Arce &

Paredes 2000):

$$Y(n) = median\left[|A_l| \Diamond sgn(A_l) Y(n-l) \Big|_{l=1}^{N} + |B_k| \Diamond sgn(B_k) X(n-k) \Big|_{k=-p}^{q} \right] \quad (4.7)$$

where *sgn* denotes the sign function defined as:

sgn(X) = +1 *if X≥0*

sgn(X) = −1 *if X<0*

Example:

Given $Y(n) = median[Y(n-1), X(n), X(n+1), X(n+2), X(n+3)]$

$Y(n)$ Output

$Y(n-1)$ Previous output

$X(n), X(n+1), X(n+2), X(n+3)$ Input samples

and given $Y(n) = median[4, 14, 18, 40, 10]$,

and given weight $\boldsymbol{W} = [-0.05, 0.1, 0.15, 0.1, 0.05]$,

find the output *Y(n)*.

Solution

The answer is the same as that given in weighted median filter, so *Y(n)=18*.

The only difference is that the first data $Y(n-1)$, which is 4, comes from previous output.

4.7 Symmetric Weighted Median Filters

A weighted median filter is symmetric if the integer weights

$$W = \left[W_{-N} \cdots, W_{-1}, W_0, W_1 \cdots, W_N \right] \text{ is symmetric (Yin \& et al. 1996).} \quad (4.8)$$

That is W_{-i}=W_i for i = 1, 2,...N.

4.8 Centre Weighted Median Filters (CWMF)

In centre weighted median filter, weight is only assigned to the centre input sample, and not to the rest of the input samples (Yin & et al. 1996).

In other words, only the centre sample has a weight larger than one and all other samples have weights equal to one (Burian & Kuosmanen 2002).

CWMF is specified by its filter length and centre weight, and it can be designed to possess good noise attenuation while preserving details of the original image (Burian & Kuosmanen 2002).

4.9 Adaptive Weighted Median Filters

In digital image processing, the weights in weighted median filter are assigned as a compromise between edge preservation and noise attenuation. Thus, weighted median filter is optimum for the whole image but not for local areas. A solution to overcome this drawback of weighted median filter is to assign adaptive weights according to local characteristics of the image. It is possible to design an adaptive weighted median filter whose weights change according to the local characteristics of the image, so that both maximum noise attenuation and maximum edge preservation can be achieved (Meguro & Taguchi 1996).

When an adaptive weighted median filter is applied to a particular situation, the weights can be changed according to that special application.

A simple example of adaptive weighted median filter is given here.

In this simple example of adaptive weighted median filter, weights are assigned according to the following rules:

- Weight is assigned only to the immediate input sample I_k, while keeping the weights of all other samples equal to one (Ma & et al. 2003).

- The value of the weight assigned to I_k, depends on the rank of the sample I_k, among all other samples, which are all inside the observation window (Ma & et al. 2003).

The weight w of I_k is assigned by the rule:

$$w = L + 1 - r_k \tag{4.9}$$

where L is the number of samples inside the observation window (i.e. there are L samples); r_k is the rank of the sample I_k.

Case 1:

If there are 7 samples, then L = 7.

If I_k, is ranked 1, then r_k = 1.

w= 7 + 1 - 1 = 7

The weight assigned to I_k is 7.

Case 2:

If L is still 7, and I_k is ranked 7, and therefore r_k = 7.

w = 7 + 1 - 7

w = 1

In this case, the weight assigned to I_k is 1, so the adaptive weighted median filter is reduced to median filter.

The above adaptive weighted median filter can be applied to Random Early Drop gateway (Floyd & Jacobson 1993).

Example 1:

If the immediate queue size I_k of the gateway is the smallest inside the observation window, this can be interpreted as no congestion. By following this trend, the largest possible weight is assigned to I_k.

If the queue sizes are sorted in ascending order, with the smallest sample being put on the left and the largest sample being put on the right, then the rank r_k of I_k is 1.

$$\begin{aligned} w &= L + 1 - r_k \\ &= L + 1 - 1 = L, \text{ which is the largest possible weight.} \end{aligned}$$

In this way, when I_k is the smallest, it is assigned the largest weight L. For example, if there are totally 7 samples, the weight assigned is 7.

Example 2:

Next, suppose the immediate queue size I_k of the gateway is the largest inside the observation window, it is interpreted as congestion. This time, the trend will not be followed. The smallest possible weight is assigned to I_k.

If there are L samples inside the window, and I_k is the largest, then the rank r_k of I_k is L.

$$\begin{aligned} w &= L + 1 - r_k \\ &= L + 1 - L = 1 \end{aligned}$$

The weight assigned to I_k will be 1, which is the smallest possible weight.

4.10 Summary

The median filter has been recognized as a useful filter due to its edge preserving and impulse suppressing characteristics. Thus the median filter is often utilized for removing impulsive noise (Astola & Kuosmanen 1997, Pitas & Venetsanopoulos 1990). The weighted median filter (Brownrigg 1984, Justusson 1981, Loupos & et al. 1989), which is an extension of the median filter gives more weight to some samples within the window. It emphasizes specific input samples, because in some applications, not all samples are equally important. A special case of the weighted median filter is the center weighted median filter (Ko & Lee 1991), which gives more weight only to the central value of a window because the central value is the one that is most correlated with the desired estimate.

The median filter, as well as its modifications and generalizations (Bovik & et al. 1983) are typically implemented for image processing to remove impulse noise from images while preserving the image details. In the Internet, the RTT's are impulsive. Based on the impulse suppressing characteristic of the median filter, it is expected that median filter can perform better than the weighted-average filter, in the estimation of RTT.

Chapter 5

Experimental Evaluation of Retransmission Characteristics

5.1 Introduction

Chapter 5 starts with a description on how the experiment is set up. An analysis of the experimental results is then presented. A comparison on different round trip time and a comparison on different retransmission timeout are performed between median filter and weighted-average algorithm. The histograms of different RTT values are shown. Finally the modes and outliers of the histograms are organized in a table for easy understanding.

5.2 Experimental Environment

A practical experiment is performed to test the effectiveness of median filter, when it is applied to estimate the RTO.

A simple point-to-point wireless connection is established between a laptop computer and a desktop computer. The distance between the two computers is around one meter, and the transmission rate is 24 Mbps.

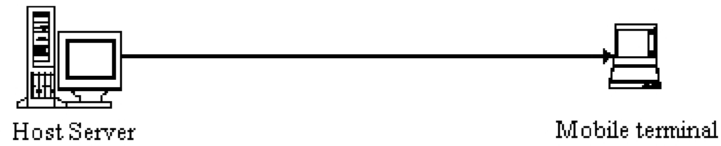

Figure 5.1: A point-to-point wireless connection

A network analyzer software is activated on the desktop computer to monitor and gather information about the network traffic. To simulate a wireless connection disturbance, the laptop computer is carried and moved around, while the desktop computer is streaming a video clip to the laptop computer.

The captured data in the network analyzer software is exported to an Excel file.
A MATLAB program is used to analyze a series of round-trip-time (RTT) values obtained experimentally from this wireless connection. The series of RTT are then sent to median filters of different sizes for filtering. The median filter window sizes used for testing are three, five, seven and nine.

5.3 Analysis of Experimental Results

The first part of the experimental results is plotted in Figure 5.2 to Figure 5.6.

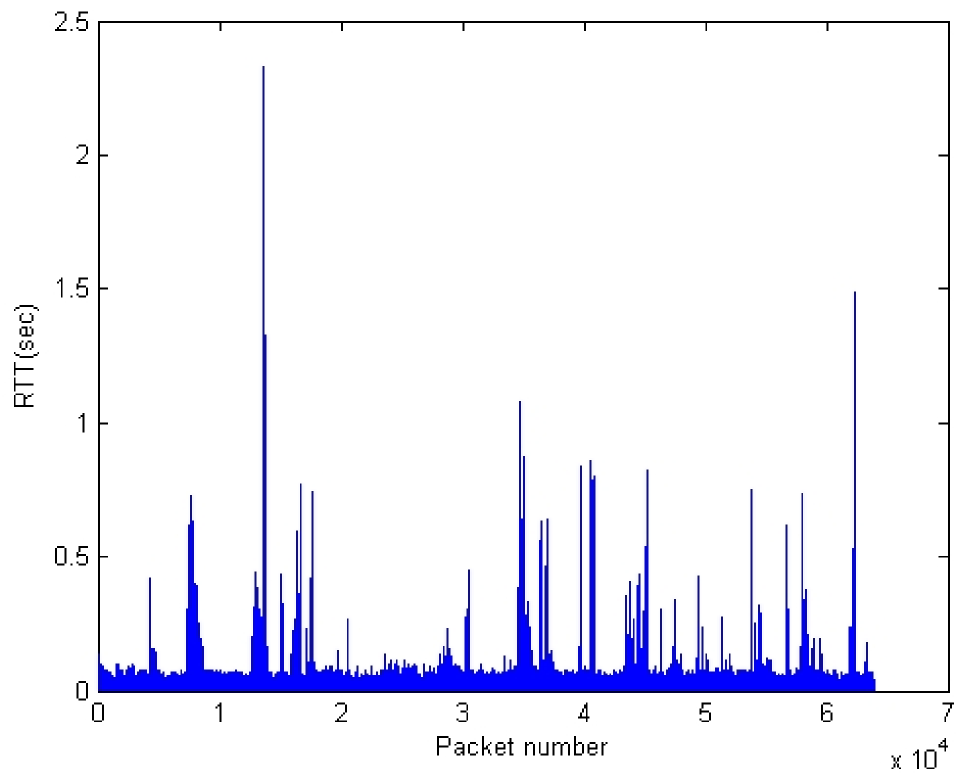

Figure 5.2: Unfiltered RTT from a wireless connection. Note that one of the RTT timing is as long as 2.4 seconds.

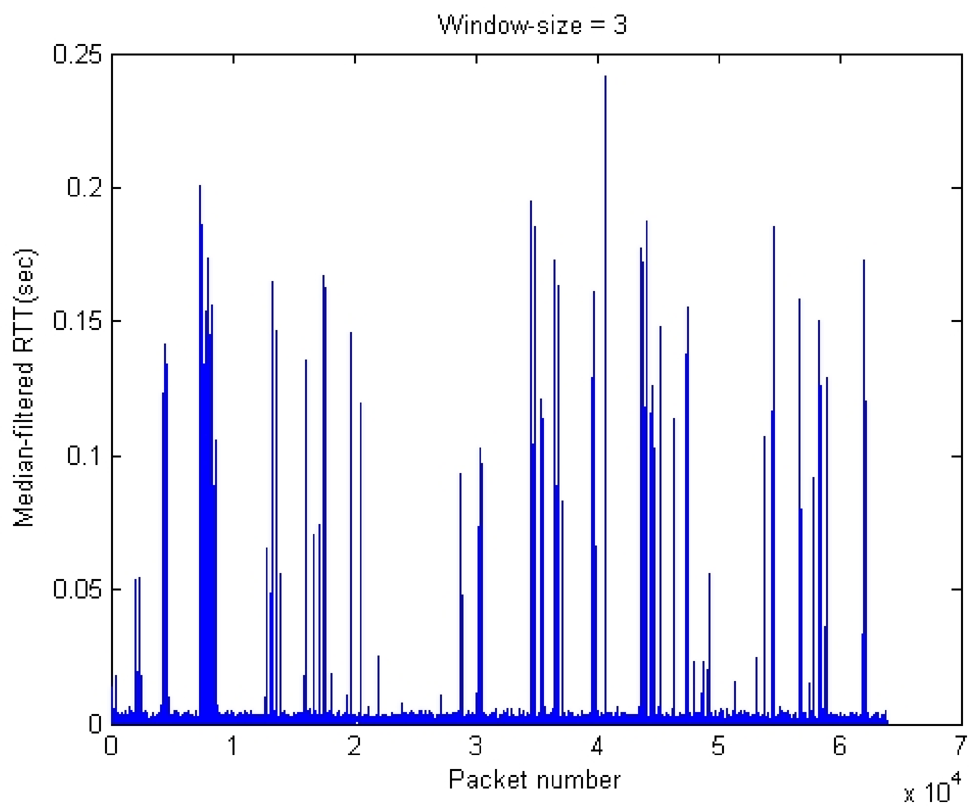

Figure 5.3: Median-filtered RTT with window size of three. The fluctuation in amplitude of RTT is reduced.

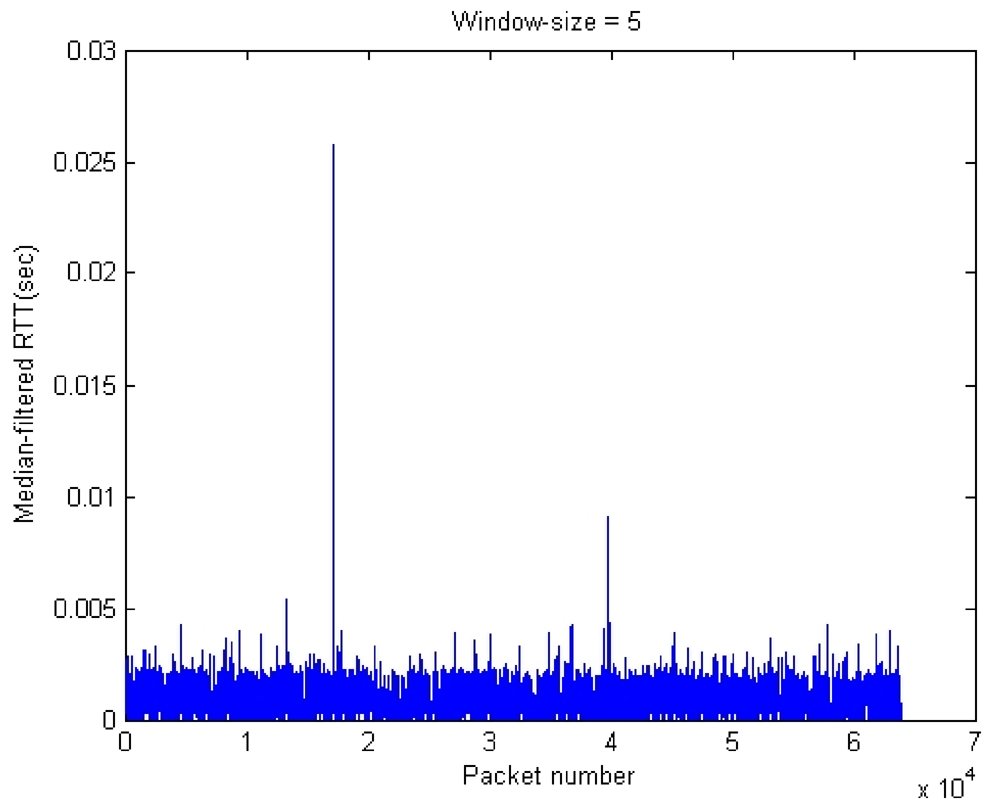

Figure 5.4: Median-filtered RTT with window size of five. Only two large RTT's are not filtered out.

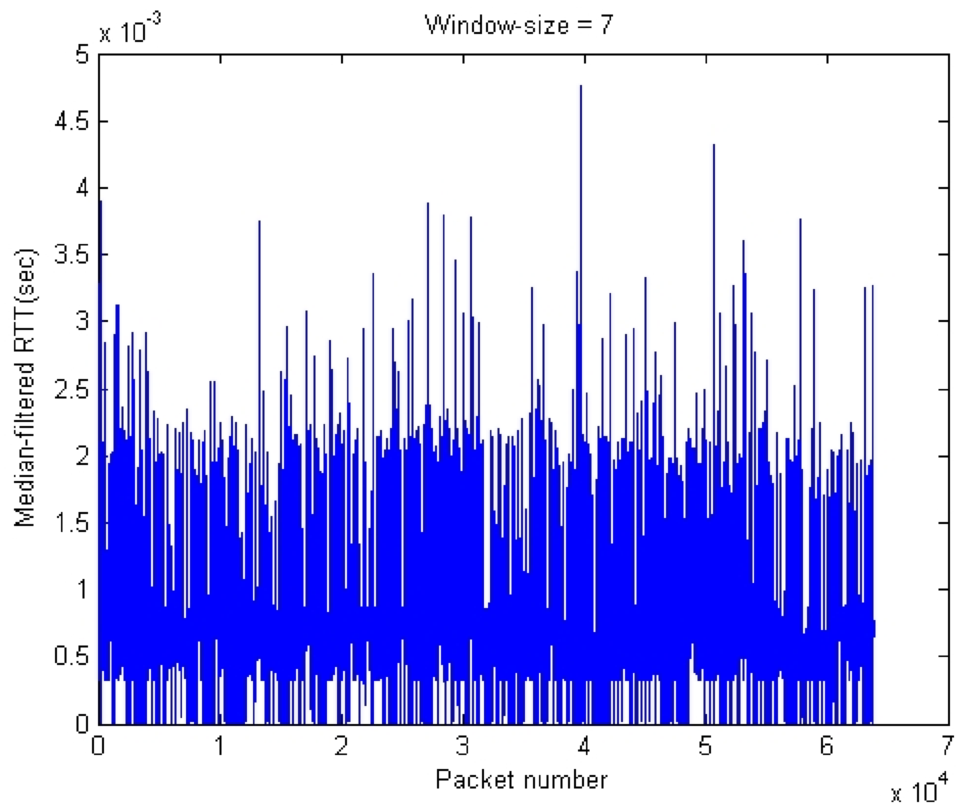

Figure 5.5: Median-filtered RTT with window size of seven. There is no significant improvement when the window size is larger than five.

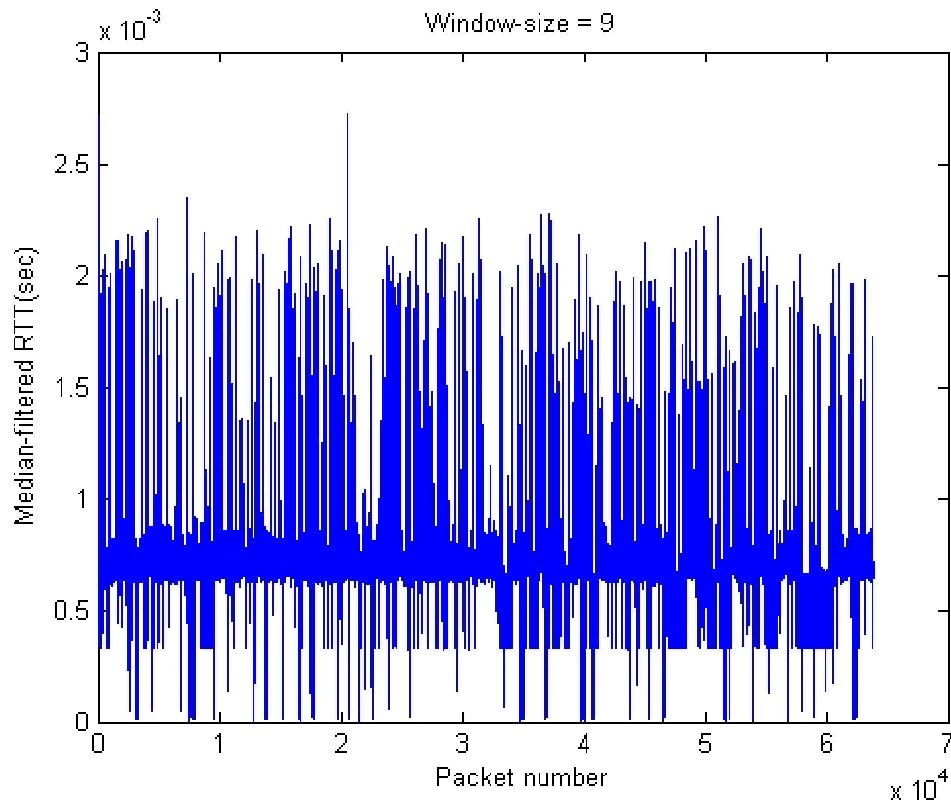

Figure 5.6: Median-filtered RTT with window size of nine. There is no significant improvement when the window size is larger than five.

Figure 5.2 shows the unfiltered RTT's from this wireless connection.

From Figure 5.3 to Figure 5.6, when the median filter window size increases, those RTT values that are greatly different from the rest, are filtered away. Also, the RTT values are getting smaller as the median filter window size increases.

From this observation, the conclusion is that median filter can improve RTT, which in turn improves the congestion window size management. With consistently stable RTT values, a wireless connection can have fairly constant transfer rate.

If suddenly there is a large sample RTT, which is larger than the estimated RTO, timeout will occur. The occurrence of timeout will cause slow start; which will lower the throughput. If the RTT is consistently stable, the chances for the sample RTT to be larger than the estimated RTO will be low, and therefore there will be fewer timeout.

5.4 Comparison of Different Round Trip Times

In 1988, Jacobson and Karels proposed their Jacobson/Karels Algorithm (Comer 2006*a*, Peterson & Davie 2000, Jacobson 1988), which is described by equations (5.1), (5.2) and (5.3):

$$SRTT = \alpha SRTT + (1 - \alpha)M \tag{5.1}$$

where SRTT (on the left) is the new Estimated RTT;
SRTT (on the right) is the old Estimated RTT;
M is the sample RTT;
α is a constant. In Jacobson/Karels Algorithm,
$\alpha = 0.875 = 7/8$, but in this practical experiment,
$\alpha = 0.9$ is chosen for simplicity.

Equation 5.1 is a weighted-average filter.

A MATLAB program is written to obtain a series of round trip time (RTT) by using equation (5.1).

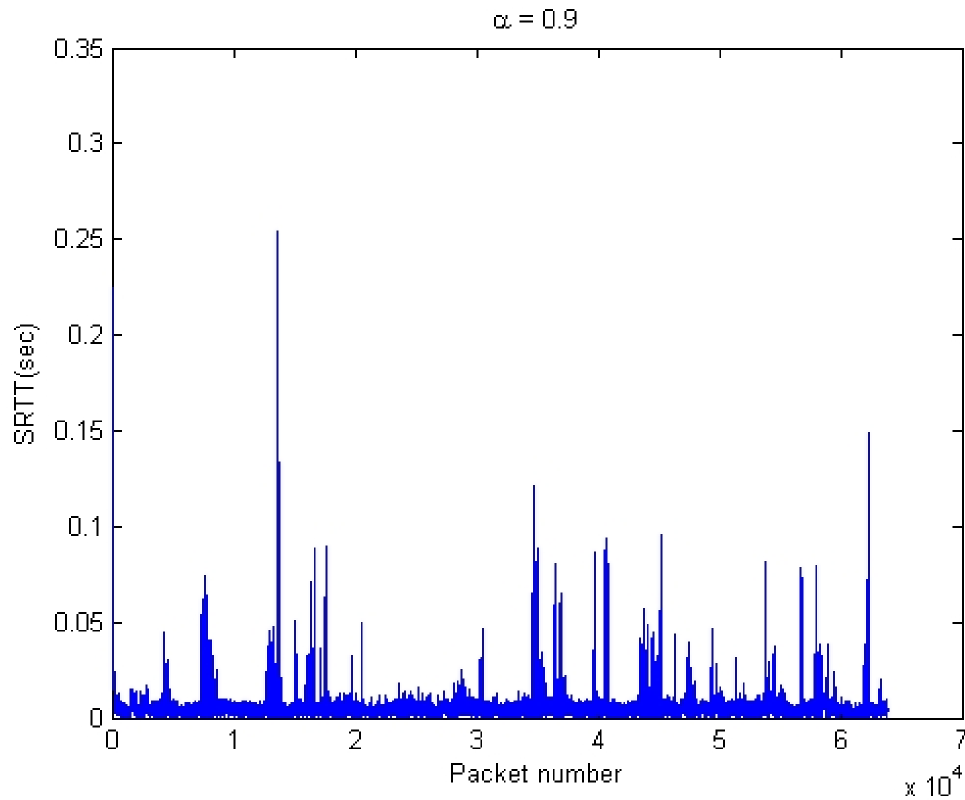

Figure 5.7: Smoothed RTT by using equation (5.1). The pattern in Figure 5.7 is the same as that in Figure 5.2, except that the amplitude in Figure 5.7 is reduced.

If Figure 5.6 is compared with Figure 5.7, the set of RTT estimated by median filter of window size nine, has less fluctuation than that estimated by weighted-average filter.

5.5 Comparison of Different Retransmission Timeouts

A further investigation on the performance of median filter and weighted-average filter, with regard to RTO, is carried out.

For simplicity, the unfiltered RTO is taken to be 1.1 times of the unfiltered RTT.

The median-filtered RTO is also taken to be 1.1 times of the median-filtered RTT.

For the weighted-average filtered RTO, Jacobson/Karels Algorithm is expressed as equations (5.2) and (5.3).

$$D = \delta D + (1 - \delta)|SRTT - M| \tag{5.2}$$

where D (on the left) is the new deviation;

D (on the right) is the old deviation;

δ is a constant, which is equal to $0.75 = 3/4$.

$$SRTO = SRTT + 4D \tag{5.3}$$

where SRTO is the next retransmission timeout;

SRTT is the new Estimated RTT;

D is the new deviation.

Only one set of samples from the unfiltered RTT is chosen for RTO comparison. That is, only sample numbers 13655 to 13665 are chosen for testing. This set of samples represents a situation where the transmission rate is bursty. Results for sample numbers 13655 to 13665 are shown in Figure 5.8 to Figure 5.13.

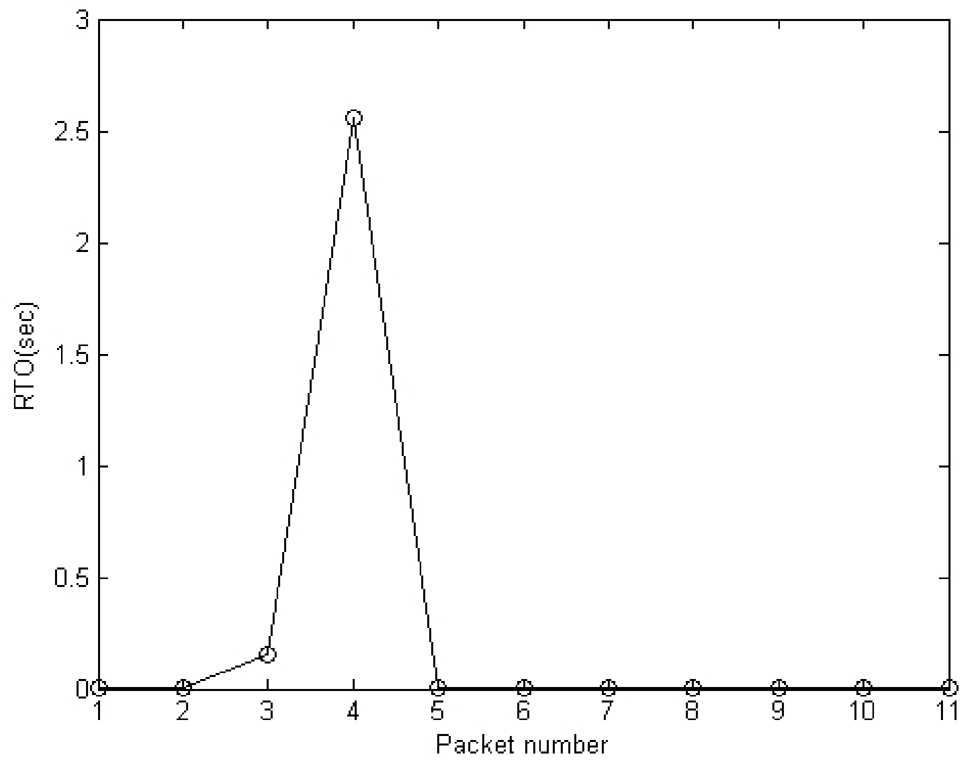

Figure 5.8: Unfiltered RTO from a wireless connection. Figure 5.8 will be compared with other RTO's in Figure 5.11.

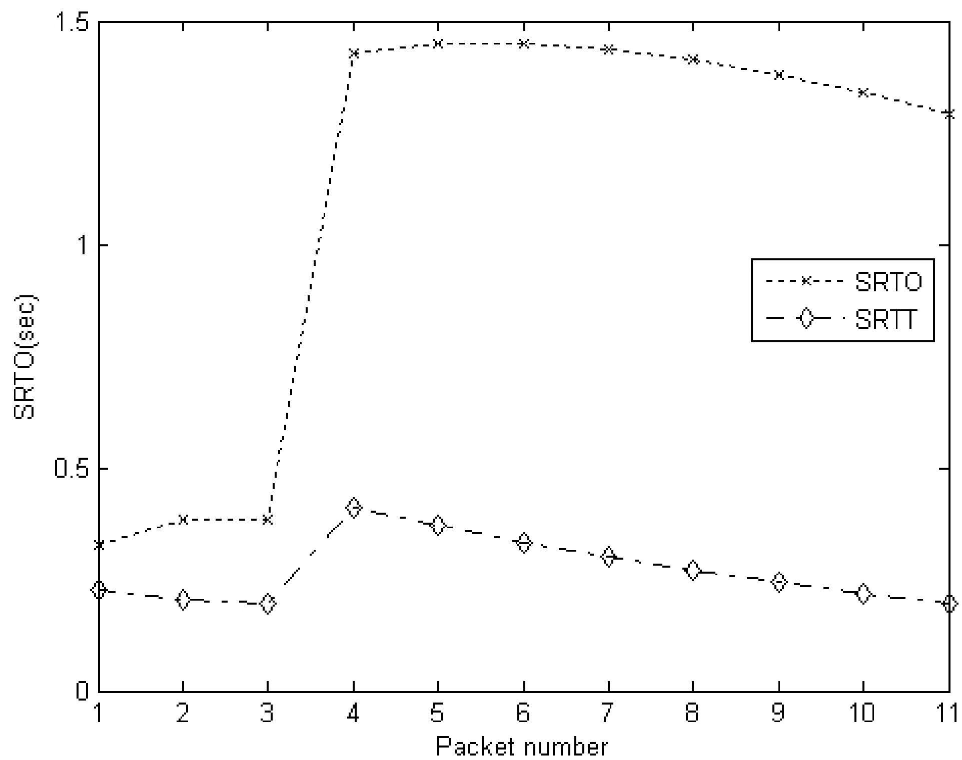

Figure 5.9: Smoothed RTO using (5.2) and (5.3). Figure 5.9 will be compared with other RTO's in Figure 5.11.

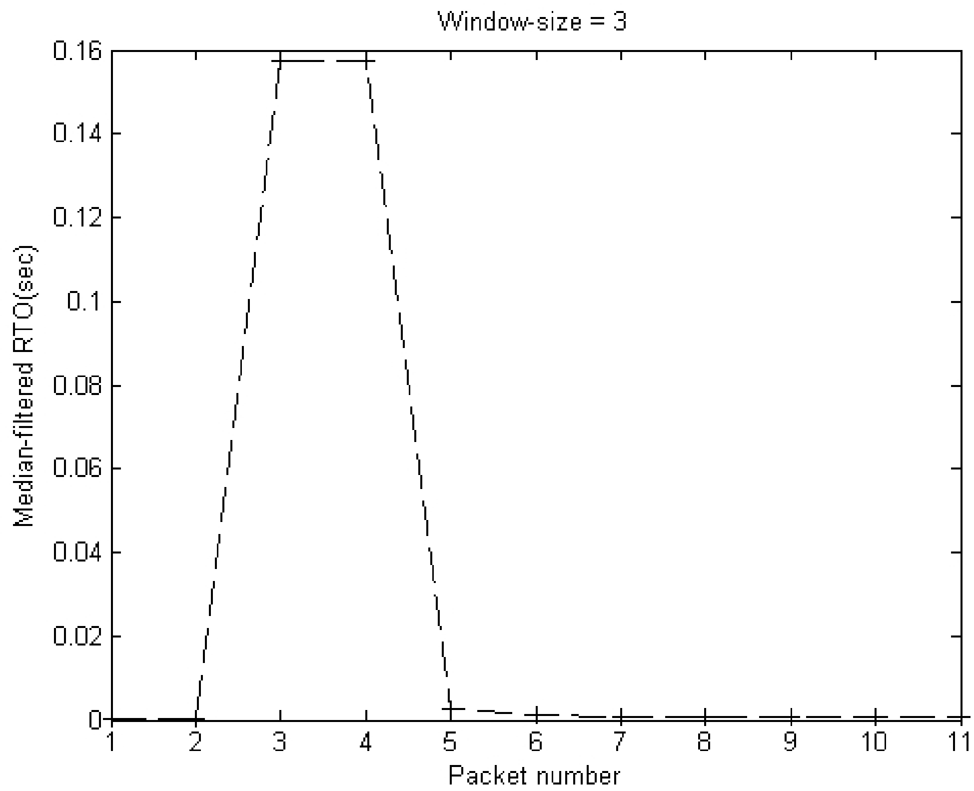

Figure 5.10: Median-filtered RTO with window size of three. Figure 5.10 will be compared with other RTO's in Figure 5.11.

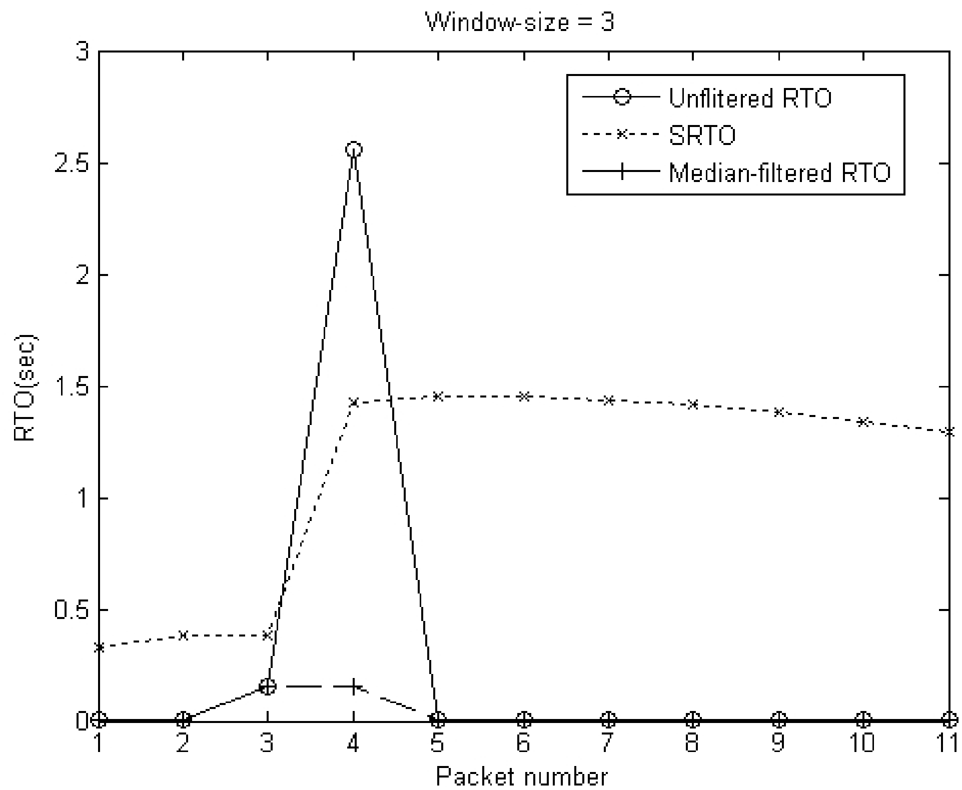

Figure 5.11: Different RTOs. Median-filtered RTO has amplitude less than those of SRTO and unfiltered RTO.

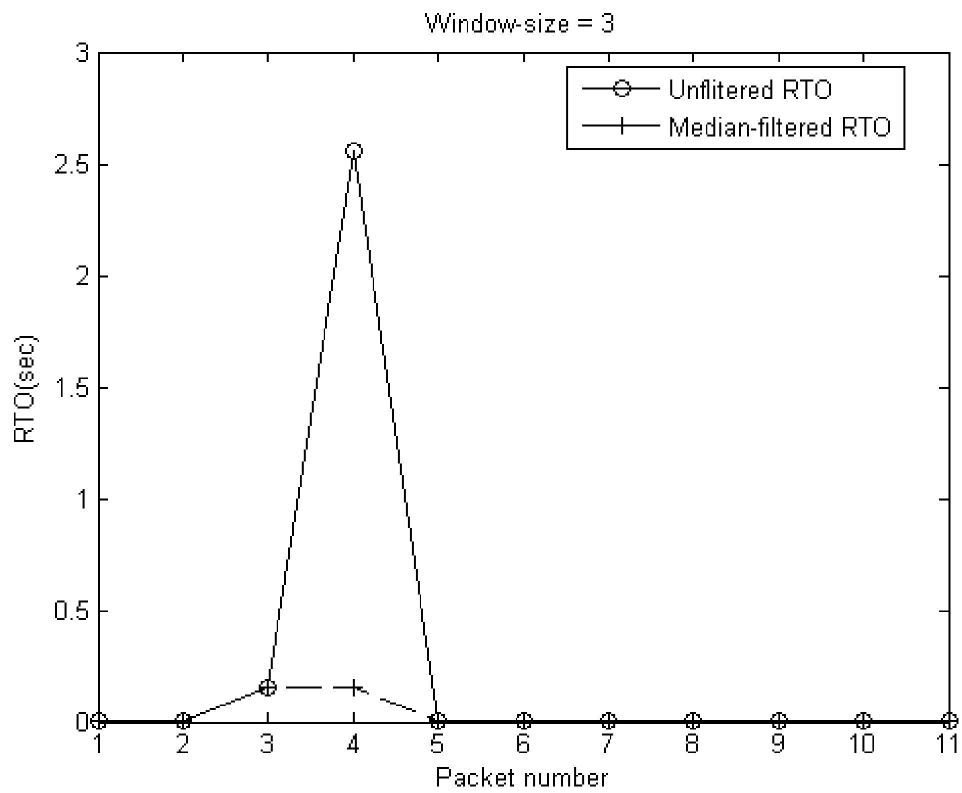

Figure 5.12: Unfiltered RTO and median-filtered RTO. Median-filtered RTO has amplitude less than that of unfiltered RTO.

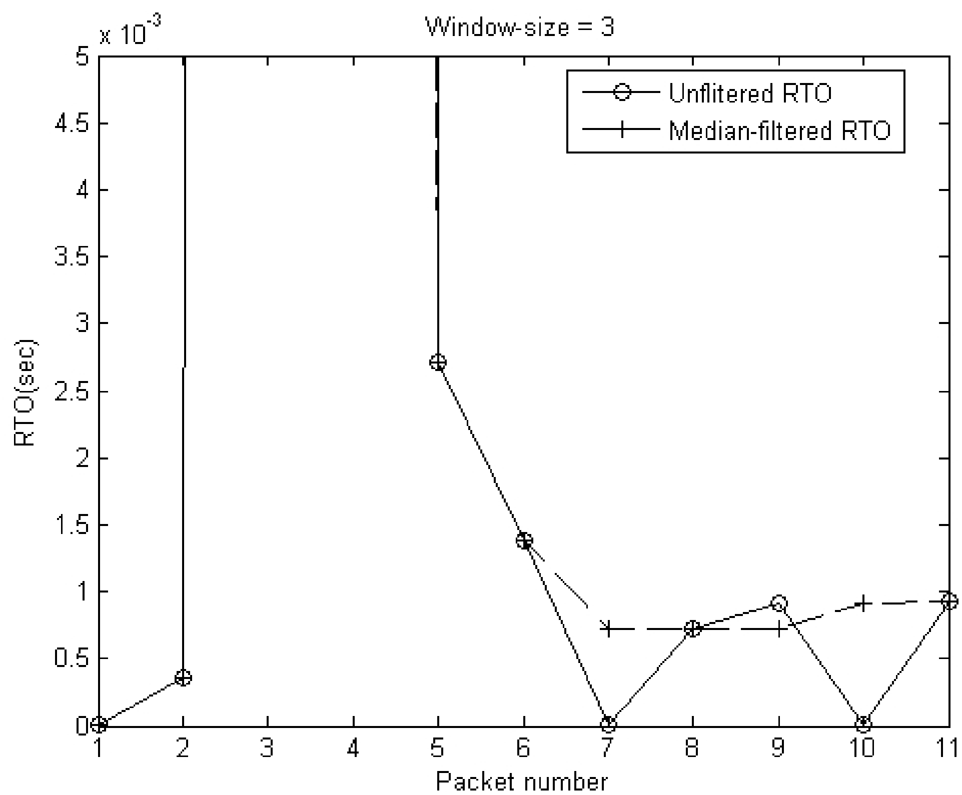

Figure 5.13: Unfiltered RTO and median-filtered RTO (Zoom-in view). Figure 5.13 is the enlargement of Figure 5.12, for the portion near time equal to zero.

Figure 5.8 shows the unfiltered RTO, which is 1.1 times of the unfiltered RTT. The retransmission timeout (i.e. smoothed RTO, or SRTO) in Figure 5.9 is obtained by using equations (5.2) and (5.3). Median-filtered RTO with window size equal to three is shown in Figure 5.10. From Figure 5.11, RTO's generated by weighted-average filter (i.e. SRTO) are much higher than those generated by median filter.

Figure 5.13 is the enlargement (i.e. zoom-in view) of Figure 5.12, for the portion near time equal to zero.

5.6 Histograms of Round Trip Times

The histograms of different RTT values are shown from Figure 5.14 to Figure 5.21.

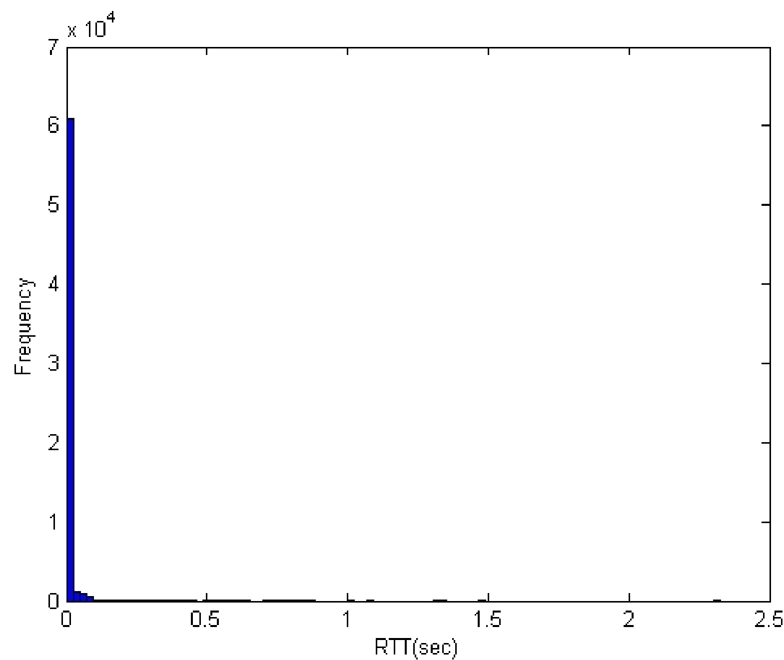

Figure 5.14: Histogram of unfiltered RTT from a wireless connection. There are RTT's of long duration lying on the right of the mode.

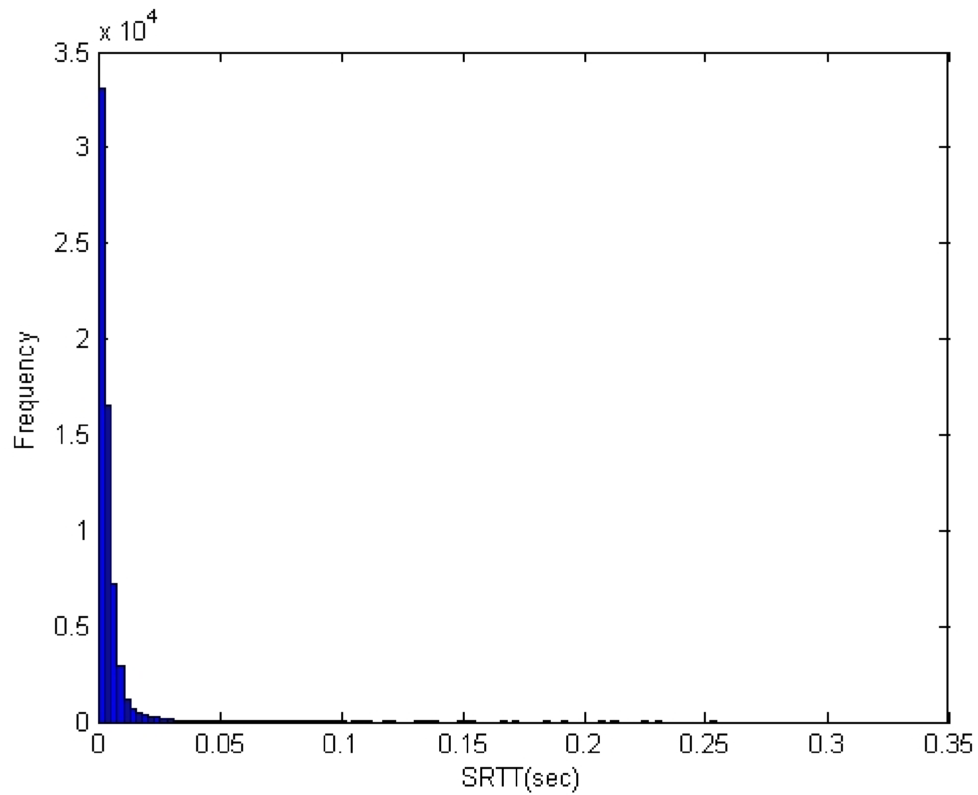

Figure 5.15: Histogram of smoothed RTT by using equation (5.1). The RTT's are distributed in a similar pattern as that in Figure 5.14, except that the duration is shorter.

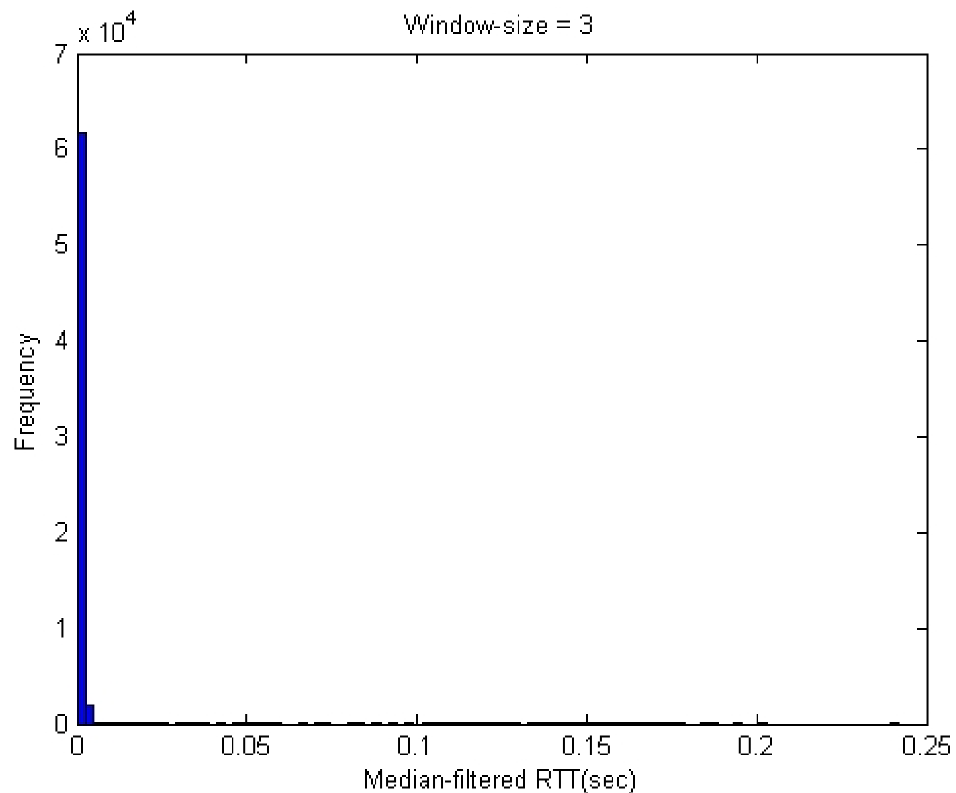

Figure 5.16: Histogram of median-filtered RTT with window size of three. For window size of 3, the RTT's are still distributed on the right side of the mode. It will be seen in later figures that the distribution pattern will change.

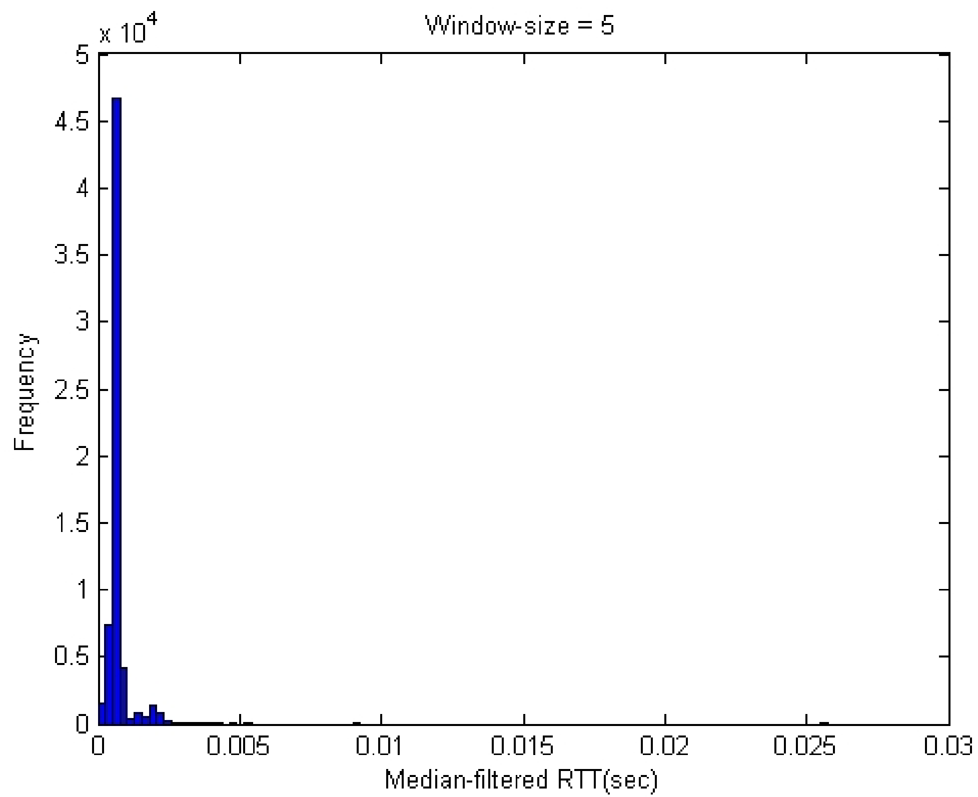

Figure 5.17: Histogram of median-filtered RTT with window size of five. Some RTT's begin to exist on the left side of the mode.

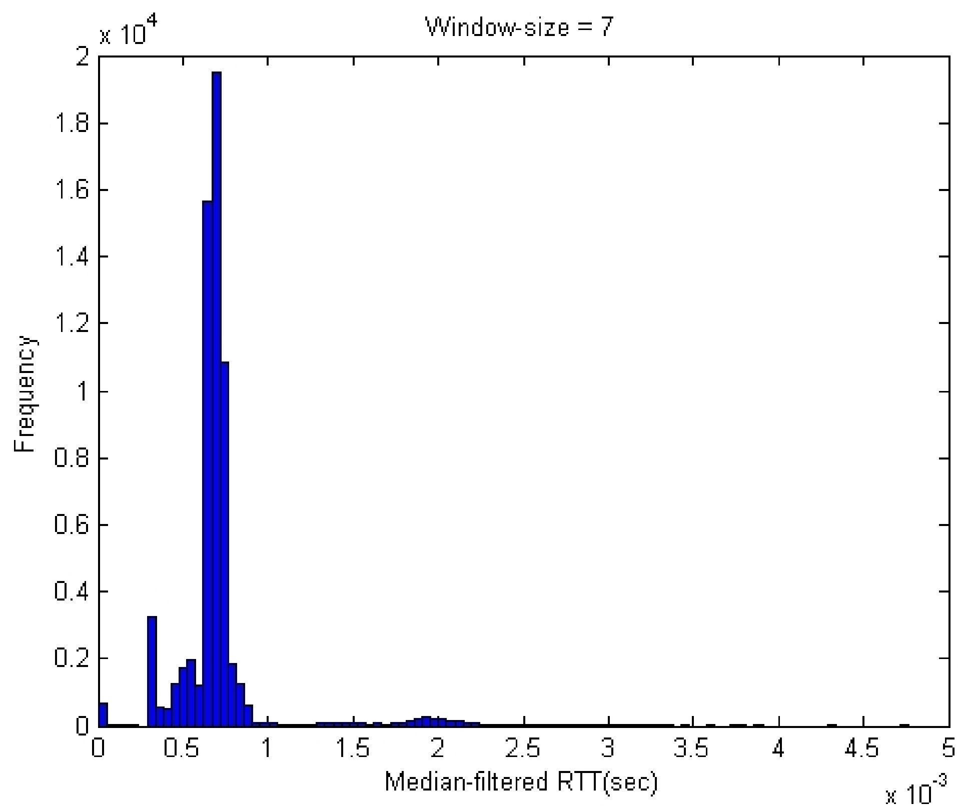

Figure 5.18: Histogram of median-filtered RTT with window size of seven. The distribution of RTT's approaches a normal distribution.

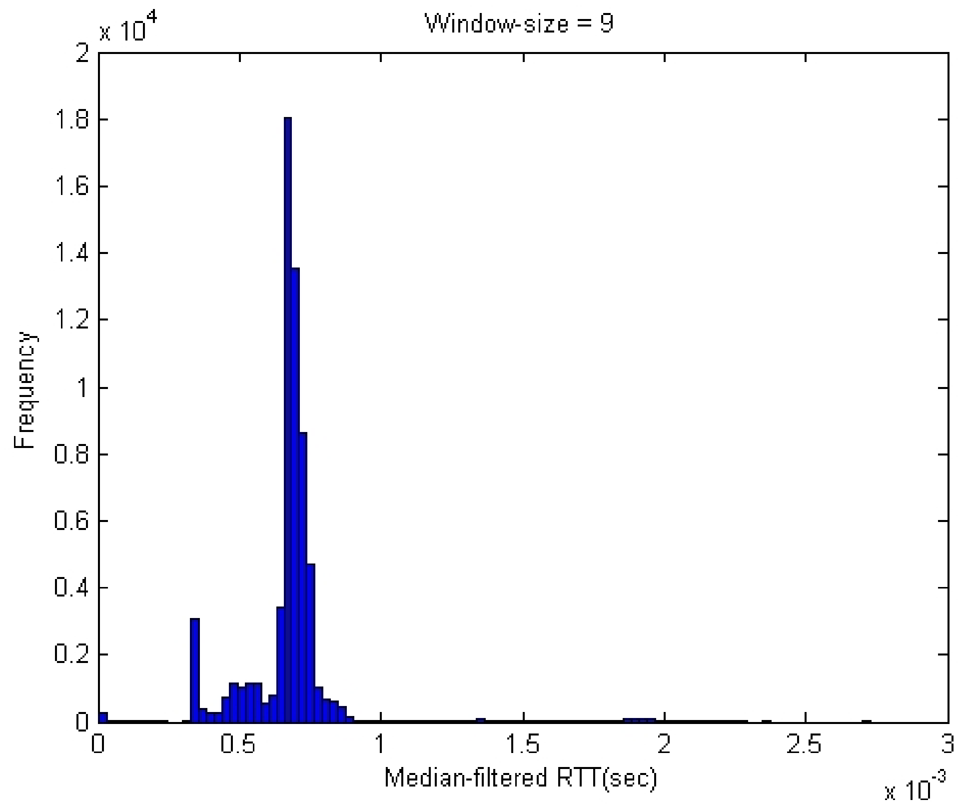

Figure 5.19: Histogram of median-filtered RTT with window size of nine. There are no significant changes after window size of seven.

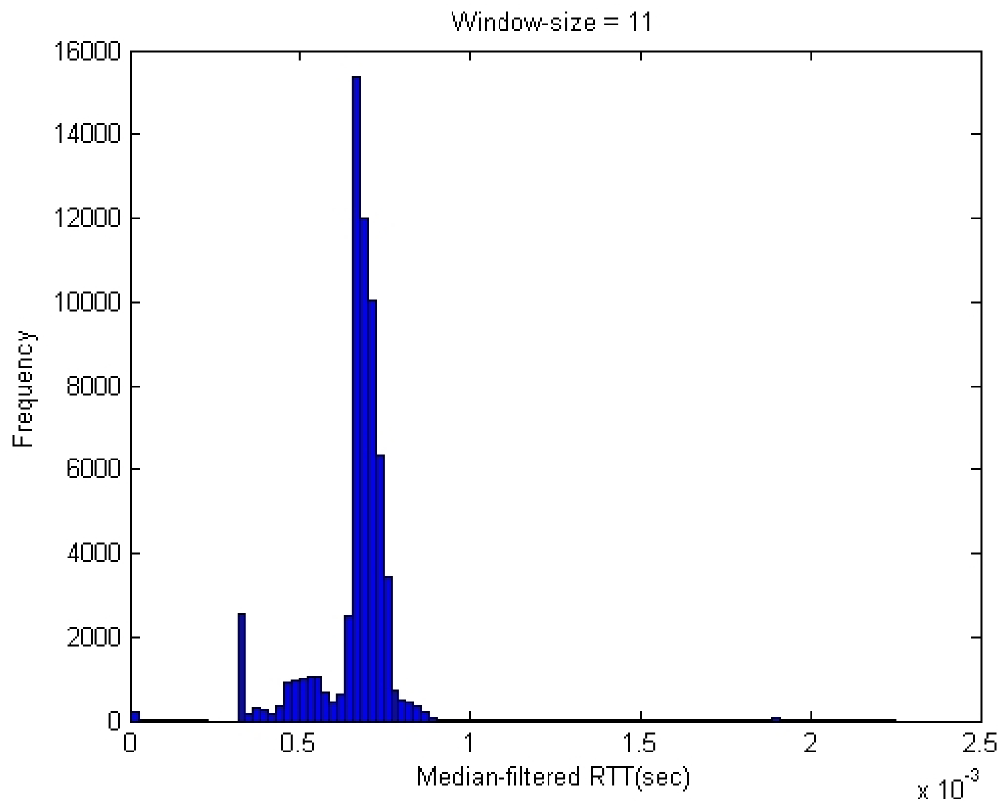

Figure 5.20: Histogram of median-filtered RTT with window size of eleven. There are no significant changes after window size of seven.

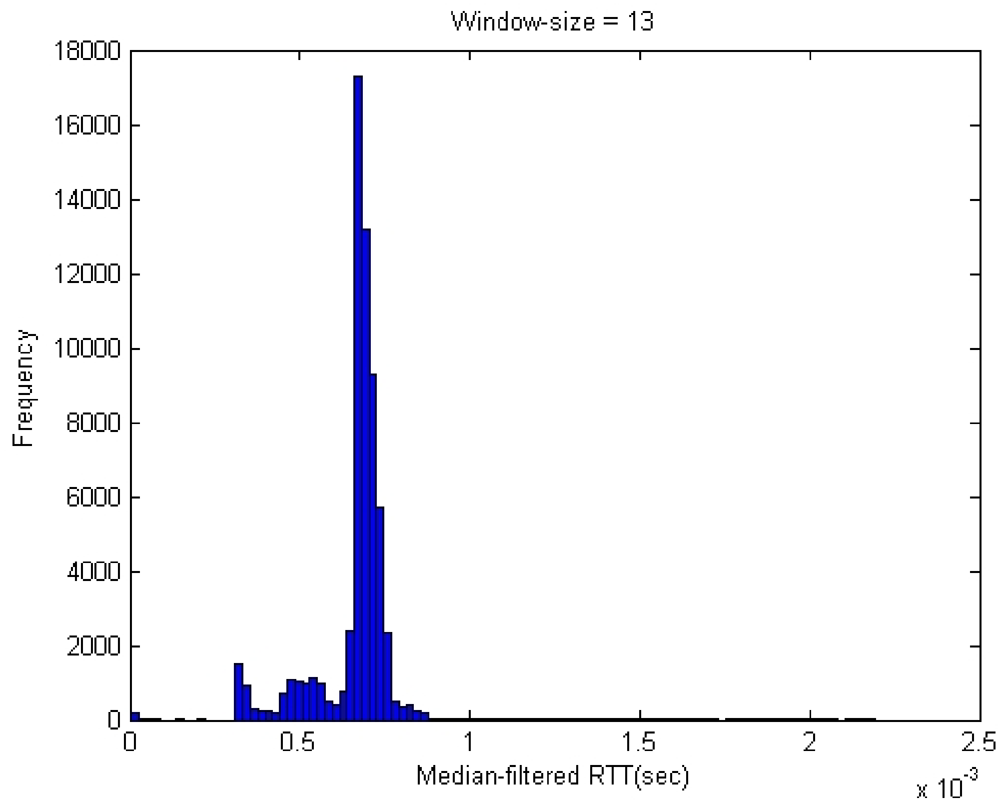

Figure 5.21: Histogram of median-filtered RTT with window size of thirteen. There are no significant changes after window size of seven.

Figure 5.14 is the histogram of unfiltered RTT. That is, Figure 5.14 is the histogram of the data shown in Figure 5.2. From Figure 5.14, unfiltered RTT does not follow a Gaussian distribution, because large RTT values exist on the right side of Figure 5.14. In addition, these large RTT values are not in great amount.

Figure 5.15 shows the histogram of smoothed RTT, smoothed by using equation (5.1). That is, Figure 5.15 is the histogram of data shown in Figure 5.7. If Figure 5.15 is compared with Figure 5.14, the histogram in Figure 5.15 shows a gradual change in RTT values. However, in Figure 5.15, the RTT values are still distributed to the right side of the mode. (The mode is the value that occurs most often, i.e. the value with highest frequency).

Figure 5.16 to Figure 5.19 are histograms of data shown in Figure 5.3 to Figure 5.6, respectively.

Figure 5.20 shows histogram of median-filtered RTT with window size of eleven.

Figure 5.21 shows histogram of median-filtered RTT with window size of thirteen.

Figure 5.16 to Figure 5.21, show histograms obtained with window sizes equal to 3, 5, 7, 9, 11, 13, respectively. From Figure 5.16 to Figure 5.21, as the window size increases, the RTT values are distributed to the left side and right side of the mode.

5.7 Mode and Outliers

The mode is defined as the RTT value that occurs most frequently in that set of filtered RTT values. The outlier is defined as the RTT value that is within ±10% of the value of the mode.

Window Size	**Mode (in millisecond)**	**Outlier = Mode ±10%**
3	1.2	99.5%
5	0.64	51.3%
7	0.69	28.7%
9	0.67	30.8%
11	0.67	31.2%
13	0.67	26.5%

Table 5.1: Effect on different window sizes

Table 5.1 shows the modes and outliers of different window sizes. From Table 5.1, as window size increases, the percentage of outliers decreases. Also, the mode has no significant change after window size of five.

If the Round Trip Times have a regular pattern, the RTT is called consistent RTT. From these observations, the conclusion is that the median filter can perform better than the weighted-average filter because consistent RTT and small RTO are obtained, which are desirable factors for high connection throughput.

5.8 Computation and Implementation Consideration

To find the median of an array, it has to be sorted first. The role of sorting algorithm is to support median filtering.

To determine the speed of bubble sort, it is necessary to code it into machine language of the platform on which it will be executed.

Suppose the bubble sort is coded into a set of instructions in machine language. These instructions can be divided into two groups. The first group of instructions will instruct the CPU (Central Processing Unit) to perform mathematical and logical operations, or to transfer data among its registers. The second group of instructions will instruct the CPU to access data in cache memory, RAM (Random Access Memory), hard disk or floppy disk. Data are fetched faster from cache memory than from RAM. Fetching data from disk is the slowest.

The first group of instructions (i.e. to perform mathematical operations) is much faster than the second group of instructions (i.e. to access data in memory). The speed of performing mathematical operations can be faster than the speed of accessing data in memory by 100 times (McAllister 2009).

5.8.1 Example on Memory Access

The following example adds up the elements of a two-dimensional matrix, consisting of n rows and n columns.

```
1. row_index=0; total_sum=0;
2. while(row_index<n)
3. {  column_index=0;
4.    while(column_index<n)
5.    {   total_sum=total_sum+data(row_index,column_index);
6.        column_index=column_index+1;
7.    }
8.    row_index=row_index+1;
9. }
```

Modern compliers recognize variables that are accessed repeatedly inside loops, and assign them to CPU registers. That is, the variables, row_index and column_index are assigned to CPU registers. Therefore, Lines 1, 2, 3, 4, 6 and 8 are not memory access instructions. Line 5 will access every element of the matrix sequentially, so it is a memory access instruction.

Moreover, Line 1 will execute once; Lines 2, 3 and 8 will execute n times; and Lines 4, 5 and 6 will execute n $\times$ n times.

The actual execution time of an algorithm will depend on:

- the platform (whether Windows or Linux is used)
- the high-level language (whether Java or MatLAB is used)
- the architecture of CPU (whether the CPU can perform parallel operations)
- the complier (whether the complier can assign variables to registers correctly)

Following the example in Section 5.8.1, the sorting algorithms will be analyzed and presented in Chapter 6.

5.9 Summary

From the experimental results, it is found that median filter can perform better than the weighted-average filter because consistent RTT and small RTO are obtained, which are desirable factors for high connection throughput.

As the bit rate of wireless networks is increasing, the time required for the median computation becomes critical. From the previous sections, it is shown that a large window size is desirable. However, this also increases the amount of time in finding the median. The fact that the median algorithm works well in the above practical experiment, motivates us to develop a fast one-dimensional (1D) sorting algorithm, which can be used for real-time median filtering. This fast 1D sorting algorithm will be presented in the next chapter.

Chapter 6

A Fast Sorting Algorithm for Median Filtering

6.1 Introduction

One of the objectives of this research is to apply median filter to enhance the performance of Transmission Control Protocol (TCP). Median filtering involves sorting to find the middle element, so a literature review on sorting algorithms is necessary. Different algorithms of sorting are described in this chapter (Clark & et al. 2002).

This chapter begins with a description on Big-O notation, followed by Shellsort and Quicksort.

In 1959, Donald L. Shell presented his Shellsort in his paper "A High-Speed Sorting Procedure" (Shell 1959). As Shellsort is a high-speed sorting procedure, a literature review of it is presented. In 1962, C.A.R. Hoare published his "Quicksort" in Computer Journal (Hoare 1962). A literature review of it will be presented here. In 1979, T.S. Huang *et al.* published their paper "A Fast Two-Dimensional Median Filtering Algorithm" (Huang & et al. 1979). It is modified to one-dimensional, so it is necessary to present a literature review on it.

6.2 Comparison of three sorting algorithms

This section presents a theoretical comparison of the execution speeds of bubble sort, Shellsort and Quicksort.

6.2.1 The Big-O Notation

The Big-O analysis, where O stands for the order of magnitude, is an analysis technique used to approximate the value of a function. As an example, let us consider the function $y = n + n\log n + 2n^2$, where log is in base 2. Table 6.1 presents the values of this function and its three terms, for values of n equal to 10^5, 10^6, 10^7, 10^8.

	n	$n\log n$	$2n^2$	y
$n = 10^5$	10^5	1.66×10^6	2×10^{10}	2.00×10^{10}
$n = 10^6$	10^6	1.99×10^7	2×10^{12}	2.00×10^{12}
$n = 10^7$	10^7	2.33×10^8	2×10^{14}	2.00×10^{14}
$n = 10^8$	10^8	2.66×10^9	2×10^{16}	2.00×10^{16}

Table 6.1: Contributions of n, $n\log n$ and $2n^2$ to the function y. It can be seen that $2n^2$ contributes most to y.

The term n^2 dominates the other terms in the function $y = n + n\log n + 2n^2$, as n gets large. To find the approximate value of y, the dominant term $2n^2$ is evaluated.

An extension of this concept of dominance of one term in a function is that, the constant multiplying the dominant term is neglected. For example, the Big-O notation for the function $y = n + n\log n + 2n^2$ is $O(n^2)$.

6.2.2 Application of Big-O Notation

The Big-O notation is used in software engineering to evaluate the execution speed of an algorithm.

For bubble sort, the Big-O notation is $O(n^2)$ (Liang 2007).

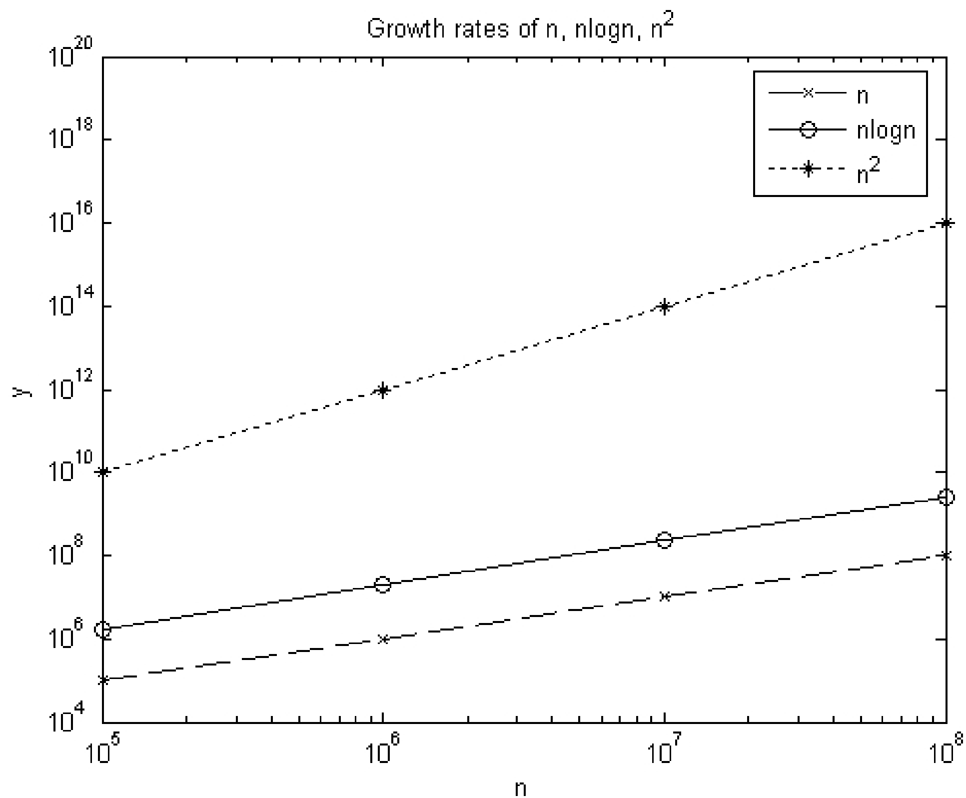

Figure 6.1: Comparison of growth n, $n\log n$ and n^2. It can be seen that the growth rate of n^2 is the largest.

For Shellsort, the Big-O notation is O($n^{1.25}$) (Weiss 2002, Kruse 1984).
For Quicksort, the Big-O notation is O($n\log n$) (Liang 2007, Weiss 2002, Main & Savitch 2005).

Based on the above discussion, it is found that the relative execution speeds of the three sorting algorithms, from slowest to fastest is: bubble sort (the slowest), Shellsort, and Quicksort (the fastest).

6.3 Shellsort

Shellsort is named after its designer Donald L. Shell. This algorithm is easiest to describe, by considering sorting an array of 16 elements, Array A=[a_1, a_2,...a_{15}, a_{16}].
In the first sort, Array A is divided into 8 groups, which are [a_1,a_9];[a_2,a_{10}];...[a_7,a_{15}];[a_8,a_{16}]. Each of the 8 groups is sorted in ascending order (Press & et al. 1989).

In the second sort, Array A is divided into 4 groups, which are [a_1, a_5, a_9, a_{13}];...[a_4, a_8, a_{12}, a_{16}]. Each of the 4 groups are sorted in ascending order (Press & et al. 1989).
In the third sort, Array A is divided into 2 groups, which are [a_1, a_3, a_5, a_7, a_9, a_{11}, a_{13}, a_{15}] and [a_2, a_4, a_6, a_8, a_{10}, a_{12}, a_{14}, a_{16}].
These 2 groups are sorted in ascending order (Press & et al. 1989).
In the final fourth sort, the whole Array A is sorted in ascending order (Press & et al. 1989).
The following example illustrates the above steps (McConnell 2001).

(a) First sort

A= [16, 7, 10, 1, 13, 11, 3, 8, 14, 4, 2, 12, 6, 5, 9, 15]

$[a_1, a_9] = [16, 14]$; $[a_2, a_{10}] = [7, 4] \ldots [a_7, a_{15}] = [3, 9]$; $[a_8, a_{16}] = [8, 15]$

After first sort,

$[a_1, a_9] = [14, 16]$; $[a_2, a_{10}] = [4, 7] \ldots [a_7, a_{15}] = [3, 9]$; $[a_8, a_{16}] = [8, 15]$

A = [14, 4, 2, 1, 5, 11, 3, 8, 16, 7, 10, 12, 6, 13, 9, 15]

(b) Second sort

$[a_1, a_5, a_9, a_{13}] = [14, 5, 16, 6]$; $[a_2, a_6, a_{10}, a_{14}] = [4, 11, 7, 13]$

$[a_4, a_8, a_{12}, a_{16}] = [1, 8, 12, 15]$; $[a_3, a_7, a_{11}, a_{15}] = [2, 3, 10, 9]$

After second sort,

$[a_1, a_5, a_9, a_{13}] = [5, 6, 14, 16]$; $[a_2, a_6, a_{10}, a_{14}] = [4, 7, 11, 13]$

$[a_4, a_8, a_{12}, a_{16}] = [1, 8, 12, 15]$; $[a_3, a_7, a_{11}, a_{15}] = [2, 3, 9, 10]$

A = [5, 4, 2, 1, 6, 7, 3, 8, 14, 11, 9, 12, 16, 13, 10, 15]

(c) Third sort

$[a_1, a_3, a_5, a_7, a_9, a_{11}, a_{13}, a_{15}] = [5, 2, 6, 3, 14, 9, 16, 10]$

After third sort,

$[a_1, a_3, a_5, a_7, a_9, a_{11}, a_{13}, a_{15}] = [2, 3, 5, 6, 9, 10, 14, 16]$

A = [2, 1, 3, 4, 5, 7, 6, 8, 9, 11, 10, 12, 14, 13, 16, 15]

(d) Fourth sort

After fourth sort,

A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

Shellsort is an efficient version of insertion sort (Oliver 1993). In insertion sort, the first element of an array is considered as a sorted list of size 1. A sorted list of size 2 is obtained, by inserting the second element into the correct location of the sorted list of

size 1. The third element is inserted into the correct location of the two-element sorted list, and so on (McConnell 2001).

In insertion sort many elements must be moved when a new element is inserted. Shellsort improves this situation by first sorting elements far apart, for example, $[a_1, a_9]$, so that elements far from their final locations are first moved closer to their final locations (Oliver 1993). In each subsequent sorting, the distances between elements are reduced by half. This is because of the fact that, in reducing the numbers of groups, from 8 groups to 4 groups to 2 groups, the distances between elements will also be reduced. In the fourth sort, the distance is equal to one, and Shellsort becomes insertion sort.

6.4 Quicksort

Quicksort was first introduced by C.A.R. Hoare (Hoare 1962) in 1962.
Quicksort chooses an element from the array, and uses it as a reference to divide the array into two sub-arrays. The left sub-array contains those elements that are smaller than the reference. The right sub-array contains those elements that are larger than or equal to the reference. The Quicksort algorithm is applied to both sub-arrays, recursively (McConnell 2001).
The reference is called the pivot. The array is rearranged by moving elements that are smaller than the pivot, to the left; elements that are larger than or equal to the pivot, to the right (McConnell 2001). This process is called partitioning (Oliver 1993).
Let the index of pivot be i. Then elements between index 1 and index (i-1), are smaller than the pivot. Elements between index (i+1) and index n are larger than or equal to the pivot. Quicksort is called recursively on the two sub-arrays. If Quicksort is called to a sub-array with one element, it does nothing.

Example

Apply Quicksort to sort the array

A = [4, 2, 3, 1, 6, 5, 7] into ascending order.

Solution

Use the first element 4 as pivot.

Move elements 2, 3, 1, which are smaller than 4, to the left; move elements 6, 5, 7, which are larger than 4, to the right.

[2, 3, 1] + [4] + [6, 5, 7]

The first element 4, by coincidence, is also the median. It partitions A into two sub-arrays of equal length. Later, in section 6.4.3, median will be shown to be the best choice for pivot.

Apply Quicksort to sub-array [2, 3, 1].

Use the first element 2 as the pivot.

Move 1 to the left; move 3 to the right.

[1] + [2] + [3] + [4] + [6, 5, 7]

Apply Quicksort to sub-array [6, 5, 7]

Use the first element 6 as pivot.

Move 5 to the left; move 7 to the right.

[1] + [2] + [3] + [4] + [5] + [6] + [7]

The sorted array in ascending order is: [1, 2, 3, 4, 5, 6, 7]

6.4.1 Quicksort Algorithm

The Quicksort algorithm is as follows (McConnell 2001, Berman & Paul 2005):

```
Quicksort (list, first, last)
list are the elements to be rearranged.
first is the index of the first element.
last is the index of the last element.

if  first < last  then
  pivot = Partition (list, first, last)
  Quicksort (list, first, pivot-1)
  Quicksort (list, pivot+1, last)
end if
/*Call Partition function to rearrange the array.
  Apply Quicksort recursively to the left, that is, from index=first
```

```
to index=pivot-1.
Apply Quicksort recursively to the right, that is, from index=pivot+1
to index=last*/
```

The Partition function designed by C.A.R. Hoare in 1962, rearranges the array from left and from right simultaneously (Berman & Paul 2005), which is faster. This will, of course, be more complicated to program.

6.4.2 Partition Function

The Partition function presented below is a simplified version, which rearranges the array from left to right. This simplified version is easier to understand and easier to program (McConnell 2001).

```
Partition (list, first, last)
          list are the elements to be rearranged
          first is the index of the first element
          last is the index of the last element
Line 1:   PivotValue = list[first]
Line 2:   PivotPoint = first
Line 3:   for index = first+1 to last do
Line 4:       if  list[index] < PivotValue then
Line 5:           PivotPoint = PivotPoint + 1
Line 6:           Swop(list[PivotPoint], list[index])
Line 7:       end if
Line 8:   end for
Line 9:   Swop(list[first], list[PivotPoint])
Line 10:  return Pivotpoint
```

Please refer to Appendix B, for those readers interested in knowing how the partition function works. Also, a Quicksort Algorithm example can be found in Appendix C.

6.4.3 More about Quicksort

C.A.R. Hoare presented the original algorithm and several of its variations in 1962 (JaJa 2000). In spite of the amount of research done to develop faster sorting algorithms in the past 48 years, it is quite surprising that Quicksort is still the best-known practical sorting algorithm. In regard to the selection of the pivot, there are three basic methods. The first method, which is the simplest, is to select the first element as the pivot. However, this method has a shortcoming. If the input array is almost sorted, the array will be partitioned very unevenly, resulting in poor performance of the Quicksort algorithm (JaJa 2000).

The second method is to select an element randomly, to be the pivot. A random-number generator can be used to help in the selection process (JaJa 2000).

The third method, which is the best, is to select the median as the pivot. If the exact median is required, the input array has to be sorted. Therefore, an approximate median is selected by computing the median of a small subset of the input array (JaJa 2000).

6.5 Fast 2D Median Filtering Algorithm

Thomas S. Huang, George J. Yang , and Gregory Y. Tang presented their paper, "A Fast Two-Dimensional Median Filtering Algorithm", in 1979. They applied their 2-D Median Filter to digital image processing, using an $(m)\times(n)$ window,(that is, an m by n window) where m is along horizontal direction and n is along vertical direction (Huang & et al. 1979).

Assuming that the grey levels inside the window is random, the average number of comparisons for Quicksort is typically proportional to mn(Huang & et al. 1979). The Fast 2-D Median Filtering Algorithm requires only approximately (2n+10) comparisons.

The $(m)\times(n)$ window is moved one column by one column across the image. After the $(m)\times(n)$ window is moved to a new position, n points on the left are out of the window, while n points on the right enter the window. The rest of the points which amount to

(mn-2n), are still inside the window (Huang & et al. 1979). The fact that those points which are still inside the window are sorted, can be exploited to minimize the number of comparisons in finding the median.

The following steps describe the 2-D Median Filtering processes:

Step 1: In the first window, calculate the median by establishing the grey level histograms. Let ltmdn be the number of pixels less than median. Count ltmdn of the first window (Huang & et al. 1979, Pitas 2000).

Step 2: Move the window to the right by one column, so that the left column of the first window is deleted, and a new column is added to the right. Update the histograms and count ltmdn of the current window (Huang & et al. 1979, Pitas 2000).

Step 3: Let T=mn/2 be the threshold. T actually is the average position. The median should lie in the proximity of T. If ltmdn > T, then the median in the current window is smaller than the median in the first window. Therefore, move down the histograms, one bin at a time. Count and update ltmdn, until ltmdn $\leq$ T. At this condition, condition at which ltmdn $\leq$ T, the median bin is reached (Huang & et al. 1979, Pitas 2000).

Step 4: Repeat Step 2 and Step 3, until boundary of the image is reached (Huang & et al. 1979, Pitas 2000).

6.5.1 Fast 1D Median Filtering Algorithm

If n = 1, the two-dimensional (m)×(n) window will be reduced to one-dimensional.

The following example illustrates an one-dimensional application of steps 1, 2, 3.

Let the first window be:

1	7	3	6	0	2	5	4	8

which are 3-bit grey levels. That is, the grey levels are from level 0 to level 8.

The first window has to be sorted by conventional sorting methods, either by bubble sort, or Shellsort, or Quicksort.

After sorting

0, 1, 2, 3, 4, 5, 6, 7, 8

Therefore, median = 4

ltmdn = 4 (There are 4 pixels which are less than the median.

They are 0, 1, 2, 3).

The window is moved to the right. Suppose on the left, 1 is deleted; on the right, 3 is added.

	7	3	6	0	2	5	4	8	3

Delete 1 and add 3 to the sorted list.

0, 2, 3, 3, 4, 5, 6, 7, 8

One element (1) less than the first window median (4) is deleted and one element (3) less than the first window median (4) is added.

ltmdn = 4 -1 +1 =4 (There are still 4 pixels which are less than the first window median 4).

The median of the second window is still equal to 4.

The window is moved to the right. Suppose on the left, 7 is deleted, and on the right, 6 is added.

		3	6	0	2	5	4	8	3	6

Delete 7 and add 6 to the sorted list.

0, 2, 3, 3, 4, 5, 6, 6, 8

No element less than the second window median (4) is added or deleted.

ltmdn = 4 -0 +0 =4

The median of the third window is still equal to 4.

The window is moved to the right. Suppose on the left, 3 is deleted, and on the right, 8 is added.

			6	0	2	5	4	8	3	6	8

Delete 3 and add 8 to the sorted list.

0, 2, 3, 4, 5, 6, 6, 8, 8

One element (3) less than the third window median (4) is deleted, and no element less than the third window median (4) is added.

ltmdn = 4 -1+0 = 3 (There are 3 pixels which are less than the third window median 4. They are 0, 2, 3).

Move up one bin (move one step to the right).

Update ltmdn = 3 + 1 = 4.

The median of the fourth window is 5.

The window is moved to the right. Suppose on the left, 6 is deleted, and on the right, 1 is added.

				0	2	5	4	8	3	6	8	1

Delete 6 and add 1 to the sorted list.

0, 1, 2, 3, 4, 5, 6, 8, 8

No element less than the fourth window median (5) is deleted, and one element (1) less than the fourth window median (5) is added.

ltmdn = 4 -0+1 = 5 (There are 5 pixels which are less than the fourth window median 5. They are 0,1,2, 3,4).

Move down one bin (move one step to the left).

Update ltmdn = 5 - 1 = 4.

The median of the fifth window is 4.

6.5.2 Huang's Algorithm in 2D Median Filtering

The following pseudo-code of Huang's Algorithm (Huang & et al. 1979) is the procedure for finding the median. There are 5 loops. Numbers are put in brackets so that loops can be identified easily.

hist is histogram array.

mdn is median value in a window.

```
if (3) ltmdn > T  then
/*the median in the current window is smaller than the median
in the previous window*/
  repeat (2)
      begin (1)
          mdn = mdn - 1
/*move down one histogram bin*/
          ltmdn = ltmdn - hist[mdn]
/*update counter ltmdn*/
      end (1)
  until (2) ltmdn less than or equal to T
/*mdn is the desired median*/
else (3)
  while (4) ltmdn + hist[mdn] less than or equal to T
/*the desired median is (still) greater than mdn*/
  do (4)
      begin (5)
          ltmdn = ltmdn + hist[mdn]
/*update counter ltmdn*/
          mdn = mdn + 1
/*move up one histogram bin*/
      end (5)
end (3)
```

Huang's Algorithm compares ltmdn (number of pixels less than median) with T (T = mn/2 = half the total number of pixels inside the window).

Therefore, the median found by Huang's Algorithm is a close approximation, not an exact value. When T is large and the number of grey levels is large, the approximation is very close to the exact value. The grey levels in Huang's Algorithm are from zero to 255, and T is also large. Thus Huang's Algorithm is acceptable.

In our one-dimensional case, where the length of the window is 9, ltmdm cannot be compared with T, which is T = 9/2 = 4.5.

For the fourth window, ltmdn = 3. When ltmdn is updated to ltmdn = 3 + 1 = 4, the answer is correct already.

By following Huang's Algorithm requiring ltmdn > T = 4.5, then one step will be moved to the right, ltmdn = 4 + 1 = 5. The answer will be wrong.

In our one-dimensional example, ltmdn is kept to be 4, which will give the correct answer. The reason is simple. There are 9 elements. After the 9 elements are sorted in ascending order, the 5th element is the median, so there are 4 elements less than the median.

6.6 Sorting Algorithm Performance

Experiments on Shellsort, Quicksort and Fast 1D Median Filtering are performed to find the medians for different window sizes.

6.6.1 Shellsort Performance

The Shellsort program is written by following the literature review in Section 6.3. The source codes will be included in Appendix D and the program results are shown in Figure 6.2.

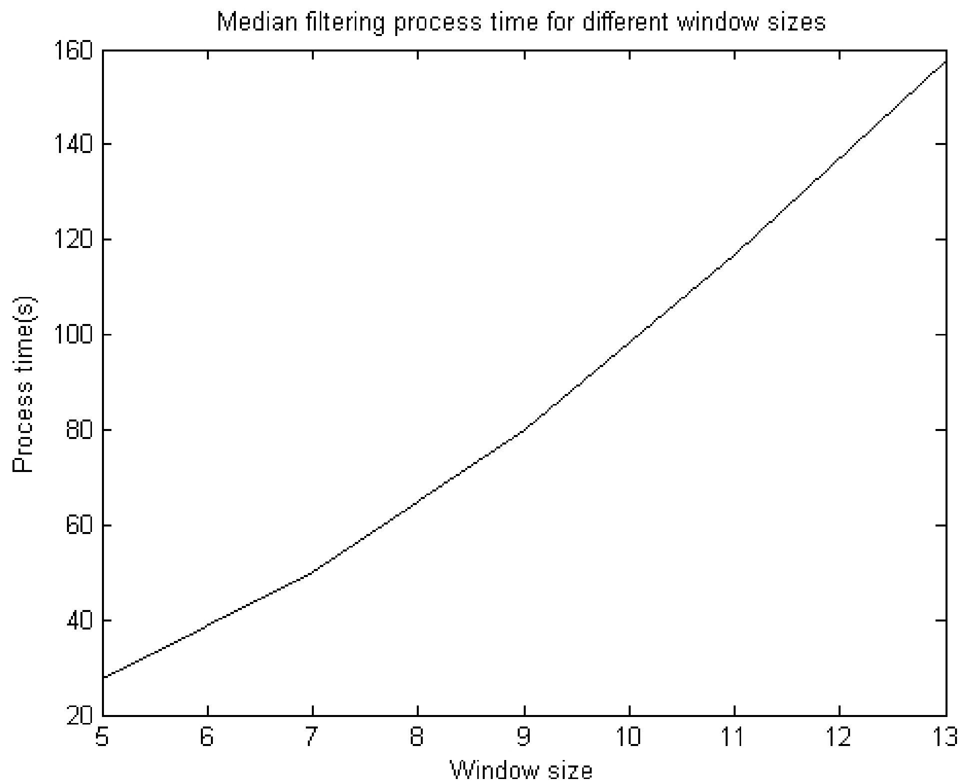

Figure 6.2: Experimental results of Shellsort

The graph in Figure 6.2 illustrates the time required for the median computation for different window sizes. It is found that computational time is directly proportional to the window size.

6.6.2 Quicksort Performance

The function Partition in our computer program Quicksort follows closely the algorithm in Section 6.4, but there are minor changes in the main program Quicksort.

The source codes will be included in Appendix E and the program results are shown in Figure 6.3.

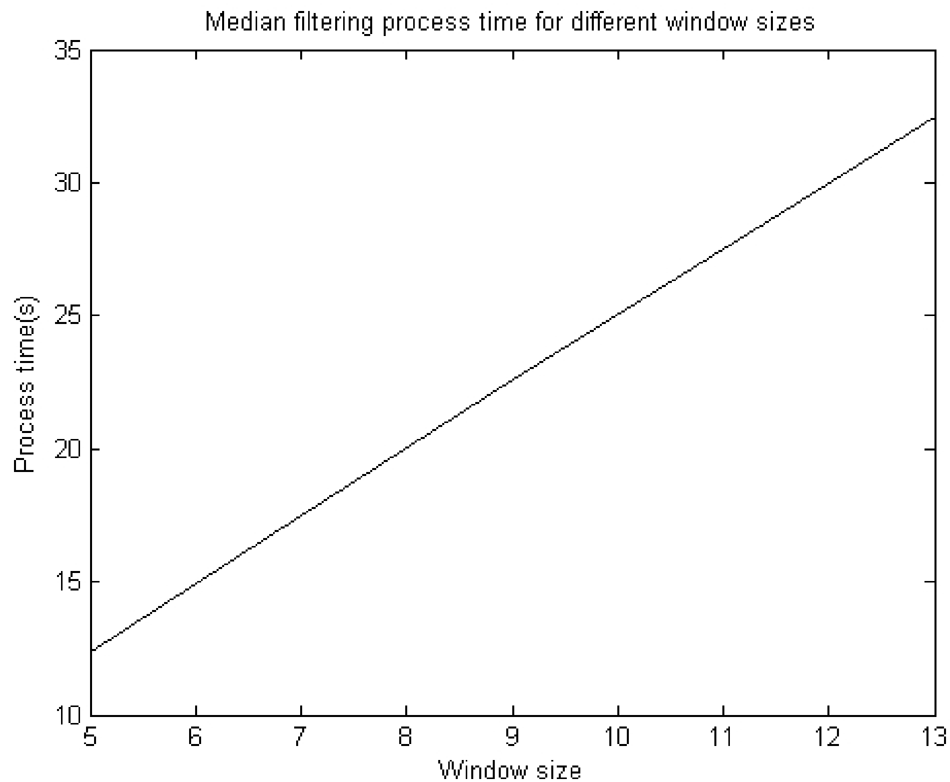

Figure 6.3: Experimental results of Quicksort

6.6.3 Fast 1D Median Filtering Algorithm Performance

In Huang's Algorithm, all data have values ranging from zero to 255. In this research, data values vary greatly. Therefore, Huang's Algorithm cannot be directly changed to one-dimensional by setting n =1 to be used in this research.

Our one-dimensional fast median filtering program works as follows:
In the one-dimensional fast median filtering program, the first window is sorted by bubble sort. The sorted outcomes are stored in array sorted-sublist.
Consider the example of the first window with 9 elements. Threshold = Truncated [9/2]=4. Median is the (threshold+1 = 4+1 = 5th) element in the sorted-sublist. There are 4 (threshold=4) elements less than median (less_than_median = threshold).

Consider the example that

a = [1,7,3,6,0,2,5,4,8,3,6,8,1] is to be scanned by a window of length 9.

The window_size = 9

length_a = 13

next_element_index are indices 10, 11, 12, 13, which are 9+1 to 13, that is 10:13. As the window is moved to the right by one step, 1 is out of the window, and 3 enters the window.

In the sorted-sublist, 1 is found and then deleted. In the sorted-sublist, elements less than 3 are found and elements greater than 3 are found. Then, 3 is inserted to the correct location in the sorted-sublist. From the sorted-sublist, the element at location, (threshold+1)=(4+1)=5, is taken as the median.

The source codes for 1D Median Filtering Algorithm will be included in Appendix F and the program results are shown in Figure 6.4.

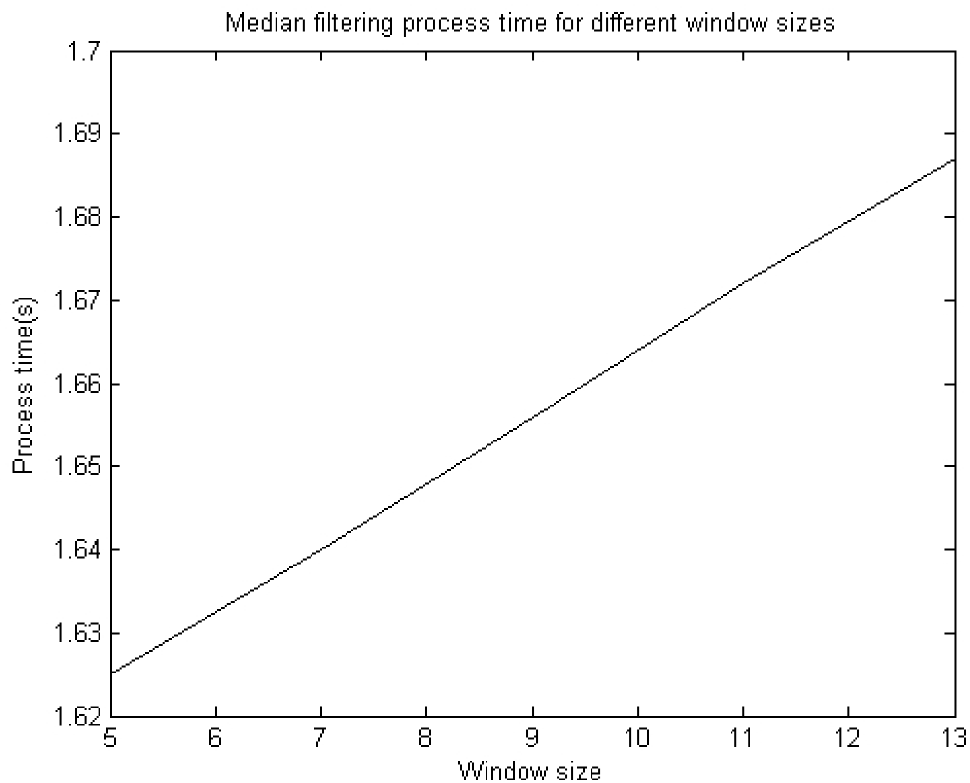

Figure 6.4: Experimental results of Fast 1D Median Filtering Algorithm

6.6.4 Comparison of Experimental Results

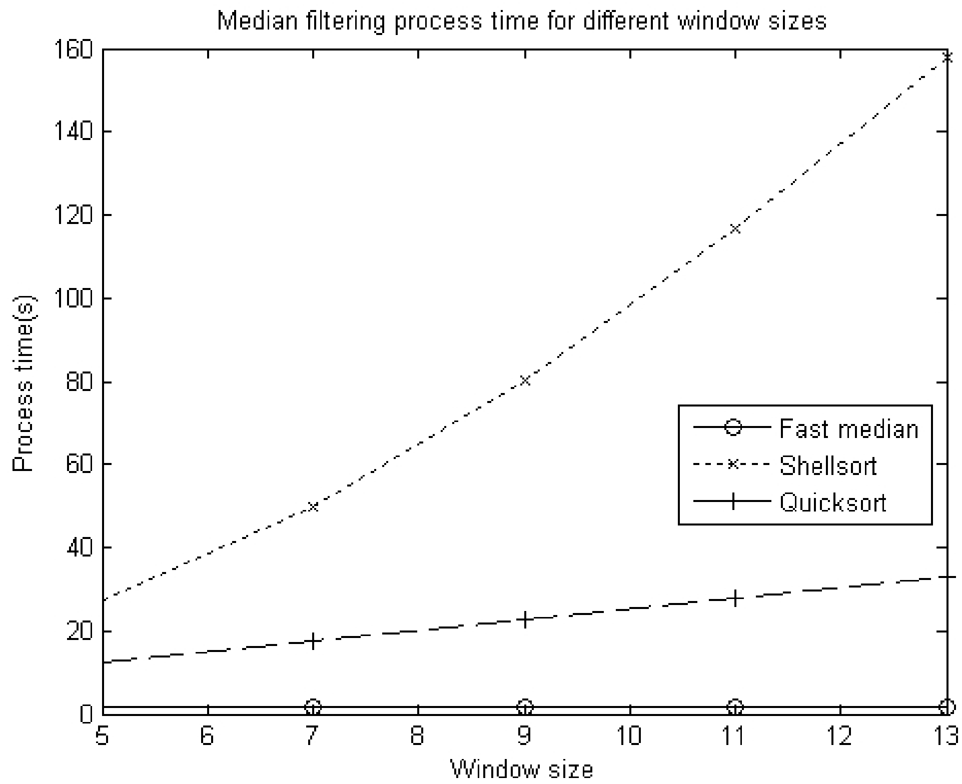

Figure 6.5: Comparison of the three sorting algorithms

When comparing the experimental results in Shellsort, Quicksort and Fast 1D Median Filtering, it is found that 1D Median Filtering has the best performance in terms of computational time to find the median for different window sizes.

6.7 Analyses of Memory Access in Algorithms

The memory access figures depend on the input array values. For the evaluation of different sorting algorithms, arrays of random numbers should be used. In this section, an array of [5, 4, 3, 2, 1] is used for simplicity. The memory access in Shellsort, Quicksort and Fast 1D Median Filtering are analyzed by using the method mentioned in Section 5.8.1 Example on Memory Access.

6.7.1 Analysis of Memory Access in Shellsort

The Shellsort program is in Appendix D. An input array = [5,4,3,2,1] is sorted by the Shellsort program to an output array=[1,2,3,4,5]. The number of memory access and the number of non-memory access instructions are counted for this sorting process.

The Shellsort consists of a Shellsort main loop and an insertion sort function. The execution control starts from the main loop. Then, the execution control goes into the insertion sort function. After the execution of the insertion sort function, the execution control returns to the main loop.

Table 6.2 shows the analysis of the Shellsort main loop. The line numbers in Table 6.2 refer to the line numbers in Appendix D.

Line No.	Memory Access(Y/N)	Number of Times Being Executed
26	N	3
27	N	3
29	N	12
31	Y	11
33	N	11
34	Y	8
37	N	11
39	Y	3
40	Y	3

Table 6.2: Memory Access Analysis on Shellsort main loop

In the Shellsort main loop, the number memory access is 25, and the number of non-memory access is 40.

Table 6.3 shows the analysis of the insertion function. In the insertion function, the number of memory access is 58, and the number of non-memory access is 45.

For the whole Shellsort program, the grand total of memory access is 83, and the grand total of non-memory access is 85.

Line No.	Memory Access(Y/N)	Number of Times Being Executed
29	N	15
30	Y	15
31	N	15
32	Y	15
34	N	15
36	Y	2
38	Y	13
41	Y	13

Table 6.3: Memory Access Analysis on insertion function

6.7.2 Analysis of Memory Access in Quicksort

The Quicksort program is in Appendix E. The Quicksort program is applied to sort the same input array=[5,4,3,2,1]. The numbers of memory access and non-memory access are counted.

The Quicksort consists of a Quicksort main loop and a partition function. The execution control starts from the main loop. It then goes into the partition function. After that, it returns to the main loop, and then goes into the partition function again. This goes on recursively until the input array is sorted.

Table 6.4 shows the analysis of the Quicksort main loop.

Line No.	Memory Access(Y/N)	Number of Times Being Executed
23	N	3
25	Y	2
28	Y	2
32	Y	2
34	Y	2
39	Y	1

Table 6.4: Memory Access Analysis on Quicksort main loop

In the Quicksort main loop,

memory access=9

non-memory access=3

Table 6.5 shows the analysis of the partition function.

Line No.	Memory Access(Y/N)	Number of Times Being Executed
19	N	2
20	Y	2
21	Y	2
23	N	7
24	N	7
25	N	4
27	Y	4
28	Y	4
29	Y	4
33	Y	2
34	Y	2
35	Y	2

Table 6.5: Memory Access Analysis on partition function

In the partition function,

memory access=22

non-memory access=20

Grand total of memory access=31

Grand total of memory access=23

6.7.3 Analysis of Memory Access in Fast 1D Median Filtering

The Fast 1D Median Filtering Algorithm is in Appendix F. The input array is [9,8,7,6,5,4,3,2,1]. The window size is 5.

Firstly, [9,8,7,6,5] are put inside the window, and are sorted to [5,6,7,8,9] by bubble

sort. The third element in the sorted array, that is 7, is taken out as the median.

Secondly, the window moves to the right by one step. 9 goes out of the window and sorted array becomes [5,6,7,8]. 4 moves into the window. 4 is inserted into the correct position in the sorted array which becomes [4,5,6,7,8]. There is no bubble sort but a simple step of insertion. Then the third element, that is 6, is taken out as the median. The window keeps on moving to the right until the array [9,8,7,6,5,4,3,2,1] is finished, giving the last median as 3.

If Shellsort and Quicksort are applied, the process will be:

1. [9,8,7,6,5] being inside the window; then sorted to [5,6,7,8,9]; then 7 taken out as median.
2. [8,7,6,5,4] being inside the window; then sorted to [4,5,6,7,8]; then 6 taken out as median.
3. the process carries on until the last median is found from the array [9,8,7,6,5,4,3,2,1].

In Shellsort and Quicksort, sorting is performed whenever a median number has to be generated. In Fast 1D Median Filtering, only the first median number is generated by using bubble sort. Thereafter, the rest of the median numbers require no sorting and they are generated by a step of simple insertion. That is why the Fast 1D Median Filtering is much faster in finding median numbers in the array [9,8,7,6,5,4,3,2,1], when compared to Quicksort and Shellsort.

The Fast 1D Median Filtering Algorithm consists of a bubble sort function and a fast median main loop. During the first time, the bubble sort is executed and then the fast median main loop is executed. From second time onwards, only the fast median main loop is executed, and there is no more bubble sort. Table 6.6 shows the analysis of the bubble sort function.

In the bubble sort function,
memory access=30
non-memory access=46

Line No.	Memory Access(Y/N)	Number of Times Being Executed
28	N	6
29	N	5
30	N	10
31	N	10
32	Y	10
33	Y	10
34	Y	10
35	N	10
38	N	5

Table 6.6: Memory Access Analysis on bubble sort function

Table 6.7 shows the analysis of the fast median main loop.

Line No.	Memory Access(Y/N)	Number of Times Being Executed
29	Y	1
38	N	4
39	Y	4
41	Y	4
42	Y	4
45	N	4
46	Y	4
47	Y	4
48	Y	4
50	Y	4

Table 6.7: Memory Access Analysis on fast median main loop

In the fast median main loop,

memory access=29

non-memory access=8

Since from finding the second median number onwards, only the fast median main loop is executed with memory access equal to 29, non-memory access equal to 8, it is obvious

that the Fast 1D Median Filtering Algorithm is faster than Quicksort and Shellsort, where every time there is sorting, resulting in large memory access and non-memory access.

6.8 Summary

A median number can only be found from a series of numbers which are sorted in ascending or descending order. Sorting is a time consuming process and the time to sort a series of numbers, is directly proportional to the window size of the median filter (Huang & et al. 1979). As the bit rate of wireless networks is increasing, the time required for the median computation becomes critical. The 1D Median Filtering Algorithm developed in this chapter could be applied in this project for sorting purpose to improve computational efficiency.

Chapter 7

Simulations and Analysis of Results

7.1 Introduction

This chapter starts with an overview on the Network Simulator, version 2,(NS2) (ISI 2008*a*) which is used for our simulations. Next, the method to replace weighted-average algorithm with median filter algorithm is briefly described. Some important settings in the simulation script are explained. Experiment 1 is conducted with traffic source Exponential, to compare the goodput and packet drop percentage between weighted-average algorithm and median filter algorithm. Experiment 2 is the same as Experiment 1, except that the traffic source is Pareto. Results of the simulations are analyzed and presented in figures and tables. It is found that median filter can do better than weighted-average in the presence of bursty traffic flows. Finally, simulations of wireless networks are performed.

7.2 The Network Simulator NS2

This chapter presents results obtained from simulations using NS2, which is popular with the network research community. NS2 is a powerful open-source network simulator

capable of simulating complex wired or wireless network infrastructure.

NS2 is an event driven, object oriented network simulator (ISI 2008*c*). It can be used to simulate network protocols such as TCP and UDP, traffic sources such as FTP, CBR, Exponential and Pareto, and queue managements such as RED and DropTail (Issariyakul & Hossain 2009).

The NS2 simulator uses two languages, the C++ and Object-Oriented Tool Command Language (OTcl). The reason to use two languages is due to the fact that different tasks have different requirements. In the simulations of protocols, efficiency in manipulating bytes and run-time speed, are very important. C++ is the suitable programming language to accomplish this task.

On the other hand, in network studies where different network scenarios are tested, the easiness and convenience to change some parameters (such as to change traffic sources, to change bandwidth in links and to change queue types) becomes important. In such cases, OTcl is the appropriate programming language to achieve these tasks, because OTcl can be changed very quickly and interactively (ISI 2008*c*).

The NS2 simulator contains two hierarchies, a class hierarchy in C++ (the compiled hierarchy) and a corresponding hierarchy in OTcl (the interpreted hierarchy) (ISI 2008*c*). The compiled hierarchy and the interpreted hierarchy, have one-to-one correspondence between them. In this way, the compiled C++ objects can be controlled using OTcl programming language.

The protocol TCP Tahoe is implemented by tcp.cc, which is written in C++. This implementation supports slow start, congestion avoidance and fast retransmit, but it does not implement fast recovery.

Our median filter algorithm is included into the tcp.cc source codes to replace the weighted average algorithm. In order for the revised tcp.cc to work properly in NS2, the command "make" is used in the Linux environment to compile it again. After the compilation, the median filter algorithm in tcp.cc can be used for simulations.

7.3 Median Filter Algorithm in TCP Tahoe

The equation to find the Exponential Weighted Moving Average (EWMA) in Jacobson's Algorithm is given as:

$$A_i = (1 - G_1)A_{i-1} + G_1 M_{i-1} \tag{7.1}$$

where A_i is the new Estimated RTT;

A_{i-1} is the old Estimated RTT;

M_{i-1} is the Sample RTT;

G_1 is constant which is equal to 1/8.

The above equation is replaced with

$$A_i = MEDIAN(M_{i-5}, M_{i-4}, M_{i-3}, M_{i-2}, M_{i-1}) \tag{7.2}$$

A review of Jacobson's algorithm in tcp.cc shows that the variable t_srtt_ is used for A_i. The variable t_rtt_ is used for M_{i-1}. Then, EWMA is replaced by the Median equation.

The MEDIAN equation is implemented as follows:

Step (1) Store the latest five M's (Sample RTT's).

Step (2) Move the five M's to an array.

Step (3) Sort the array in ascending order.

Step (4) Take out the median from the array.

Step (5) Put the median back to the variable t_srtt_.

The Median Filter Algorithm inside the tcp.cc is listed in Appendix H.

7.3.1 Median Filters of sizes 7 and 9

The Median Filter Algorithm is firstly used to implement median filter of size 5. Next, the algorithm is used to implement median filter of size 7. There are only minor changes in the algorithm between median filters of size 5 and size 7. The changes are shown below:

To add two more locations to store RTT for sorting: store_6 = 1 and store_7 = 1. (1 is the initial value of RTT).

To change array size from 5 to 7: ara[5] to ara[7].

To increase the maximum size for sampling from 5 to 7: max = 5 to max = 7.

The rest of the tcp.cc program remains the same. The changes in median filter of size 9, are the same as the changes in median filter of size 7.

7.4 Computation of Retransmission Timeout (RTO)

The RTO in Jacobson's Algorithm is calculated by the equation:

$$R_i = A_i + KV_i \tag{7.3}$$

where R_i is the next RTO;
A_i is the new Estimated RTT ;
V_i is the new deviation;
K is a constant which is equal to 4.

In our Median Filter Algorithm, the equation RTO = 1.25 A_i is used.
In RFC (Request For Comments) 793, RTO = 2 A_i. This idea is used in our Median Filter Algorithm, but the 2 is modified to 1.25.

7.5 The Simulation Script

A tcl (Tool Command Language) script, testing.tcl is written to test for the improvement made by median filter which is embedded in the modified tcp.cc. The testing.tcl script is in Appendix I. Two experiments are performed, both of which use the same script testing.tcl, but with different settings. The following paragraphs describe the choices of the settings in the script testing.tcl.

- set donam - there are two choices, 0 and 1. If donam is set to 1, the script will produce a nam (**n**etwork **a**ni**m**ator). For 0, there will be no nam for display.
- set endtime - endtime is the variable that controls how long the script is to be run in seconds. In this script, endtime is set to 85.

- set qsize - qsize is the variable that controls the length of queue in DropTail. In this script, qsize is set to 40.
- set traftype - there are two types of traffic used in the simulations, namely, Exponential and Pareto. traftype (traffic type) can be set to Exponential or Pareto.
- set npeers - the **n**umber of **peers** can be set from 2 to 10. When npeers is 6, there will be 6 source nodes and 6 destination nodes in the nam. The reason why from 2 to 10 is that, in this script, there are only 10 colors in the colorlist, and that limits the number of peers to 10. Npeers cannot be 1 as this will cause run time error when the following statement in the script is run.

```
set [expr 270+20+140/($npeers-1)* ($n-1)]
```

Therefore, npeers can be set from 2 to 10. npeers is set to 6.

7.5.1 Throughput and Goodput

According to RFC 1242, the definition of throughput is :
Throughput is the rate at which none of the transmitted bits are dropped (Bradner 1991).

Also, according to RFC 2647, the definition of goodput is :
Goodput is the number of bits per unit time forwarded to the correct destination, minus any bits retransmitted (Newman 1999).

In these simulation experiments, the aggregate throughput is the sum of the throughput of the six TCP sources.

The following quantities are defined :
tput_1 = throughput of source s1.
tput_2 = throughput of source s2.
tput_3 = throughput of source s3.
tput_4 = throughput of source s4.
tput_5 = throughput of source s5.
tput_6 = throughput of source s6.

aggregate throughput = tput_1 + tput_2 + tput_3 + tput_4 + tput_5 + tput_6

The aggregate retransmitted bytes is the sum of retransmitted bytes of sources s1, s2, s3, s4, s5 and s6, at every sampling instant.

The aggregate goodput is equal to the aggregate throughput minus the aggregate retransmitted bytes.

The aggregate goodput is then multiplied by 8 (so as to change bytes to bits) and divided by 1000 (so as to change bits to kilobits). Therefore, the unit for the aggregate goodput is kilobits/sec.

7.6 Experiment 1: Exponential Traffic Simulation

Jacobson performed a practical experiment and plotted RTT and RTO in the same figure, in order to prove that his Algorithm is superior to RFC 793. Jacobson showed that there were resemblance in shapes and closeness in lines. Our simulations in this section are supported by Jacobson's approach of finding the relationship between RTT and RTO.

In this experiment, npeers is set to 6, and qsize to 40.
There are 6 sources and 6 destinations. s1 sends packets to d1; s2 sends packets to d2, and so on. s1 through s6 are TCP agents. d1 through d6 are TCP-sink agents. TCP-sink agents generate and send acknowledgements to the sources. They then free the received packets.

The traffic source attached to the TCP is Exponential. That is, traftype is set to "Exponential". All links from sources to R0 (i.e. s1 to R0, s2 to R0 and so on) are duplex links, with bandwidth of 10 Megabits (Mb), delay of 10 ms, and Drop Tail queues.

R0R1 is a duplex link, with bandwidth of 1 Megabits, delay of 100 ms and Drop Tail queues. All links from R1 to destinations (i.e. R1 to d1, R1 to d2 and so on) are duplex links, with bandwidth of 10 Megabits (Mb), delay of 10 ms, and Drop Tail queues.

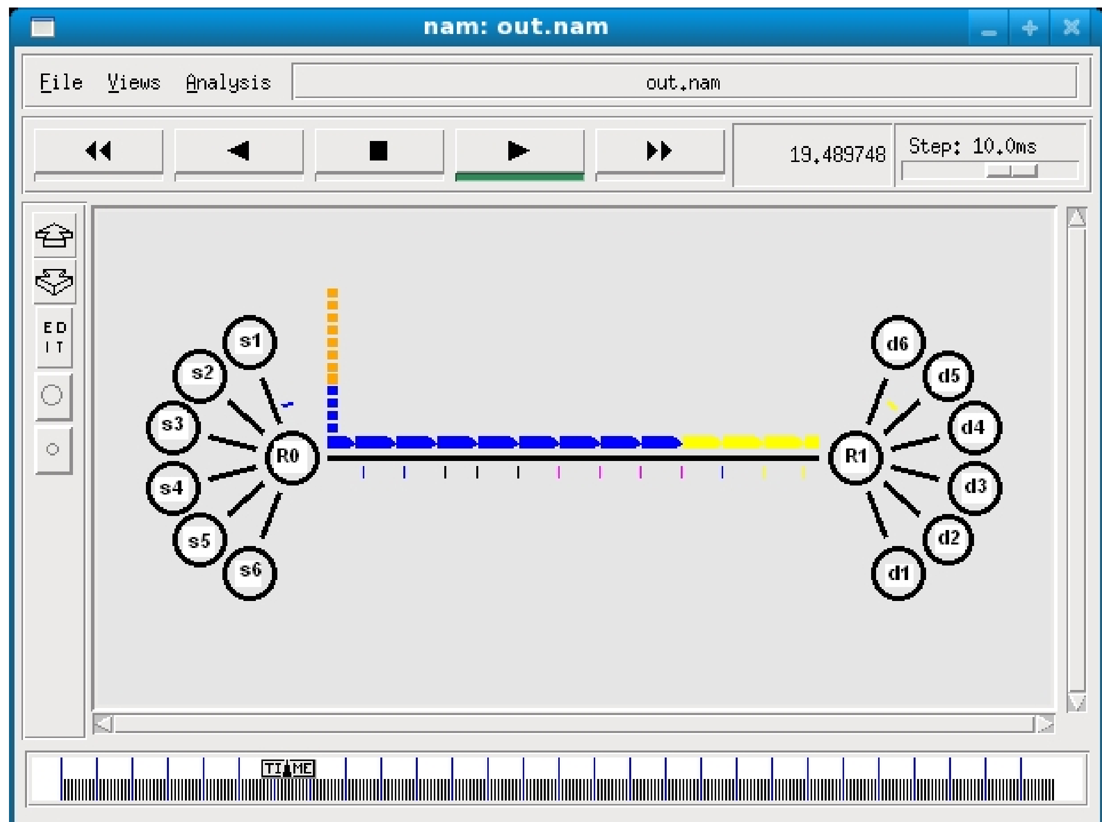

Figure 7.1: Simulation Network of Experiment 1, s1 sends packets to d1; s2 to d2 and so on. Links from sources to RO and from R1 to destinations are 10 Mb. ROR1 is a bottleneck of 1 Mb.

R0R1 is a bottleneck with bandwidth of 1 Megabits only in order to cause congestion. The reason to use DropTail is that, the positive improvement made by median filter, should be easier to be seen in DropTail. The RED gateway is designed to mitigate congestion, and that will make the improvement less obvious.

A small queue size of 40 (i.e. qsize in testing.tcl) in DropTail can also cause congestion.

7.6.1 Settings and Experimental Analysis

testing.tcl script is used for testing goodputs and tcp parameters. Firstly, the original tcp.cc in NS2.33 without median filter is used to run testing.tcl. Secondly, the modified tcp.cc with median filter is used to run testing.tcl. The accumulative goodput are plotted out in Figure 7.2.

In this simulation, the execution time is set at 85 seconds. For every 0.1 second, a sample of goodput is taken. The sampling starts at 2 seconds and ends at 85 seconds, so a total of 840 samples of goodputs are taken.

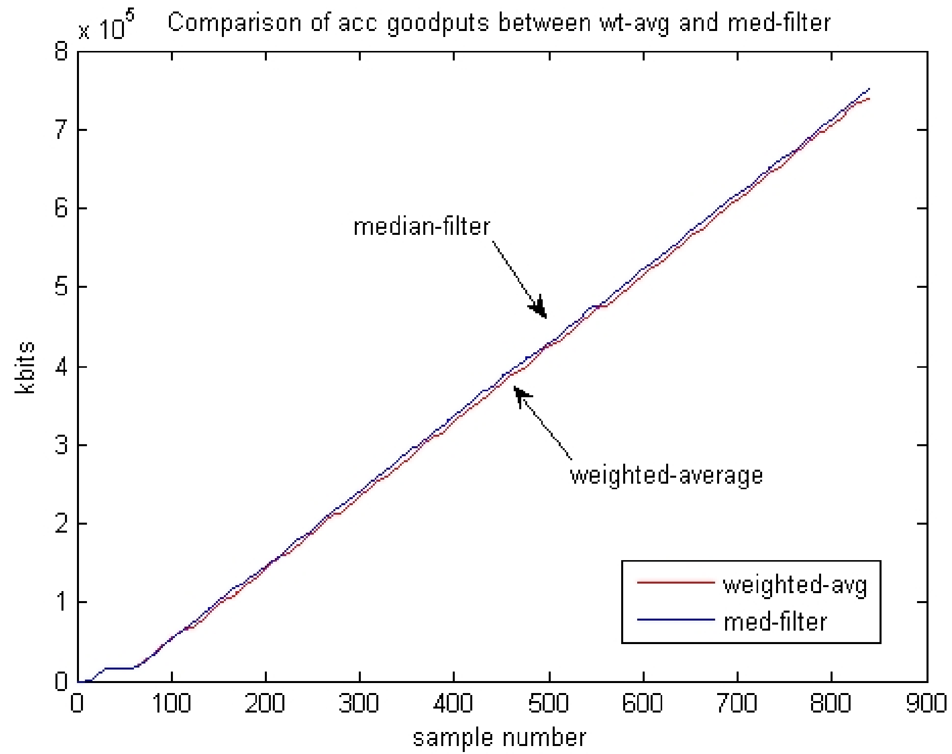

Figure 7.2: Comparison of accumulative goodput between median filter and weighted-average filter. This figure shows that a median filter size 5 can deliver higher goodput.

	throughput	**goodput**	**pkt drop %**
with median filter of size 5	760240	752160	1.06
without median filter	753440	740960	1.66

Table 7.1: Exponential traffic and analysis for median filter size 5 and weighted-average filter

The packet drop percentage is calculated by the formula:

$$\text{packet drop percentage} = ((\text{throughput} - \text{goodput})/\text{throughput}) \times 100\% \tag{7.4}$$

The accumulative goodput is calculated as follows:
The largest sample number is 840. The accumulative goodput at sample number 840 is equal to goodput at sample number 1, plus goodput at sample number 2, number 3 and so on up to sample number 840. Without median filter, the accumulative goodput at sample number 840 is 740960 as shown in Table 7.1.

Two improvements made by median filter of size 5 are:

1. The total goodput from tcp.cc with median filter of size 5 is 752160 while that from the original tcp.cc with Jacobson's weighted-average filter is 740960. Therefore, median filter of size 5 can deliver more goodput.
2. The packet drop percentage of median filter of size 5 is 1.06%, while that of the weighted-average filter is 1.66%. Therefore, median filter of size 5 has lower packet drop percentage.

Next, RTT and RTO are plotted in the same figure for comparison.
Inside the script of testing.tcl, near the end, there are statements

```
#write tcp params to file
```

$currsrtt is added to the end of the following statement, so that currsrtt can be saved to tcpfile.

```
put $tcpfile``···$currtt, $currrttvar···$currsrtt''
```

For median filter of size 5, compute $RTO = 1.25 \times currsrtt$
RTO calculated from Jacobson's Algorithm is $RTO = currsrtt + 4 \times currrttvar$

Please note that in the testing.tcl script, the variable currsrtt (**curr**ent **s**moothed **rtt**) is used for A_i (A_i is the new Estimated RTT). The variable currrttvar (**curr**ent **rtt** **var**iance) is used for V_i (V_i is the new deviation).

Figure 7.3 shows RTT and RTO from the original tcp.cc. Figure7.4 shows RTT and RTO from median filter of size 5.

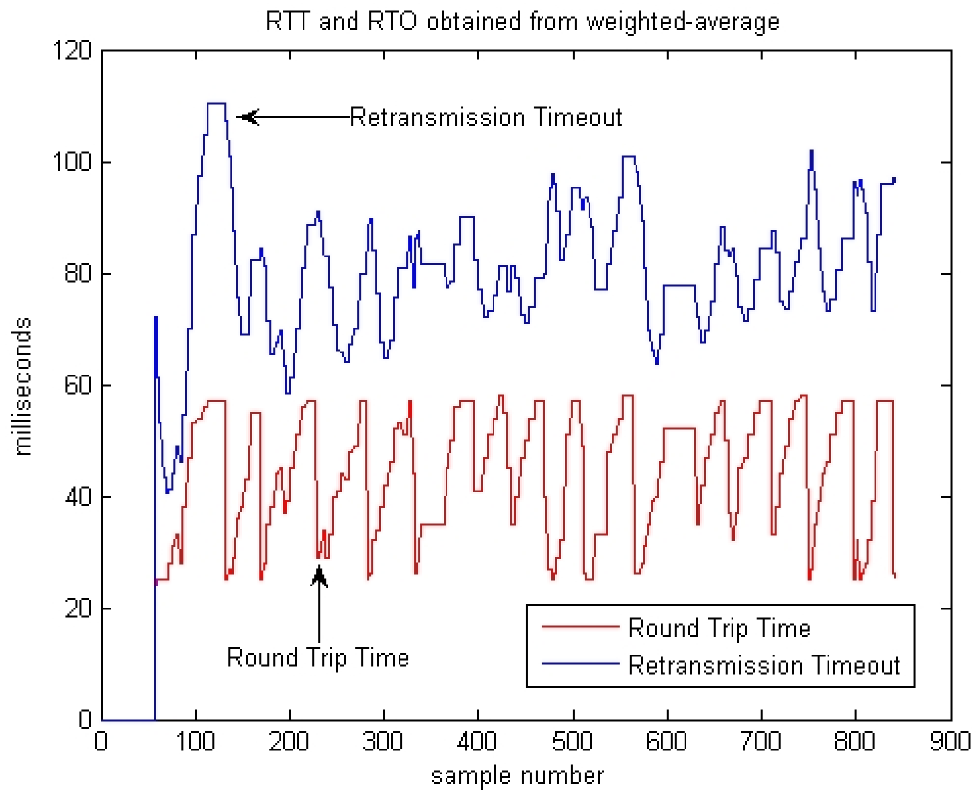

Figure 7.3: RTT and RTO obtained from weighted-average. The RTT curve and RTO curve are farther apart. The response to retransmit when a packet is lost will be too slow.

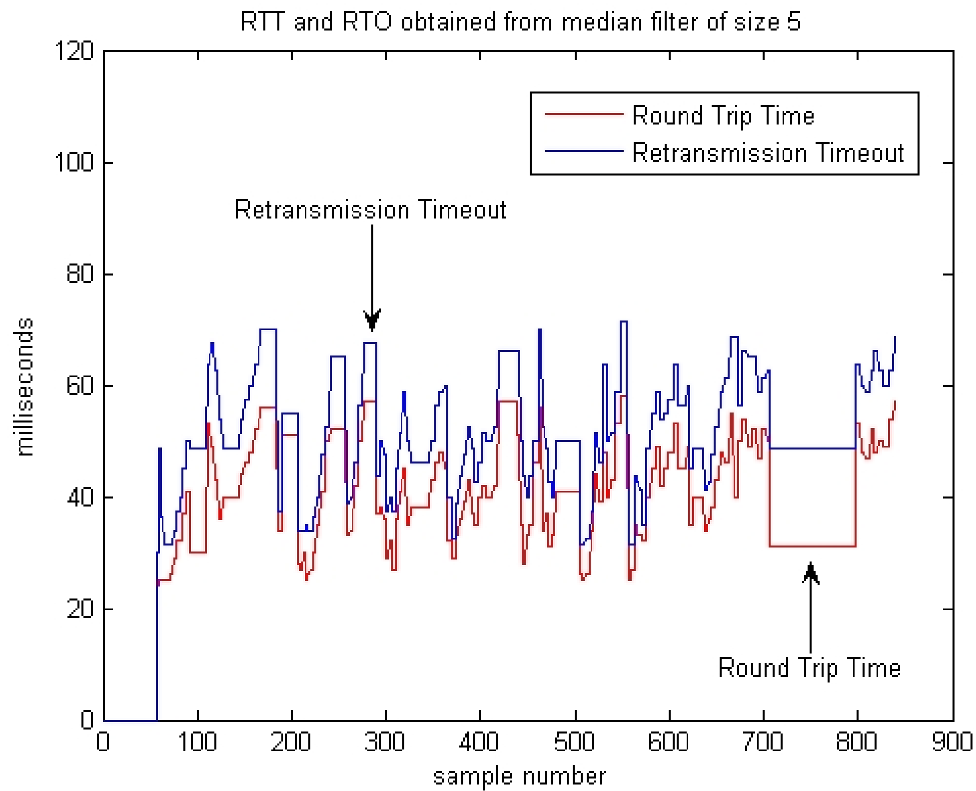

Figure 7.4: RTT and RTO obtained from median filter of size 5. The RTT curve and RTO curve are closer to each other. This will probably increase the throughput.

In Figure 7.3, the RTO obtained from weighted-average algorithm is much larger than the RTT, so the response to retransmit when a packet is lost will be too slow; and this will decrease the throughput of the transmission path (Comer 2006*a*).

A desirable RTO is one which is close to, and always larger than the RTT. In Figure 7.4, the RTO estimated by median filter will probably increase the throughput and at the same time decrease congestion in the transmission path (Comer 2006*a*).

The size of median filter increases from size 5 to size 7 and size 9. The RTO values obtained from median filters of size 7 and size 9 are shown in Figure 7.5 and Figure 7.6.

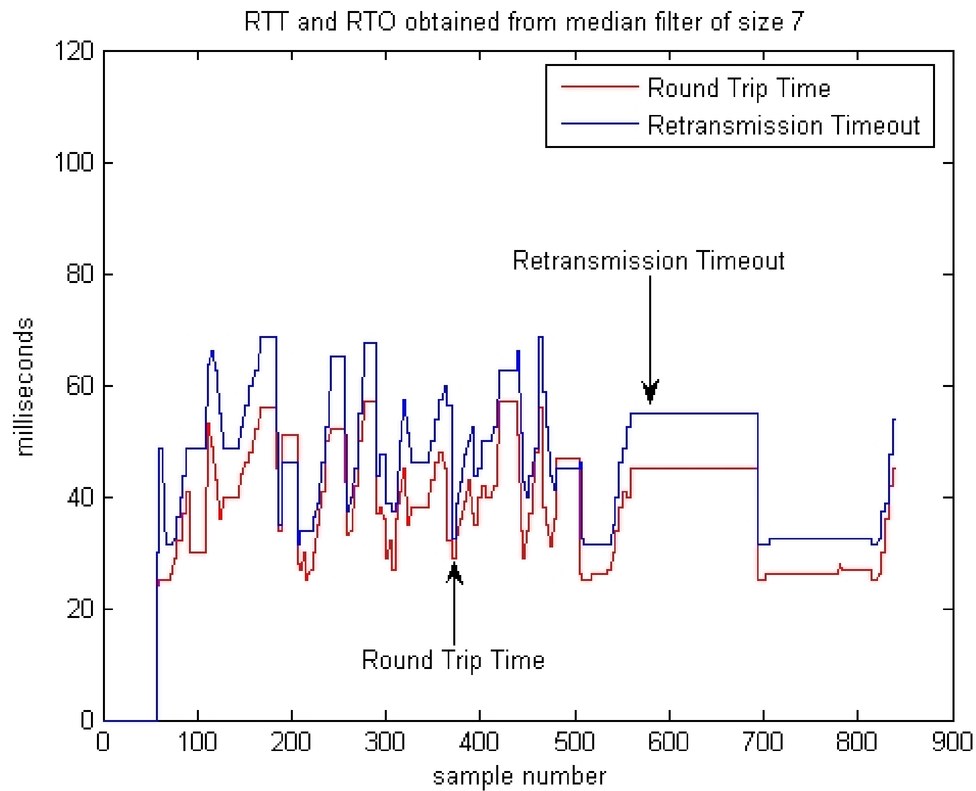

Figure 7.5: RTT and RTO obtained from median filter of size 7. Since median filter of size 7 delivers fewer goodput than size 5 does, size 7 is not the best choice.

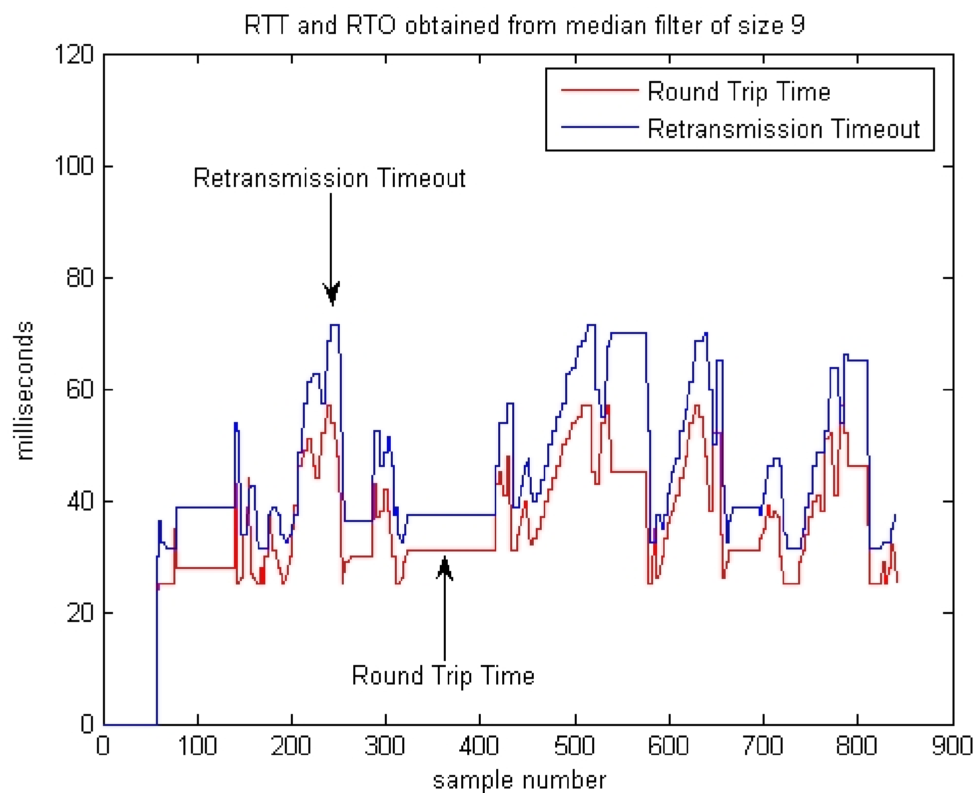

Figure 7.6: RTT and RTO obtained from median filter of size 9. Since median filter of size 9 delivers fewer goodput than size 5 does, size 9 is not the best choice.

The packet drop percentage in the transmission is calculated by the following formula:

packet drop percentage = ((throughput - goodput) / throughput) $\times$ 100%

The packet drop percentages of median filters of size 5, size 7, size 9 and the original tcp.cc are shown in Table 7.2.

	throughput	**goodput**	**pkt drop %**
median filter size 5	760240	752160	1.06
median filter size 7	653280	647360	0.91
median filter size 9	728720	722480	0.86
without median filter	753440	740960	1.66

Table 7.2: Comparison of packet drop percentages. It is found that all packet drop percentages of median filters are lower than that of the weighted-average filter which is 1.66%.

The conclusion is that, all packet drop percentages of median filters are lower than that of the weighted-average which is 1.66%. Another conclusion is that, median filter of size 5 has the highest goodput. There are fewer goodput for median filter of size 7 and size 9, so size 5 is the optimum choice of size.

From Table 7.2, it is shown that the throughput of median filter size 5 is 760240, and that of weighted-average without median filter is 753440. It is due to the following fact: When Figure 7.3, which is RTT and RTO obtained from weighted-average, is compared with Figure 7.4, which is RTT and RTO obtained from median filter of size 5, it is found that in the case of weighted-average, the curve of RTO is further from the curve of RTT. In the case of median filter, the curve of RTO is closer to the curve of RTT. A desirable RTO is one which is close to, and always larger than the RTT. The reason is that, if the RTO is too large, the response to retransmit when a packet is lost will be too slow, and this will decrease the throughput of the transmission path (Comer 2006*a*).

7.7 Experiment 2: Pareto Traffic Simulation

In 1995, two researchers Paxson and Floyd found that FTP data connections had burst arrival rate. In addition, the distribution of the number of bytes in each burst has a

heavy right tail.

In statistics, heavy-tailed distributions are probability distributions whose tails are not exponential bounded. That is, they have heavier tails than the exponential distributions. One of the simplest heavy-tailed distributions is the Pareto distribution, so Pareto traffic is used in simulations in this section to simulate bursty traffic conditions.

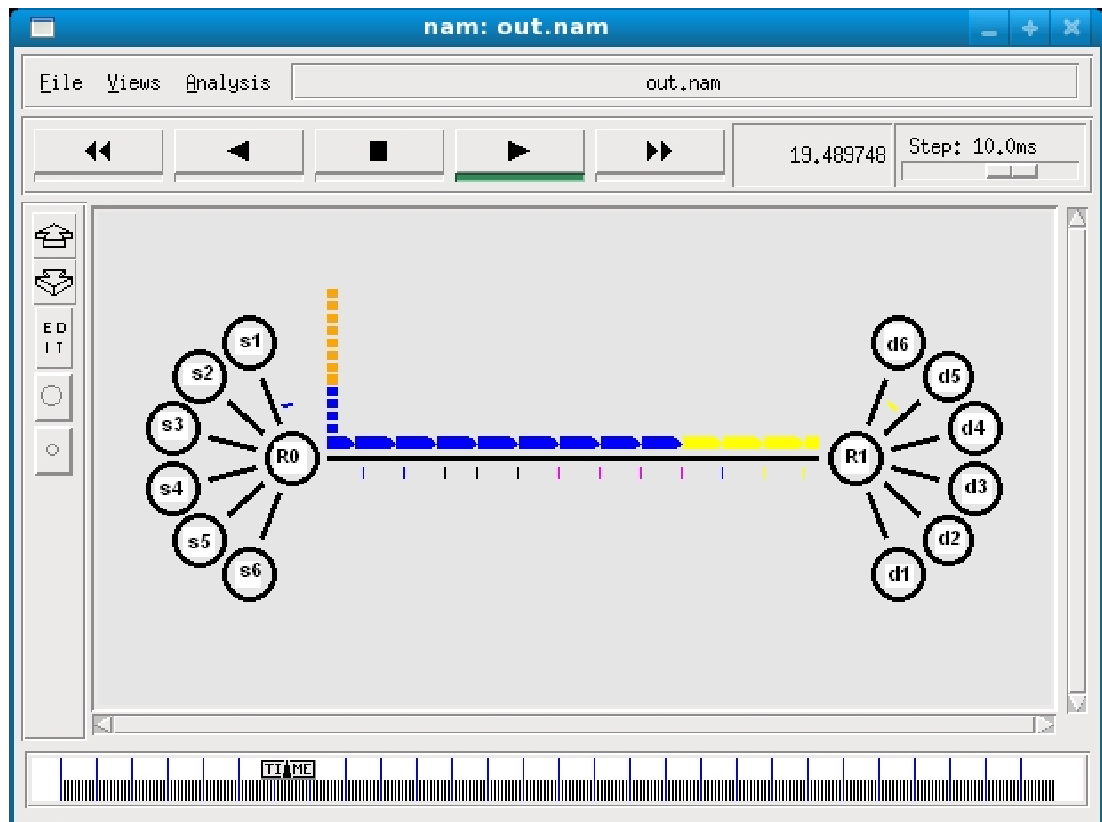

Figure 7.7: Simulation Network of Experiment 2, s1 sends packets to d1; s2 to d2 and so on. Links from sources to RO and from R1 to destinations are 10 Mb. R0R1 is a bottleneck of 1 Mb.

In this experiment, npeers is set to 6 and qsize to 40.

There are 6 sources and 6 destinations. s1 sends packets to d1; s2 sends packets to d2, and so on. s1 through s6 are TCP agents. d1 through d6 are TCP-sink agents. The traffic source attached to the TCP is Pareto. That is, traftype is set to “Pareto”. All links from sources to R0 (i.e. s1 to R0, s2 to R0 and so on) are duplex links, with bandwidth of 10 Megabits (Mb), delay of 10 ms, and Drop Tail queues.

R0R1 is a duplex link, with bandwidth of 1 Megabits, delay of 100 ms and Drop Tail queues. All links from R1 to destinations (i.e. R1 to d1, R1 to d2 and so on) are duplex links, with bandwidth of 10 Megabits (Mb), delay of 10 ms, and Drop Tail queues.

7.7.1 Settings and Experimental Analysis

testing.tcl script is used for testing goodput and tcp parameters. Firstly, the original tcp.cc in NS2.33 which do not have median filter is used, to run testing.tcl. Secondly, the modified tcp.cc with median filter is used to run testing.tcl. The accumulative goodput are plotted out in Figure 7.8.

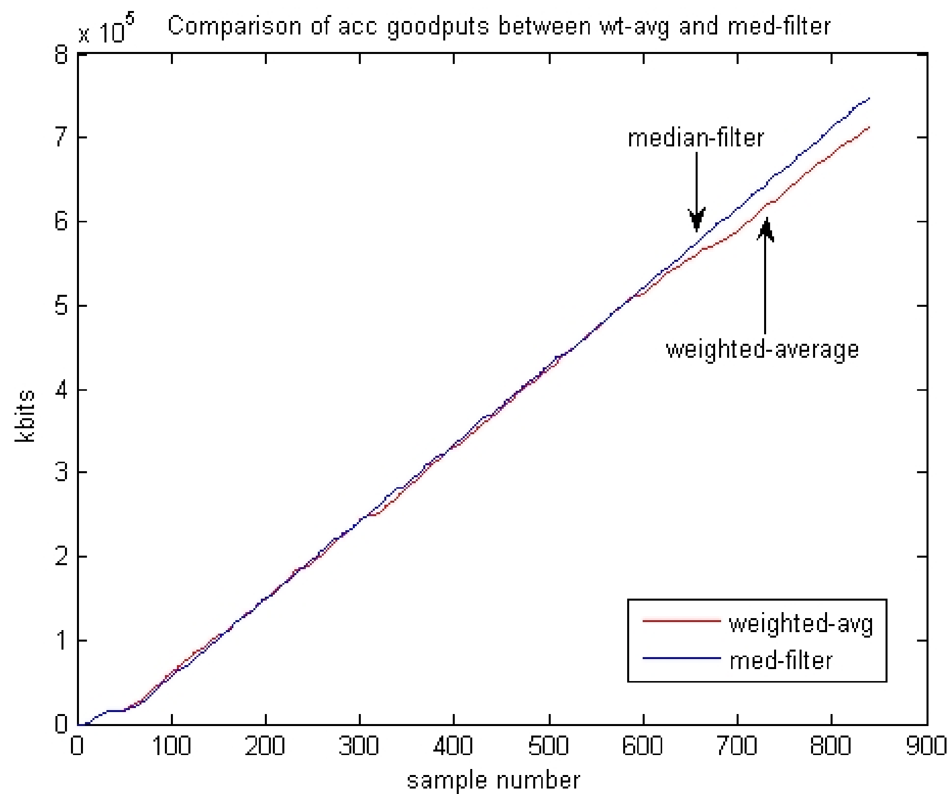

Figure 7.8: Comparison of accumulative goodput between median filter and weighted-average filter. This figure shows median filter size 5 can deliver more goodput.

	throughput	**goodput**	**pkt drop %**
with median filter of size 5	760560	748560	1.58
without median filter	725760	711760	1.93

Table 7.3: Pareto traffic and analysis for median filter size 5 and weighted-average filter

Features in the comparisons are:

1. The total goodput from tcp.cc with median filter of size five is 748560, while that from the original tcp.cc is 711760. Therefore, median filter can deliver more goodput.
2. The packet drop percentage of median filter of size five is 1.58%, while that of the original tcp.cc is 1.93%. Therefore median filter has lower packet drop percentage.
3. TCP Median outperforms TCP Tahoe in throughput by 5%.

Next, the graphs of RTT and RTO are plotted in the same figure for comparison.
Inside the script, there are statements in the procedure tcpparameters:

```
# write tcp params to file
```

$currsrtt is added to the end of the following statement in order to save currsrtt values in the tcp file.

```
put $tcpfile‘‘···$currrtt $currrttvar···$currsrtt’’
```

For median filter of size 5, compute RTO = 1.25 × currsrtt.

RTO for Jacobson's Algorithm is RTO = currsrtt + 4 × currrttvar

Figure 7.10 shows that the RTO from median filter of size 5, is better than the RTO in Figure 7.9 using weighted-average.

Lastly, the median filter increases from size 5 to size 7 and size 9. The RTO obtained from median filters of size 7 and size 9 is shown in Figure 7.11 and Figure 7.12.

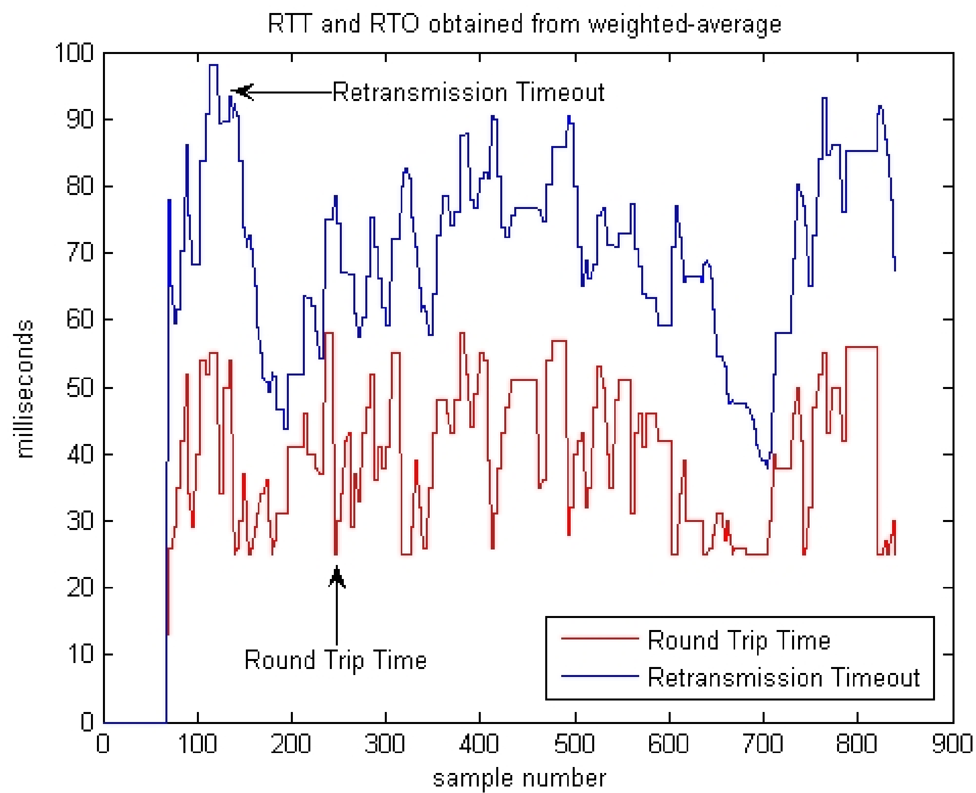

Figure 7.9: RTT and RTO obtained from weighted-average. The RTT curve and RTO curve are farther apart. The response to retransmit when a packet is lost will be too slow.

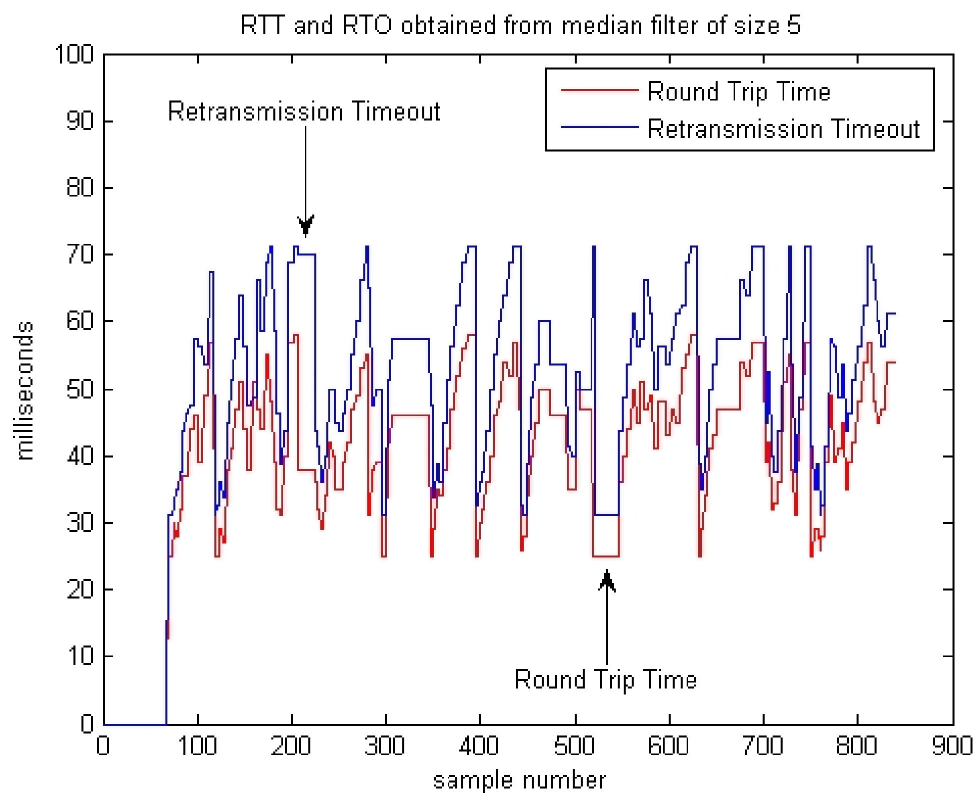

Figure 7.10: RTT and RTO obtained from median filter of size 5. . The RTT curve and RTO curve are closer to each other. This will probably increase the throughput.

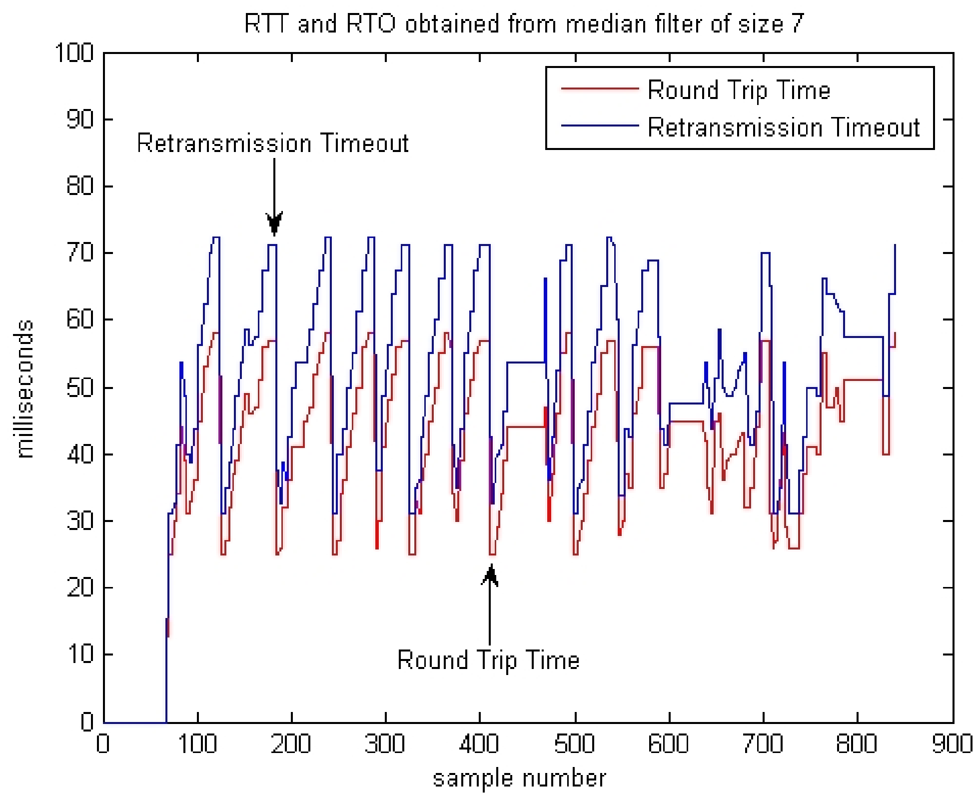

Figure 7.11: RTT and RTO obtained from median filter of size 7. Since median filter of size 7 delivers fewer goodput than size 5 does, size 7 is not the best choice.

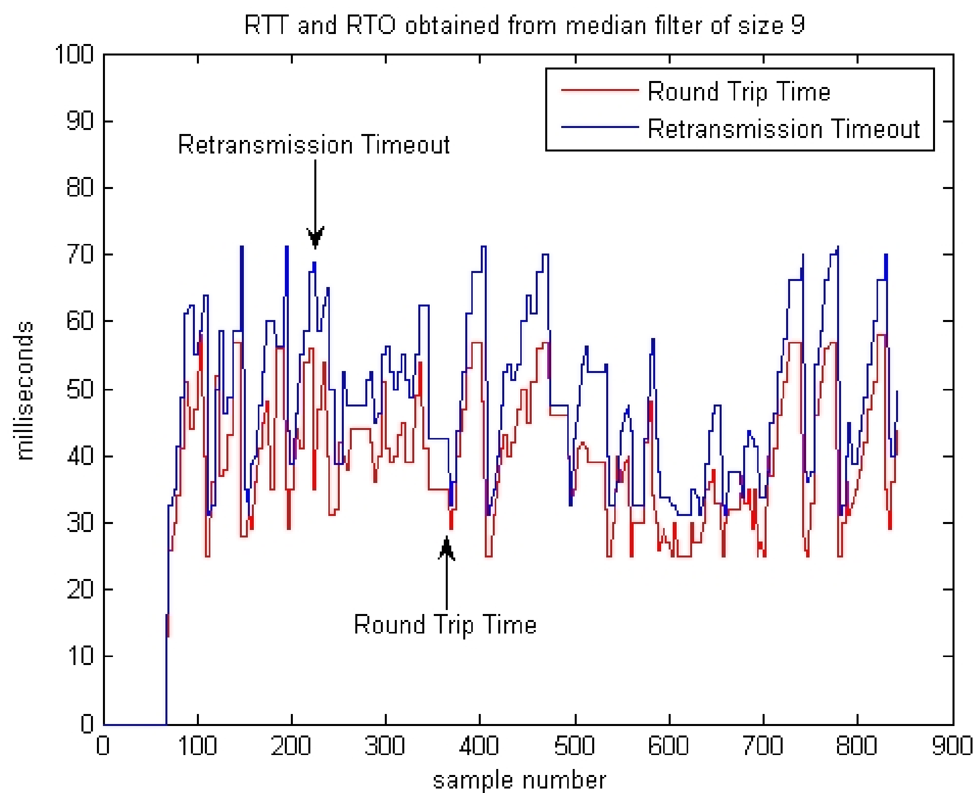

Figure 7.12: RTT and RTO obtained from median filter of size 9. Since median filter of size 9 delivers fewer goodput than size 5 does, size 9 is not the best choice.

The packet drop percentages of median filters of size 5, size 7 and size 9 are shown in Table 7.4.

	throughput	**goodput**	**pkt drop %**
median filter size 5	760560	748560	1.58
median filter size 7	676640	667520	1.35
median filter size 9	756080	744400	1.55
without median filter	725760	711760	1.93

Table 7.4: Packet drop percentages of median filters with size 5, 7 and 9. It is found that all packet drop percentages from median filters are lower than the packet drop percentage from the weighted-average, which is 1.93%.

The packet drop percentage in the transmission is calculated by the following formula:
packet drop percentage = ((throughput - goodput) / throughput) $\times$ 100%

The conclusion for packet drop percentages is that, all packet drop percentages from median filters of sizes 5, 7, 9 are lower than the packet drop percentage from the weighted-average, which is 1.93%

When comparing goodput among median filters of sizes 5, 7, and 9, the goodput from median filter of size 5 is the highest, and it is obvious that, 5 is the best choice of size for median filter.

In the Table 7.4, it is shown that the throughput of median filter size 5 is 760560, and that of weighted-average without median filter is 725760. It is due to the fact:
When Figure 7.9, which is RTT and RTO obtained from weighted-average, is compared with Figure 7.10, which is RTT and RTO obtained from median filter of size 5, it is found that in the case of weighted-average, the curve of RTO is further away from the curve of RTT. In the case of median filter, the curve of RTO is closer to the curve of RTT.
A desirable RTO is one which is close to, and always larger than the RTT. The reason is that, if the RTO is too large, the response to retransmit when a packet is lost will be too slow, and this will decrease the throughput of the transmission path (Comer 2006*a*).

7.8 Error Model for Pareto Traffic

Inside the NS2, there is a built-in error model, which can be used to impose packet losses in the transmission link (Floyd 1997).

7.8.1 Overview of Error Model

In order to introduce errors into network simulations, the error model can be inserted between two nodes. Upon receiving a packet, the error model determines whether to simulate the packet being in error or not. If the packet is simulated not in error, the error model will forward the packet to its peer node. If the packet is to have an error applied to it or its contents, then there are two different ways to process the packet. If a drop-target is specified during the creation of the error model, the error model will drop the packet into the drop-target. A drop-target is specified in our error model, so packets are dropped into the drop-target. If no drop-target is specified, the error model will mark the error-flag in the packet's header. In this case, other agents in the simulation network will have to process the marked packet (ISI 2008*c*).

The command, $loss_model set rate_, is used to set the error rate. The error rates used in the illustration examples in "The ns Manual" are 0.01 and 0.02, so the typical error rates are 1% and 2% (ISI 2008*c*).

The command, $loss_model unit, can be used to set the unit of the error rate. The unit can be in packets or bits. The unit used in our model is packet.

7.8.2 Error Model Applied to TCP Tahoe

In this simulation experiment, all settings and steps in carrying out the experiment, are exactly the same as those in Section 7.7 Experiment 2, except that an error model is installed in the link R0R1 (Fall & Floyd 1996).

The lossrate in the script, testing.tcl is set to 0.01 (i.e. 1%). The script is executed and the throughput then recorded. The lossrate is then set to 0.02, 0.03, 0.04 and so on,

until 1.0 (i.e. 100%). Scripts with these settings are executed and the corresponding throughputs are recorded. These throughputs are plotted against the corresponding lossrates in Figure 7.13.

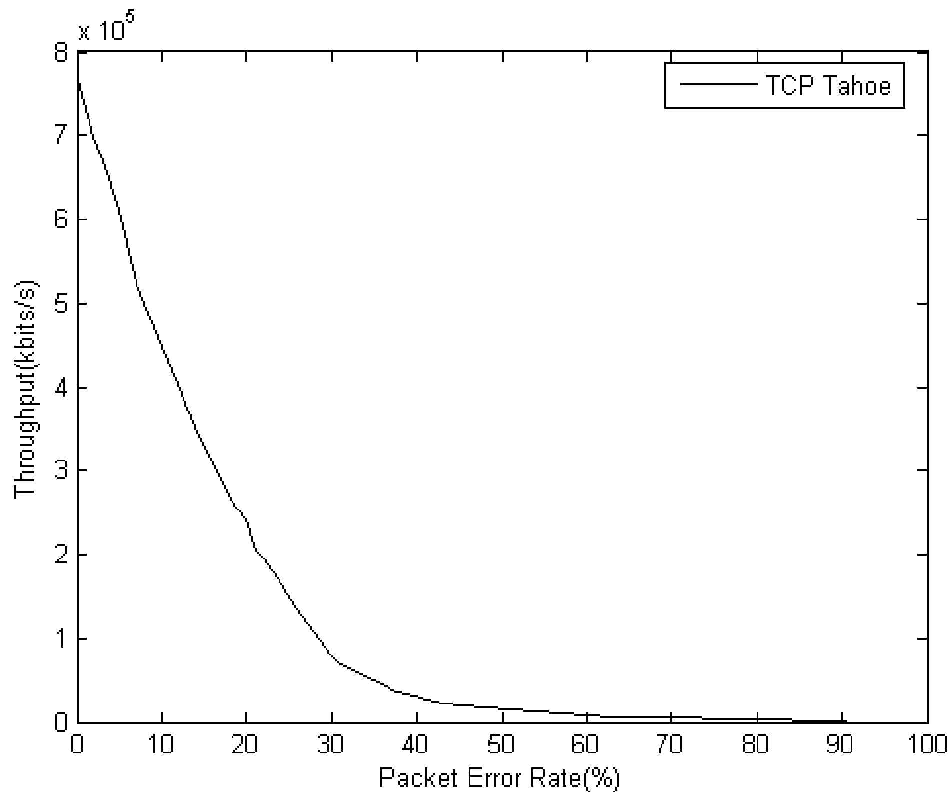

Figure 7.13: TCP Tahoe throughput versus lossrate. As the lossrate increases, the throughput decreases.

Figure 7.13 shows that as the lossrate increases from 0.01 to 1.0, the throughput of TCP Tahoe decreases.

7.8.3 Error Model Applied to TCP Median

This simulation experiment is exactly the same as Section 7.8.2, except that TCP Tahoe is replaced by TCP Median (Fall & Floyd 1996). The lossrate is set from 0.01, 0.02. 0.03 and so on until 1.0. Scripts are executed with these settings, and the corresponding throughputs are recorded. Throughputs are plotted against lossrates in Figure 7.14.

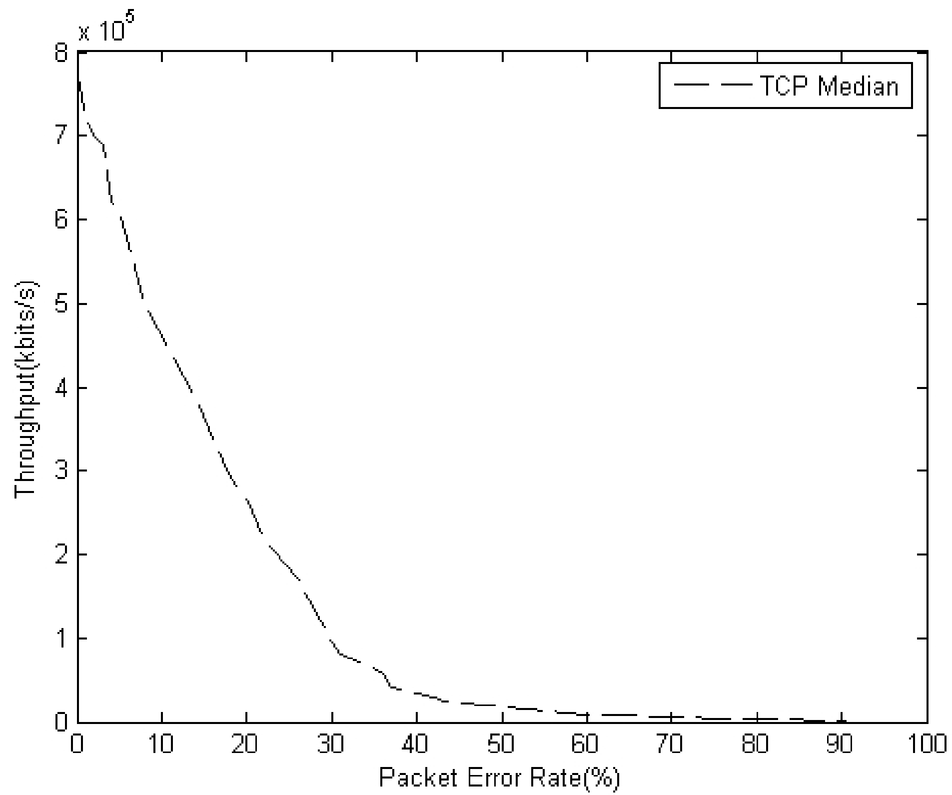

Figure 7.14: TCP Median throughput versus lossrate. As the lossrate increases, the throughput decreases.

Figure 7.14 shows that as the lossrate increases from 0.01 to 1.0, the throughput of TCP Median decreases.

7.8.4 Comparison of TCP Tahoe and TCP Median

In order to compare the throughputs between TCP Tahoe and TCP Median, throughput in Figure 7.13 and Figure 7.14 are plotted in the same Figure 7.15.

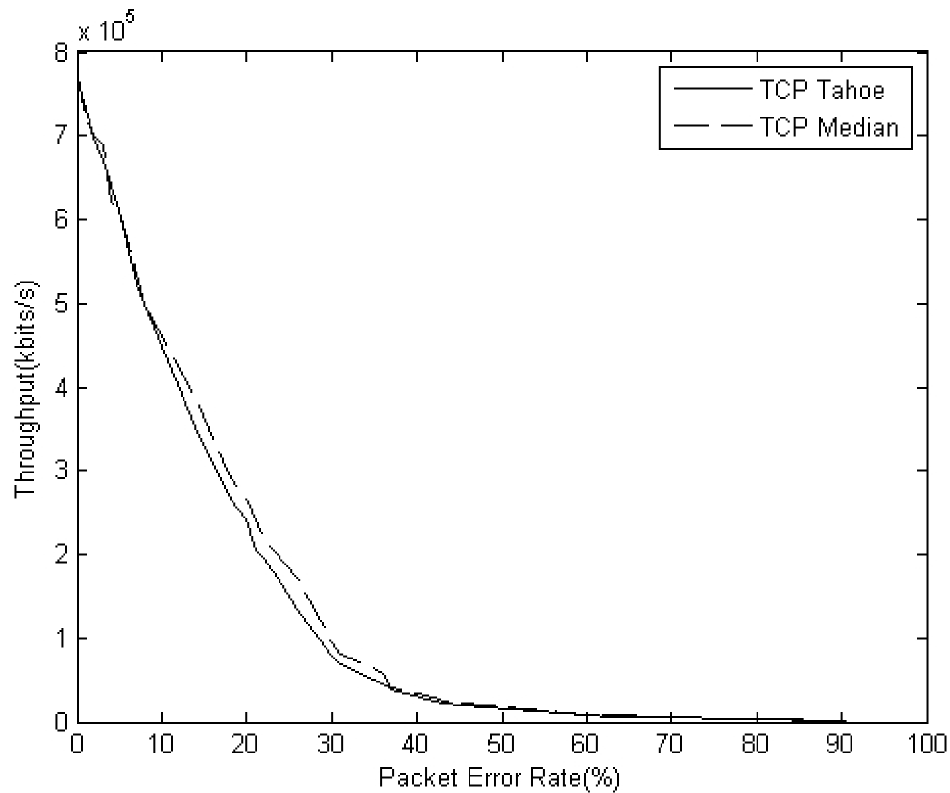

Figure 7.15: Throughput of TCP Median decreases to a lesser extent. At high loss rates, there is little to differentiate the two.

The conclusion is that, as the lossrate increases, both throughputs from TCP Tahoe and TCP Median decrease, but TCP Median decreases to a lesser extent.

7.9 Simulation of Two Wireless Nodes

The simulation in this section consists of two wireless nodes, node_0 and node_1, which are 50 meters apart. For this wireless link, the bandwidth is 2Mb, and the delay is 1ms. For easy investigation, both wireless nodes are not moving.

Both wireless nodes are designed to be stationary because wireless connection throughput can be affected by the movement of the wireless nodes. When stationary, the wireless connection throughput will only depend on the distance between the two wireless nodes, and the propagation model used in the simulation.

The propagation model describes how wireless signals travel in an environment. For a more realistic simulation, shadowing model is chosen for the propagation model. Outdoor environment is chosen in this simulation. This type of environment yields lower wireless path loss, and less multipath propagation effects.

A TCP source is attached to node_0, and a TCP sink is attached to node_1. Then an Application/FTP is attached to the TCP source.

7.9.1 TCP Connection over a Wireless Link

A test is performed by sending FTP from source to sink. This test ensures that a wireless connection is established, before an error model is imposed to the wireless link.

7.9.2 Error Model Imposed on the Link

The error model will be defined as :

```
proc UniformErr {} {
        global ns_ err lossrate
        set err [new ErrorModel]
        $err set rate_ $lossrate
        $err unit pkt
        $err ranvar [new RandomVariable/Uniform]
        return $err
}
```

The Tcl script is in Appendix J. This wireless connection with error model imposed, is run with TCP Tahoe and then run with TCP Median. The throughputs form TCP Tahoe and TCP Median are plotted in Figure 7.16. For packet loss rate smaller than 0.01%, TCP Tahoe and TCP Median have similar performance. At packet loss rate of 0.1%, TCP Median starts to outperform. TCP Median delivers more throughput than TCP Tahoe by 10%, at a very practical loss rate of 1%. After packet loss rate of 10%, there is little to differentiate.

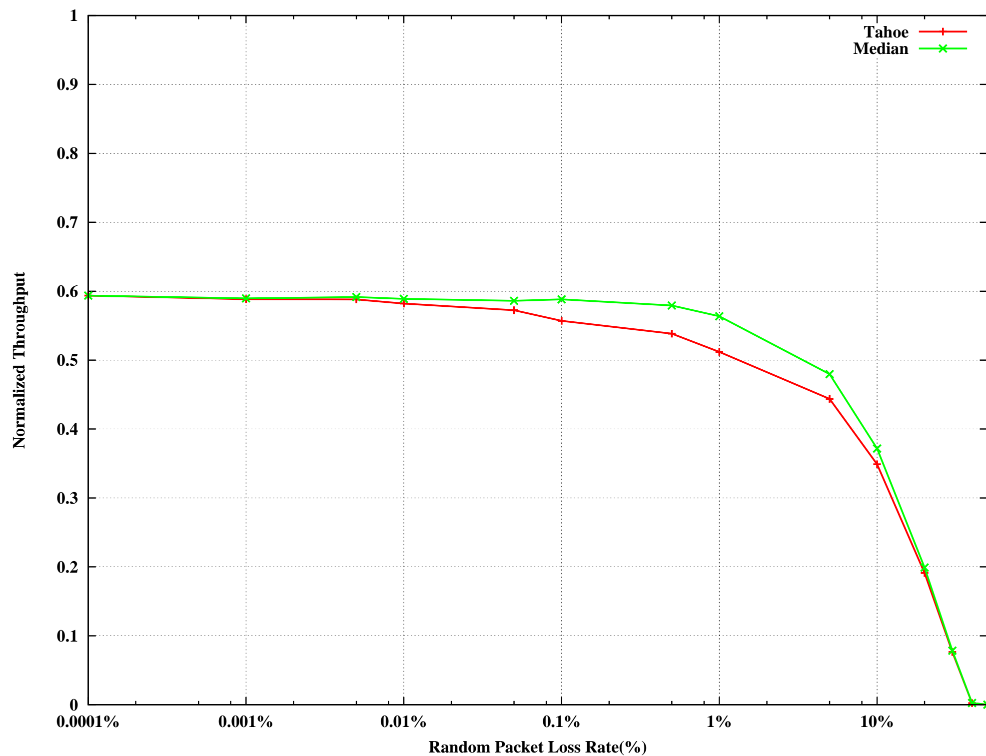

Figure 7.16: Throughputs of TCP Tahoe and TCP Median versus packet loss rate. At packet loss rate of 1%, TCP Median outperforms TCP Tahoe by 10%.

7.10 Mixed Wireless/Wired Simulation

Data is transferred from a wired node to a wireless node through a base station. This setting is more complex than the setting of two simple wireless nodes, because of the management of routing packets from the wired node to the wireless node. To manage this routing, a base station is needed, which acts as a gateway for the wired domains and wireless domains.

Wired-cum-wireless situation is more practical than the simulation of two wireless nodes. It is common to find networks including both wired and wireless connections.

The simulation environment of this section is depicted in Figure 7.17. The Tcl script is in Appendix K.

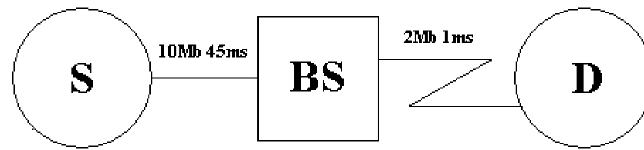

Figure 7.17: Mixed Wireless/Wired Simulation. Packets are sent from S to D via BS.

The source (S) is connected to a wireless base station (BS) through a 10Mb link with 45ms delay. The base station is linked to the destination (D), which is a wireless mobile node, through a 2 Mb lossy channel with delay 1ms. A TCP agent with FTP application, delivers packets from the source to destination. The destination (D) moves at 0.5m/s towards the base station (BS). D is initially 50 meters from BS.

The simulations for TCP Tahoe and TCP Median are performed. The packet loss rate at the wireless link varies from 0.0001% to 10%. The simulation time is 100s.

The throughputs from the simulations versus the packet loss rate is shown in Figure 7.18, where x-axis is in logarithmic scale, and the y-axis is normalized.

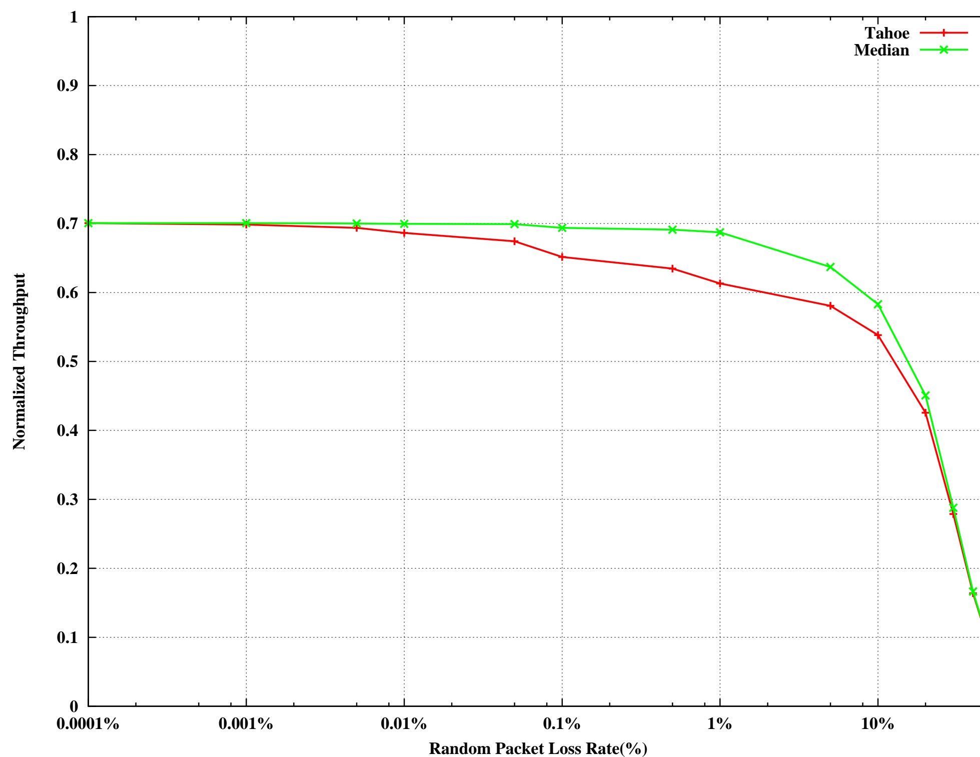

Figure 7.18: Throughputs of TCP Tahoe and TCP Median versus packet loss rate. At packet loss rate of 1%, TCP Median outperforms TCP Tahoe by 12%.

At the starting packet loss rate of 0.0001%, the TCP Tahoe and TCP Median have similar performance. At packet loss rate of 0.01%, TCP Median starts to outperform. TCP Median delivers more throughput than TCP Tahoe by 12%, at a very practical loss rate of 1%. After packet loss rate of 10%, the two throughputs come closer and closer to each other.

7.11 Performance of TCP Median in Wireless Scenarios

In the simulations of two wireless nodes, TCP Median outperforms TCP Tahoe by 10% in throughput, at packet loss rate of 1%. In the mixed wireless/wired simulations, at packet loss rate of 1%, TCP Median outperforms TCP Tahoe by 12% in throughput.

Results from Sections 7.9 and 7.10 indicate that median filter is useful for increasing the throughput. It should be noted that, a median number can only be found from a series of numbers which are sorted in ascending or descending orders, and sorting is a time consuming process. Therefore, great improvement in TCP Median could not be expected. Some improvement found in TCP Median in real-time simulation, is already an acceptable performance.

7.12 Summary

The estimation of RTO based on median filter algorithm is simulated and analyzed in Experiment 1 and Experiment 2. The median filter is a useful filter due to its edge preserving and impulse suppressing characteristics for image processing applications. From the experimental results, it is found that median filter performs well in bursty traffic flows. The median filter can perform better than the weighted-average filter because small RTO is obtained, which is a desirable factor for alleviating traffic congestion. In the mixed wireless/wired simulations, results show that TCP Median outperforms TCP Tahoe by 12% in terms of throughput, at typical packet error rate of 1%.

The median filter is useful but the median number can only be found from a series of numbers which are sorted in ascending or descending orders, and sorting is a time consuming process. Therefore, great improvement in TCP Median could not be expected. Some improvement found in TCP Median in real-time simulation, is already an acceptable performance.

Chapter 8

Conclusions and Further Work

8.1 Introduction

This chapter is intended for highlighting the important points of discoveries found in the project. It consists of the project overview, the project achievements and project publications.

The project overview presents the motivation, objective and literature reviews of this project. The project achievements include the development of a new one-dimensional median filtering, a practical experiment to achieve better RTO, and two simulations to show that median filter outperforms weighted-average filter in terms of goodput and loss percentage. This chapter closes by mentioning some possible further work that might be undertaken.

8.2 Project Overview

This section covers some issues on motivation, objectives and literature review of this project.

8.2.1 Project Motivation

Wireless networks are becoming more and more popular. They allow users to access the Internet with their laptop computers in any locations inside their offices or universities, so wireless networks can provide convenience and mobility to their users (Goldsmith 2005).

In the Internet, reliable data transfer service is provided by Transmission Control Protocol (TCP) (Kurose & Ross 2005). TCP was designed for use in wired network and it does not work so well in wireless networks (Balakrishnan & et al. 1997), because wireless networks have high bit-error rates due to channel fading, noise and interference. Although bit error rates due to channel fading, noise and interference are handled in lower layers (PHY, MAC), they might affect TCP congestion avoidance algorithm.

Packet losses caused by high bit-error rates in wireless networks, are wrongly interpreted by TCP congestion avoidance algorithm as congestion in the link (Balakrishnan & et al. 1997).

TCP was designed for wired networks. How does TCP perform in wireless networks? This question motivated this research project.

8.2.2 Project Objective

The main objective of this research project is to enhance TCP performance in managing congestion over wireless networks, by using median filters. Median filters will be used to estimate the Round Trip Time (RTT). Then, the RTT is used to determine the Retransmission Timeout (RTO).

In 1995, two researchers Paxson and Floyd found that FTP data connections had bursty arrival rate. In addition, the distribution of the number of bytes in each burst has a heavy right tail (Paxson & Floyd 1994).

In statistics, heavy-tailed distributions are probability distributions whose tails are not exponentially bounded (Asmussen 2003). That is, they have heavier tails than the

exponential distribution.

The median filter has been recognized as a useful non-linear filter due to its edge preserving and impulse suppressing characteristics, so it is efficient in removing impulsive noise (Nodes & Gallagher Jr. 1982). The median filter has been applied to many areas of signal processing, particularly in image processing to remove positive and negative impulsive noise which causes white dots and black dots on photographs. Thus it is expected that it can perform well for heavy-tailed distributions.

In this research project, the median filter is applied to network simulations for retransmission timeout estimation, in the presence of bursty traffic flows. Our simulation results show that, the median filter can perform better than the exponential weighted moving average filter, in terms of throughput and loss percentage, i.e. the median filter delivers more throughput and loses fewer packets.

8.2.3 Summary of Literature Review

After describing the motivation and main objective of this project, there is an overview on literature reviews, which are collections of relevant background knowledge necessary to do this project.

Chapter 2 is a literature review on TCP. The major components of TCP, such as slow start, congestion avoidance, fast retransmit and fast recovery are presented here.

Slow start adds another window to the sender's TCP: the congestion window. Each time an ACK is received, the congestion window is increased by one packet. This provides an exponential growth in congestion window during slow start.

Congestion avoidance and slow start require two variables to be maintained for each connection: a congestion window, cwnd, and a slow start threshold, ssthresh.

If cwnd is less than or equal to ssthresh, TCP is in slow start; otherwise TCP is performing congestion avoidance. Slow start continues until TCP reaches ssthresh, and then congestion avoidance takes over.

Congestion avoidance is a linear growth of cwnd, compared to slow start's exponential growth.

Since TCP does not know whether a duplicate ACK is caused by a lost packet or just a reordering of packets, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the packets, there will be only one or two duplicate ACKs before the reordered packet is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a packet has been lost. TCP then performs a retransmission of what appears to be the missing packet, without waiting for a retransmission timer to expire (Stevens 1997). This is called fast retransmit.

After fast retransmit sends what appears to be the missing packet, congestion avoidance, but not slow start is performed. This is called the fast recovery (Jacobson 1990, Stevens 1994). Fast recovery is implemented in TCP Reno only. It is not implemented in TCP Tahoe. The fast retransmit algorithm first appeared in Tahoe release, and it was followed by slow start. The fast recovery algorithm appeared in Reno release (Stevens 1997, Allman & et al. 1999, Floyd & Henderson 1999). To sum up, when timeout occurs, both TCP Reno and TCP Tahoe go back to slow start. When three duplicate ACK's are received, TCP Tahoe performs slow start, but TCP Reno performs congestion avoidance.

Chapter 3 is a literature review on Round Trip Time and Retransmission Timeout (RTO). The evolution of RTO from its original algorithm to Karn/Partridge algorithm, and then to Jacobson/Karels algorithm are described in this chapter.

The reliability of delivery of data is guaranteed in Transmission Control Protocol (TCP) by the following procedure. When a host computer sends a packet, it starts a timer. If the host receives an acknowledgement (ACK) before the timer expires, it sends a new packet and restarts the timer. The time between the starting of the timer and the receiving of ACK is called the Round Trip Time (RTT) (Comer 2006*a*, Peterson & Davie 2000).

If the timer expires before the host receives an ACK, the host will retransmit the same

packet again. The time between the starting of the timer and the expiring of the timer is called Retransmission Timeout (RTO) (Comer 2006*a*, Kurose & Ross 2005, Peterson & Davie 2000).

Chapter 4 describes the median filters. This chapter of literature reviews includes median filters and some of their modifications. These modifications are ranked-order filters, recursive median filters and many others.

The median filter has been recognized as a useful non-linear filter due to its edge preserving and impulse suppressing characteristics, so it is efficient in removing impulsive noise (Nodes & Gallagher Jr. 1982). The median filter has been applied to many areas of signal processing, particularly in image processing to remove positive and negative impulsive noise which causes white dots and black dots on photographs. Thus it is expected that it can perform well for heavy-tailed distributions.

8.3 Project Achievements

In this research project, three achievements have been successfully established. The details of these achievements will be presented in this section.

8.3.1 Development of a New One-Dimensional Median Filtering

The research work in Chapter 6 introduces a fast one-dimensional median filtering algorithm. Median filtering involves sorting an array into ascending order and selecting the element at the middle. At the beginning of this research project, bubble sort was used because of its simplicity in programming logic and its popularity. Bubble sort is easy to program and therefore it is used by many people for sorting. Median filter will be included into TCP for doing simulations. In the simulations of protocols such TCP, efficiency in manipulating bytes and run-time speed are very important. The speed of program execution has to be fast, so a fast sorting algorithm is required. Therefore, other faster sorting algorithms, which are more complicated than bubble sort, such as Shellsort and Quicksort are reviewed. Finally a fast one-dimensional median filtering

algorithm was formulated for TCP, based on the work of T.S. Huang *et al.*

Shellsort, Quicksort and the fast one-dimensional median filtering algorithm are experimentally used to sort the same set of data, and then the timings, spent by the three methods to sort the same set of data, are compared. Experiments show that the fast one-dimensional median filtering algorithm is faster than Quicksort and much faster than Shellsort. The one-dimensional median filtering algorithm could satisfy our need of a fast sorting algorithm.
It is worthwhile to try to program the fast one-dimensional median filtering algorithm to replace the bubble sort. Some improvement in TCP performance is expected after replacement.

8.3.2 Practical Experiment to achieve better RTO

In Chapter 5, a practical experiment is presented. This practical experiment is performed to test the effectiveness of median filter, when it is applied to estimate the RTT.

A simple point-to-point wireless connection is established between a laptop computer and a desktop computer. The distance between the two computers is around one meter, and the transmission rate is 24 Mbps.

Network analyzer software is activated on the desktop computer to monitor and gather information about the network traffic. To simulate a wireless connection disturbance, the laptop computer is carried and moved around, while the desktop computer is streaming a video clip to the laptop computer.

The captured data from the network analyzer is then analyzed. Results show that median filter algorithm performs better than the weighted-average algorithm.

8.3.3 Median Filter Outperforms Weighted-Average in Simulations

Inside NS2, there is a protocol tcp.cc which is written in C++. A median filter of size 5 is included into the tcp.cc source codes to replace the weighted average algorithm.

Two experiments are conducted to show that median filter outperforms weighted-average algorithm. In experiment 1, the traffic source is Exponential.

testing.tcl script is used for testing goodputs and tcp parameters. Firstly, the original tcp.cc in NS2.33 without median filter is used to run testing.tcl. Secondly, the modified tcp.cc with median filter is used to run testing.tcl.

From the experimental results, it is found that the accumulated goodput of weighted-average filter is 740960 which is much lower than the goodput 760240 generated by median filter.
The loss percentage of median filter of size 5 is 1.06%, while that of the weighted-average filter is 1.66%. Therefore, median filter of size 5 has lower loss percentage.

In Experiment 2, the traffic source is Pareto. The goal of this experiment is to show that under bursty traffic flows, median filter can do better than weighted-average filter. The testing.tcl script is used for testing goodputs and tcp parameters. Firstly, the original tcp.cc in NS2.33 without median filter is used to run testing.tcl. Secondly, the modified tcp.cc with median filter is used to run testing.tcl.

Two features in the comparison of goodput are:

1. The total goodput from tcp.cc with median filter of size five is 748560, while that from the original tcp.cc is 711760. Therefore, median filter can deliver more goodput.

2. The loss percentage of median filter of size five is 1.58%, while that of the original tcp.cc is 1.93%. Therefore median filter has lower loss percentage.

8.4 Publications arising from this Research

The following publications were produced as a result of this project.

Auc Fai Chan and John Leis, "Comparison of Weighted-Average and Median Filters for Wireless Retransmission Timeout Estimation," *the International Conference on Signal Processing and Communication Systems, ICSPCS' 2008*, Gold Coast, Queensland, Australia, 15-17th, Dec., 2008.

Auc Fai Chan and John Leis, "Median Filtering Simulation of Bursty Traffic," (in draft) for *the International Conference on Signal Processing and Communication Systems, ICSPCS' 2010*, Gold Coast, Queensland, Australia, 13-15th, Dec., 2010.

8.5 Further Work

Further research in TCP performance might include: median filter in Reno TCP, median filter in Full TCP, and the implementation of one-dimensional median filtering algorithm into the practical experiment of transmitting video clips from desktop computer to laptop computer.

8.5.1 Median Filter in Reno TCP

The Transmission Control Protocol in simulations is Tahoe TCP. The Reno TCP implements fast recovery. Also, upon receiving three duplicated acknowledgements, Reno TCP responds with congestion avoidance algorithm (i.e. by setting threshold equal to half the current window, and then increasing the congestion window linearly).
Median filter can be applied in Reno TCP and perform the simulations.

8.5.2 Median Filter in Full TCP

In our simulation experiments, the Tahoe TCP is a one-way TCP which only sends packets from sources to destinations.

The Full TCP is a new edition of the Transmission Control Protocol family. It differs from the other family members in three aspects:

1. Connections may be established or disconnected by sending SYN/FIN packets.
2. Bidirectional (i.e. two-way) data transfer is supported.
3. Sequence numbers are in bytes instead of in packets.

A typical TCP connection proceeds with an active opener sending a SYN. The passive opener responds with a SYN + ACK. The active opener then responds with an ACK, and some time later, sends the first segment with data (Kasinadhuni 2008).

The Full TCP, by default, sends data on the third segment of an initial three-way handshake (Kasinadhuni 2008). This behavior is different from the typical TCP mentioned above.

The codes for creation of the Full TCP agent are similar to the codes for creation of the other TCP agents, but with a statement $sink listen.

```
set src [new Agent/TCP/FullTcp] # create agent; src stands for source;
set sink [new Agent/TCP/FullTcp] # create agent;
$ns_ attach-agent $node_(s1) $src # bind src to node;
$ns_ attach-agent $node_(k1) $sink # bind sink to node;

$src set fid_ 0 # set flow ID field;
$sink set fid_ 0 # set flow ID field;
$ns_ connect $src $sink # active connection src to sink;
$sink listen # this statement will figure out who its peer is; $src set window_ 100;
```

Median filter is applied in Full TCP and perform simulations to investigate the results.

8.5.3 The Fast One-Dimensional Median Filtering Algorithm

Median filtering involves sorting an array into ascending order and selecting the element at the middle. One of the further developments is to find ways to decrease the steps in sorting in order to improve the computational efficiency. For the practical experiment, it should be easy to replace bubble sort with the fast one-dimensional median filtering algorithm developed in this project. For simulation with NS2, it will be a complicated task to modify the program tcp.cc using the fast one-dimensional medium filtering algorithm.

In both cases, it is worthwhile to try to program the fast one-dimensional median filtering algorithm to replace the bubble sort. Some improvement in TCP performance is expected after replacement.

The main objective, of including the median filter into TCP to replace the weighted-average filter, was met. Another objective, to show that the median filter outperforms the weighted-average filter through simulations, was also successful. The findings and analyses conducted in this project would provide a good basis for further research in the area of TCP over wireless networks.

Bibliography

Allman, M. & et al. (1999), *RFC 2581: TCP Congestion Control.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc2581*

Arce, G. R. & Gallagher Jr., N. C. (1982), 'State Description for the Root-Signal Set of Median Filters', *IEEE Transactions on Acoustics, Speech and Signal Processing* **30**(6), 894 – 902.

Arce, G. R. & Paredes, J. L. (2000), 'Recursive Weighted Median Filters admitting Negative Weights and Their Optimization', *IEEE Transactions on Signal Processing* **48**(3), 768 – 779.

Arce, G. R. & Stevenson, R. L. (1987), 'On the Synthesis of Median Filter Systems', *IEEE Transactions on Circuits and Systems* **34**(4), 420 – 429.

Asmussen, S. (2003), *Applied Probability and Queues*, 2 edn, Springer, United States.

Astola, J. & Kuosmanen, P. (1997), *Fundamental of Nonlinear Filtering*, CRC Press, Boca Raton.

Balakrishnan, H. & et al. (1997), 'A Comparison of Mechanisms for Improving TCP Performance over Wireless Links', *IEEE/ACM Transactions on Networking* **5**(6), 756 – 769.

Berman, K. A. & Paul, J. L. (2005), *Algorithms: Sequential, Parallel and Distributed*, Course Technology Inc, Massachusetts, pp. 49 – 54.

Bovik, A. C. & et al. (1983), 'A Generalization of Median Filtering using Linear Com-

binations of Order Statistics', *IEEE Transactions on Acoustics, Speech and Signal Processing* **31**(6), 1342 – 1350.

Braden, B. & et al. (1998), *RFC 2309: Recommendations on Queue Management and Congestion Avoidance in the Internet.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc2309*

Braden, R. (1989*a*), *RFC 1122: Requirements for Internet Hosts - Communication Layers.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc1122*

Braden, R. (1989*b*), *RFC 1123: Requirements for Internet Hosts - Application and Support.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc1123*

Bradner, S. (1991), *RFC 1242: Benchmarking Terminology for Network Interconnection Devices.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc1242*

Brownrigg, D. R. K. (1984), 'The Weighted Median Filter', *Communications of the ACM* **27**(8), 807 – 818.

Burian, A. & Kuosmanen, P. (2002), 'Tuning the Smoothness of the Recursive Median Filter', *IEEE Transactions on Signal Processing* **50**(7), 1631 – 1639.

Cerf, V. G. & Kahn, R. E. (1974), 'A Protocol for Packet Network Intercommunication', *IEEE Transactions on Communications* **22**(5), 637 – 647.

Clark, D. D. & et al. (2002), 'An Analysis of TCP Processing Overhead', *IEEE Communications Magazine* **40**(5), 94 – 101.

Clarkson, P. M. & Stark, H. (1995), *Signal Processing Methods for Audio, Images and Telecommunications*, Academic Press Ltd, London, pp. 109 – 113.

Comer, D. E. (2006*a*), *Internetworking with TCP/IP: Principles, Protocols, and Architecture Volume I*, 5 edn, Pearson Education Inc, New Jersey, pp. 207 – 210.

Comer, D. E. (2006*b*), *Internetworking with TCP/IP: Principles, Protocols, and Architecture Volume I*, 5 edn, Pearson Education Inc, New Jersey, New Jersey, pp. 204 – 206.

Drossman, H. & Veirs, V. (2002), *Nuclear Power.* viewed on 12 January 2010.
URL: *http://www.coloradocollege.edu/dept/ev/courses/EV212/Block5_2002/fission.htm*

ElAarag, H. (2002), 'Improving TCP Performance over Mobile Networks', *ACM Computing Surveys* **34**(3), 357 – 374.

Fall, K. & Floyd, S. (1996), 'Simulation-based Comparisons of Tahoe, Reno, and SACK TCP', *Computer Communication Review* **26**(3), 5 – 21.

Fielding, R. & et al (1999), *RFC 2616: Hypertext Transfer Protocol-HTTP 1.1.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc2616*

Floyd, S. (1996), Issues of TCP with SACK, Technical report, Penn State College of Information Sciences and Technology.
URL: *http://citeseer.ist.psu.edu/article/floyd96issues.html*

Floyd, S. (1997), *RED: Discussions of Byte and Packet Modes.* viewed on 26 December 2009.
URL: *http://ee.lbl.gov/floyd/REDaveraging.txt*

Floyd, S. (2000), *RFC 2914: Congestion Control Principles.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc2914*

Floyd, S. & Henderson, T. (1999), *RFC 2582: The NewReno Modification to TCP's Fast Recovery Algorithm.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc2582*

Floyd, S. & Jacobson, V. (1993), 'Random Early Detection Gateways for Congestion Avoidance', *IEEE/ACM Transactions on Networking* **1**(4), 397 – 413.

Freund, J. E. & Walpole, R. E. (1987), *Mathematical Statistics*, 4 edn, Prentice Hall,, New Jersey, p. 218.

Gallagher Jr., N. C. & Wise, G. L. (1981), 'A Theoretical Analysis of the Properties of Median Filters', *IEEE Transactions on Acoustics, Speech and Signal Processing* **29**(6), 1136 – 1141.

Gerla, M. & et al. (2001), 'TCP Westwood: congestion window control using bandwidth estimation', *IEEE GLOBECOM* **3**, 1698 – 1702.

Goldsmith, A. (2005), *Overview of Wireless Communication*, Cambridge University Press, New York.

Haavisto, P. & et al. (1991), 'Median Based Idempotent Filters', *Journal of Circuits, Systems, and Computers* **1**(2), 125 – 148.

Hei, X. J. (2001), The Self-Similar Traffic Modeling in the Internet, Technical report, HKUST.
URL: *http://www.ee.ust.hk/~heixj/publication/comp660f/*

Heidemann, J. & et al. (1997), 'Modeling the performance of HTTP over several transport protocols', *IEEE/ACM Transactions on Networking* **5**(5), 616 – 630.

Hoare, C. A. R. (1962), 'Quicksort', *Computer Journal* **5**(1), 10 – 15.

Huang, T. S. & et al. (1979), 'A Fast Two-Dimensional Median Filtering Algorithm', *IEEE Transactions on Acoustics, Speech, and Signal Processing* **27**(1), 13 – 18.

Huston, G. (2000), 'The Future for TCP', *CISCO Internet Protocol Journal* **3**(3), 2 – 27.

Huston, G. (2006), 'Gigabit TCP', *CISCO Internet Protocol Journal* **9**(2), 2 – 26.

ISI (2008*a*), *ns-2 Simulation Tool Home Page.* viewed on 20 March 2008.
URL: *http://www.isi.edu/nsnam/ns/*

ISI (2008*b*), *Running Wireless Simulation in ns.* viewed on 26 December 2009.
URL: *http://www.isi.edu/nsnam/ns/tutorial*

ISI (2008*c*), *The Network Simulator ns-2: Documentation.* viewed on 20 March 2008.
URL: *http://www.isi.edu/nsnam/ns/ns-documentation.html*

Issariyakul, T. & Hossain, E. (2009), *Introduction to Network Simulator NS2*, Springer, New York, p. 38.

Jacobson, V. (1988), 'Congestion Avoidance and Control', *ACM SIGCOMM Computer Communication Review* **18**(4), 314 – 329.

Jacobson, V. (1990), *Modified TCP Congestion Avoidance Algorithm.* viewed on 26 December 2009.

URL: *ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail*

JaJa, J. (2000), 'A Perspective on Quicksort', *Computing in Science & Engineering* **2**(1), 43 – 49.

Justusson, B. I. (1981), *Median filtering: Statistical Properties*, Springer, New York.

Karn, P. & Partridge, C. (1995), 'Improving Round-Trip Time Estimates in Reliable Transport Protocols', *ACM SIGCOMM Computer Communication Review* **25**(1), 66 – 74.

Kasinadhuni, O. (2008), *TCP Validation.* viewed on 13 January 2010.

URL: *http://www.cs.odu.edu/~gpd/msprojects/okasinad.0/TCP_Validation.ppt*

Ko, S. J. & Lee, Y. H. (1991), 'Center Weighted Median Filters and their Applications to Image Enhancement', *IEEE Transactions on Circuits and Systems* **38**(9), 984 – 993.

Kruse, R. L. (1984), *Data Structures and Program Design*, 3 edn, Prentice Hall, United States, pp. 302 – 304.

Kurose, J. F. & Ross, K. W. (2005), *Computer Networking: A Top-Down Approach Featuring the Internet*, 3 edn, Pearson Education Inc, New Jersey, pp. 236 – 239.

Liang, Y. D. (2007), *Introduction to Programming with C++*, Pearson Education Inc, United States, pp. 518 – 533.

Loupos, T. & et al. (1989), 'An Adaptive Weighted Median Filter for Speckle Suppression in Medical Ultrasonic Images', *IEEE Transactions on Circuits and Systems* **36**(1), 129 – 135.

Ma, L. P. & et al. (2003), 'RED Gateway Congestion Control using Median Queue Size Estimates', *IEEE Transactions on Signal Processing* **51**(8), 2149 – 2164.

Ma, L. P. & et al. (2004), 'TCP retransmission timeout algorithm using weighted medians', *IEEE Signal Processing Letters* **11**(6), 569 – 572.

Main, M. & Savitch, W. (2005), *Data Structures and Other Objects Using C++*, 3 edn, Pearson Education Inc, United States, pp. 646 – 655.

McAllister, W. (2009), *Data Structures and Algorithms Using JAVA*, Jones and Bartlett Publishers, United States.

McConnell, J. J. (2001), *Analysis of Algorithms: An Active Learning Approach*, Jones and Barlett Publishers, Massachusetts, pp. 57 – 93.

Meguro, M. & Taguchi, A. (1996), 'Adaptive Weighted Median Filters by using Fussy Techniques', *ISCAS '96, 'Connecting the World'* **2**, 9 – 12.

Miller, R. & Keeler, E. (1997), *Internet Explorer: Virtuoso*, 2 edn, MIS Press Inc, New York, pp. 584 – 585.

Mitra, S. K. & Kaiser, J. F. (1993), *Handbook for Digital Signal Processing*, John Wiley & Sons Inc, New York, pp. 953 – 979.

Newman, D. (1999), *RFC 2647: Benchmarking Terminology for Firewall Performance.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc2647*

Nodes, T. A. & Gallagher Jr., N. C. (1982), 'Median Filters: Some Modifications and Their Properties', *IEEE Transactions on Acoustics, Speech and Signal Processing* **30**(5), 739 – 746.

Oliver, I. (1993), *Programming Classics: Implementing the Worlds Best Algorithms*, Prentice Hall, Australia, pp. 164 – 182.

Patil, B. & et al. (2003), *IP in Wireless Networks*, Prentice Hall, United States, p. 42.

Paxson, V. & Floyd, S. (1994), 'Wide-Area Traffic: The Failure of Poisson Modeling', *ACM SIGCOMM* pp. 257 – 268.

Peterson, L. L. & Davie, B. S. (2000), *Computer Networks: A System Approach*, 4 edn, Morgan Kaufmann Publishers, San Francisco, pp. 204 – 206.

Pilosof, S. & et al. (2003), 'Understanding TCP Fairness over Wireless LAN', *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies* **2**, 863 – 872.

Pitas, I. (2000), *Digital Image Processing Algorithms and Applications*, John Wiley & Sons Inc, New York, pp. 139 – 145.

Pitas, I. & Venetsanopoulos, A. N. (1990), *Nonlinear Digital Filters: Principles and Applications*, Kluwer Academic Publishers, Boston.

Poduri, K. & Nichols, K. (1998), *RFC 2415: Simulation Studies of Increased Initial TCP Window Size.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc2415*

Poirier, J. (2009), *U.S. wireless users satisfied, FCC should help more: GAO.* viewed on 13 January 2010.
URL: *http://www.ibtimes.co.uk/articles/20091210/uwireless-users-satisfied-fcc-should-help-more-gao.htm*

Postel, J. (1981), *RFC 793: Transmission Control Protocol.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc793*

Poularikas, A. D. (1999), *The Handbook of Formulas and Tables for Signal Processing*, CRC Press, Florida, pp. 41–1.

Pratt, W. K. (1978), *Digital Image Processing*, John Wiley & Sons Inc, New York.

Press, W. H. & et al. (1989), *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, pp. 226 – 237.

Reddy, P. C. & Rao, A. (2006), 'Performance Comparison of TCP Congestion Control Algorithm', *IE(I) Journal-CP* **87**.

Reed, W. J. (2003), 'The Pareto Law of Incomes - An Explanation and an Extension', *Physica A: Statistical Mechanics and its Applications* **319**, 469 – 486.

Rhee, I. J. & et al. (2008), 'CUBIC: a new TCP-friendly high-speed TCP variant', *ACM SIGOPS Operating Systems Review* **42**(5), 64 – 74.

Shell, D. (1959), 'A High-Speed Sorting Procedure', *Communications of the ACM* **2**(7), 30 – 32.

Stevens, W. (1997), *RFC 2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms.* viewed on 26 December 2009.
URL: *http://tools.ietf.org/html/rfc2001*

Stevens, W. R. (1994), *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, United States.

Stevens, W. R. & Wright, G. R. (1995), *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, United States.

Stewart, W. (1996), *DARPA Over the Years.* viewed on 25 June 2008.
URL: *http://www.livinginternet.com/i/iidarpa.htm*

Tukey, J. W. (1974), 'Nonlinear (Nonsuperposable) Methods for Smoothing Data', *Congressional Record, EASCON* p. 673.

Weiss, M. A. (2002), *Data Structures and Problem Solving Using Java*, 2 edn, Addison Wesley, United States, pp. 289 – 303.

Willinger, W. & et al. (1997), 'Self-Similarity through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level', *IEEE/ACM Transactions on Networking* **5**(1), 71 – 86.

Xu, L. S. & et al. (2004), 'Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks', *IEEE INFOCOM* **4**, 2514 – 2524.

Yin, L. & et al. (1996), 'Weighted Median Filters: A Tutorial', *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* **43**(3), 157 – 192.

Appendix A

Papers Published in Connection with this Research

Comparison of Weighted-Average and Median Filters for Wireless Retransmission Timeout Estimation

Auc Fai Chan, John Leis
Faculty of Engineering and Surveying
University of Southern Queensland
Toowoomba Queensland 4350

***Abstract*—Estimation of retransmission timeout (RTO) in TCP networks is important in order to maximize throughput. If the RTO is too small, packets will be unnecessarily declared lost, thus resulting in a retransmission. If RTO is too large, applications will wait too long before declaring a packet to be lost, thus reducing throughput. The conventional method due to Karn and Jacobson for estimating RTO is using a smoothing filter [1], [2], and this was later refined to utilize mean deviation measurements [1], [2]. In wireless links, the delay can often be impulsive, and weighted-average filters perform poorly. This paper presents results for mobile wireless transmission, and investigates the use of a median filter for timeout estimation in the presence of competing traffic flows. We demonstrate the applicability of this approach to estimating TCP round-trip times, and UDP packet jitter.**

***Index Terms*—median filter TCP congestion wireless**

I. Introduction

As the Internet expands, the number of TCP segments to be transmitted increases greatly, and consequently, there is congestion. Researchers have designed two different approaches to address the problem of transmission congestion. The first method is to modify the existing TCP/IP, examples of which are sliding windows and Random Early Drop [3]. The second method is to design a new protocol, such as Snoop Protocol [4]. The aim of this paper is to accurately estimate round-trip time (RTT) (which in turn affects RTO). All these will be described in following sections.

II. Transmission Control Protocol/Internet Protocol (TCP/IP)

In early 1970s, the Transmission Control Protocol/Internet Protocol (TCP/IP) was developed by Defense Advanced Research Projects Agency (DARPA), an agency inside the U.S. Department of Defense [5].

TCP resides in the transport layer of the conventional seven-layer Open Systems Interconnection (OSI) model. It is above the IP layer and below the upper layers. TCP is not loaded into router (also called gateway in some books) to support user data transfer. It resides in the host computer.

TCP provides the following services to upper layers:

- Connection-oriented data management;
- Reliable data transfer;
- Resequencing;
- Flow control (sliding window); and
- Precedence (i.e. priority) and security [6].

III. Acknowledgement (ACK) and Sliding Window

TCP uses a technique called positive acknowledgement with retransmission to make a connection reliable.

When a packet of data arrives at the receiver site, the receiver site transmits a positive acknowledgement (ACK) message to the sender site. The sender site will transmit a new packet of data only when the ACK message is received. The sender will start a timer when it transmits a packet of data.

When the timer expires before ACK message is received, the sender site will transmit the packet of data again.

To prevent duplicate packets being sent due to long delays in the network, the sequence number of the packet of data is also sent back with the ACK message.

Sliding window protocol improves TCP by letting the sender side transmit certain number of packets of data without waiting for ACK messages. The window size controls the number of packets of data to be sent before any ACK messages are received.

The success of TCP retransmission algorithm depends on the management of RTT [7].

As the number of computers inside the network grows, the problem of ACK messages and retransmissions becomes complicated, and there will be congestion in the network. Congestion is the focus of this research. The objective of this research is to apply the median filter to improve the performance of TCP to alleviate congestion.

IV. Median Filter

The following example demonstrates the operation of median filter [8].

Example Apply median filter with a window size of 3 to the array x(n), where x(n)=[2 80 6 3]. y[1]=Median[2 2 80]=2 y[2]=Median[2 80 6]=Median[2 6 80]=6 y[3]=Median[80 6 3]=Median[3 6 80]=6 y[4]=Median[6 3 3]=Median[3 3 6]=3 Therefore z(n)=[2 6 6 3] The original signal x(n)=[2 80 6 3], after being filtered by median filter, becomes z(n)=[2 6 6 3]

The properties of median filter are stated below [9], [10]:

1) The filter belongs to the class of Order Statistic (OS) filter.
2) Order Statistic filters operate according to the following steps:
   - The original signal x(n) is sorted in ascending order to produce y(n).
   - The median of y(n) is selected

   x(n) → Rank (sorting number in ascending order) → y(n) → Apply appropriate OS-filter-coefficient → z(n)

   x(n) = original signal
   y(n) = ranked signal
   z(n) = filtered signal
3) For median filter, the filter-coefficient matrix ,a, can be expressed as a = [0..0 1 0..0].
4) The filter has a non-linear frequency response due to its filter-coefficient matrix.
5) Noisy step signal is rounded at the step.
6) The filter will not attenuate noise well, if the noise comes from a noisy ramp signal
7) The filter eliminates impulse noise. However, if the impulse noise is close to an edge, the noise may be removed but the signal edge moves toward the impulse noise (edge jitter).
8) The filter preserves signal edges.

V. Median-filtering results from a wireless point-to-point connection

A simple point-to-point wireless connection is established between a notebook computer and a desktop computer. The distance between two computers is around one metre, and the transmission rate is 24Mbps. A network analyzer software is activated on the desktop computer to monitor and gather the network traffic. To simulate a wireless connection disturbance, the notebook computer is carried and moved around, while the desktop computer is streaming a video clip from the notebook computer.

The captured data from the network analyzer software is exported to an Excel file.

A MATLAB program is written to analyze a series of round-trip-time (RTT) values obtained experimentally from a wireless connection. The series of RTT will later be sent to a median filter for evaluating the performance of median filter on RTT. The median filter window size used for testing are three, five, seven and nine.

From Fig.2 to Fig.5, when the median filter window size increases, those RTT values which are greatly different from the rest are filtered away. However, the RTT values are getting smaller as the median filter window size increases.

From this observation, median filter can improve RTT, which in turn improves the congestion window size management. With consistently stable RTT values, a wireless connection can have fairly constant transfer rate.

VI. Results

According to RFC 793 [11], the RTT can be estimated by

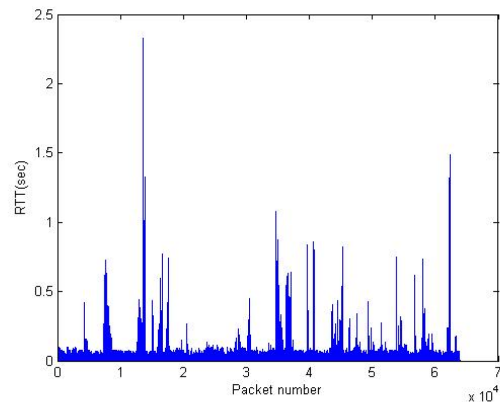

Fig. 1. Unfiltered RTT from a wireless connection.

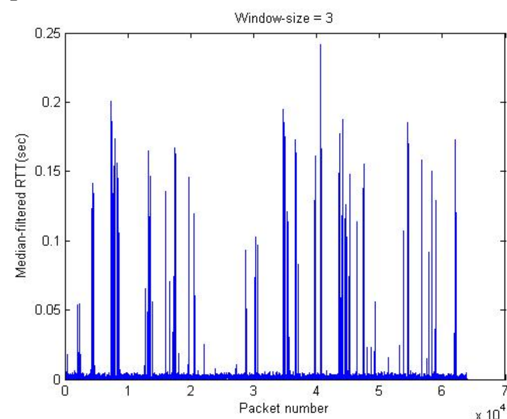

Fig. 2. Median-filtered RTT with window size equal to three.

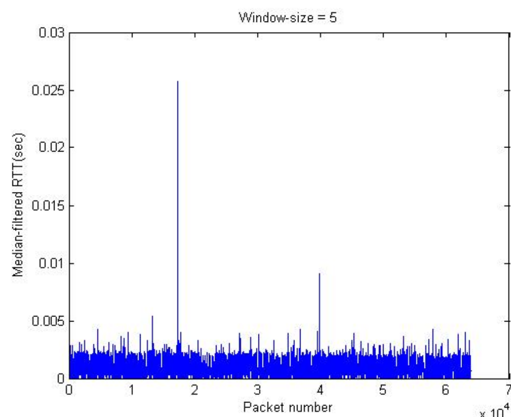

Fig. 3. Median-filtered RTT with window size equal to five.

$$SRTT = \alpha SRTT + (1 - \alpha)M \tag{1}$$

SRTT Smoothed RTT
α Smoothing factor
M Measured RTT

A MATLAB program is written to obtain a series of round-trip-time (RTT) by using (1).

If Fig. 6 is compared with Fig.5, the set of RTT estimated by median filter has less fluctuation than that estimated by weighted-average filter.

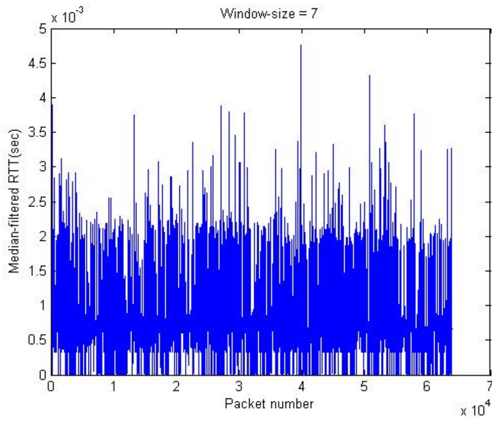

Fig. 4. Median-filtered RTT with window size equal to seven.

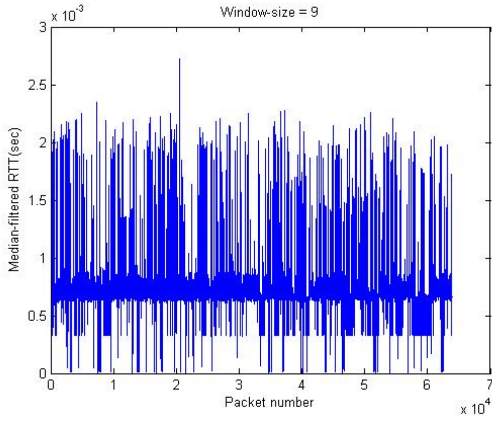

Fig. 5. Median-filtered RTT with window size equal to nine.

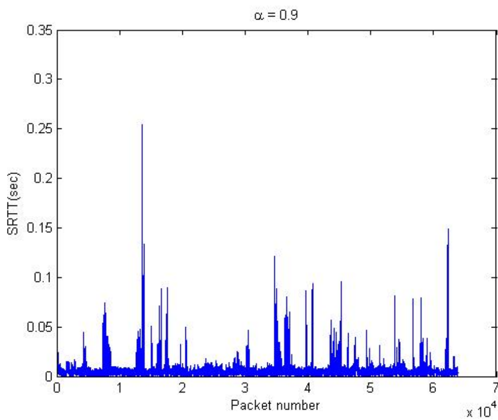

Fig. 6. Smoothed RTT by using (1)

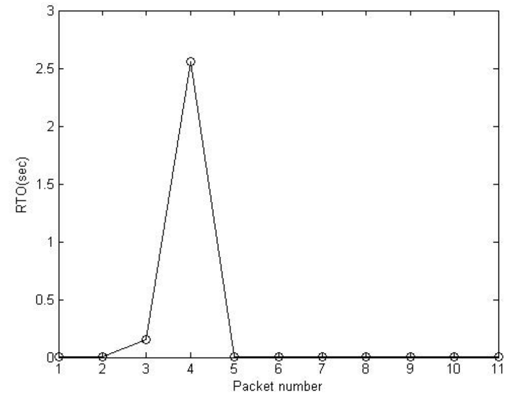

Fig. 7. Unfiltered RTO from a wireless connection.

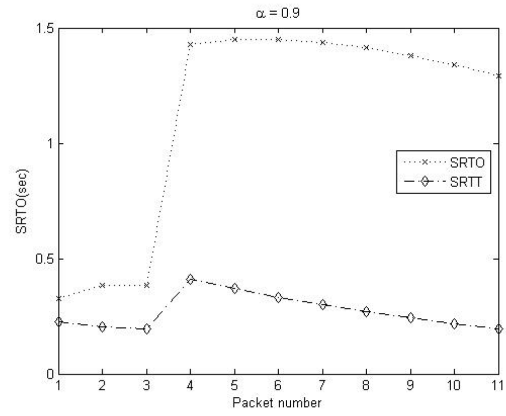

Fig. 8. Smoothed RTO using (2) and (3)

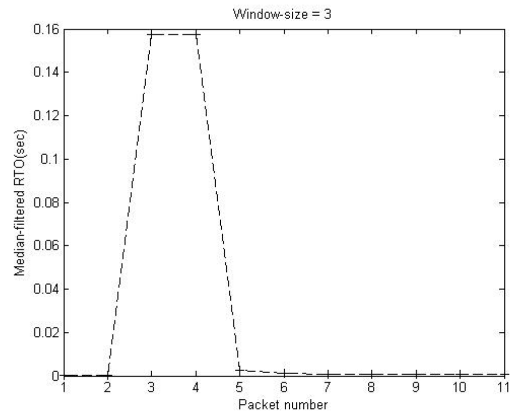

Fig. 9. Median-filtered RTO with window size equals to three.

A further investigation on the performance between median filter and weighted-average filter is carried out by using RTO.For simplicity, the unfiltered RTO and median-filtered RTO are taken to be 1.1 times of their respective RTT values. For the weighted-average filtered RTO, it is found by [12].

$$SRTO = SRTT + 4D \tag{2}$$

SRTO Smoothed RTO
SRTT Smoothed RTT
D Variance

$$D = \alpha D + (1-\alpha)|SRTT - M| \tag{3}$$

D Variance
α Smoothing factor
SRTT Smoothed RTT
M Measured RTT

From the unfiltered RTT result, only one set of samples are chosen for the RTO comparison. Sample number 13655 to 13665 are chosen for testing. This set of samples represents a situation where the transmission rate is bursty. Results for

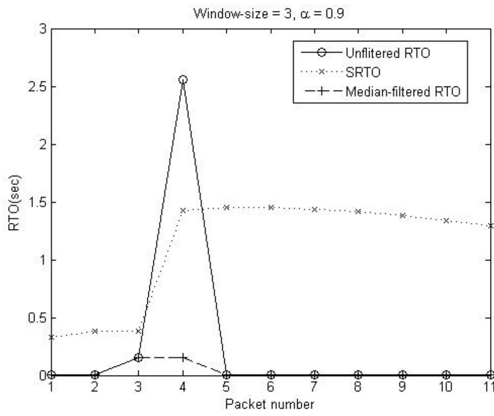

Fig. 10. Different RTOs.

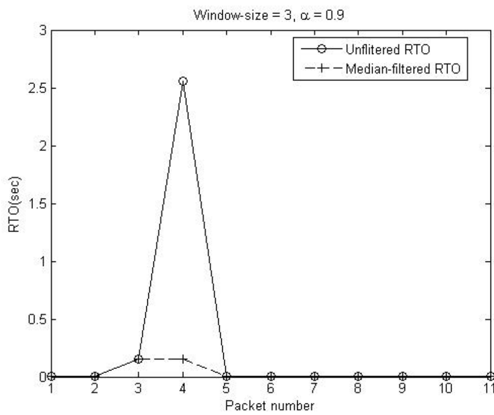

Fig. 11. Unfiltered RTO and median-filtered RTO.

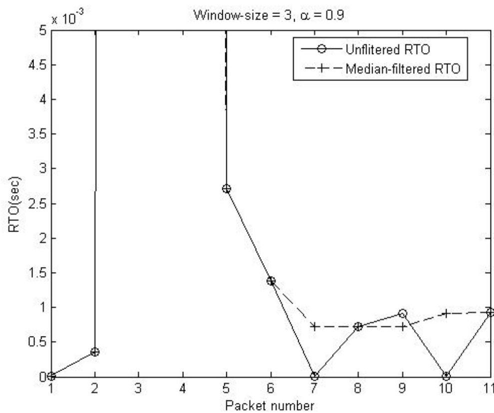

Fig. 12. Unfiltered RTO and median-filtered RTO (Zoom-in view).

sample number 13655 to 13665 are shown in Fig. 7 to Fig. 12.

From Fig. 10 to Fig. 12, weighted-average filter generates RTO timings much higher than those by median filter.

The histogram of RTT values are shown from Fig. 13 to Fig. 20. From Fig. 13, the histogram of the unfiltered RTT does not have a Gaussian distribution. That is, large RTT values exist on the right hand size of Fig. 13. However, these large RTT values are not in huge amount.

In Fig. 14, the histogram of SRTT shows a much gradual change in RTT values, compared with Fig. 13. However, the RTT values are not spread evenly.

TABLE I
EFFECT ON DIFFERENT WINDOW SIZES.

$Window\ Size$	$Mode(in\ millisecond)$	$Outlier = Mode \pm 10\%$
3	1.20	99.5%
5	0.64	51.3%
7	0.69	28.7%
9	0.67	30.8%
10	0.67	31.2%
11	0.67	26.5%

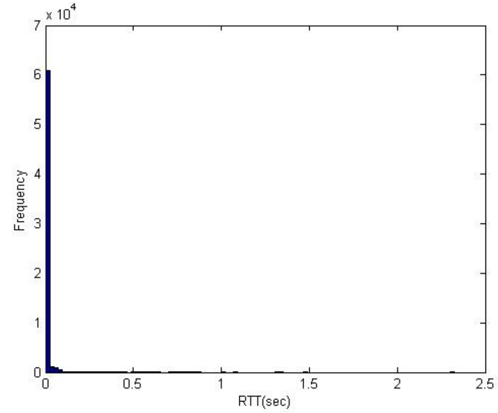

Fig. 13. Histogram of unfiltered RTT from a wireless connection.

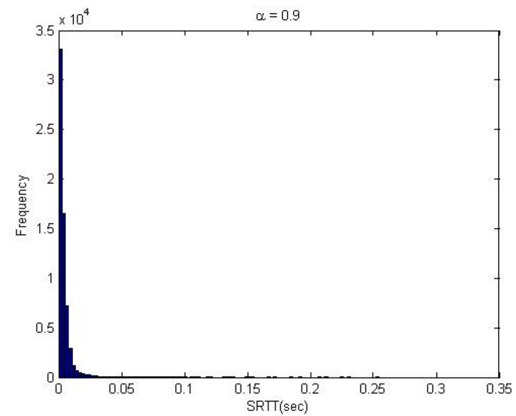

Fig. 14. Histogram of smoothed RTT by using (1)

From Fig. 15 to Fig. 20, pdf of median-filtered RTT with different window sizes are shown. From the window sizes, the pdf distribution can be significantly improved. That is, the large RTT values are being distributed to some smaller values.

In order to compare the results from different window sizes, a table is created. From table 1, mode and outlier from different window size are shown.

The mode is defined as the RTT value that occurs most in that set of filtered RTT values. Outlier is defined as the RTT value that is $\pm 10\%$ of the mode.

From table 1, the number of outliers decreases as the window size increases. This shows that exceptional large RTT values are successfully removed. However, the mode has no significant change after the window size is three.

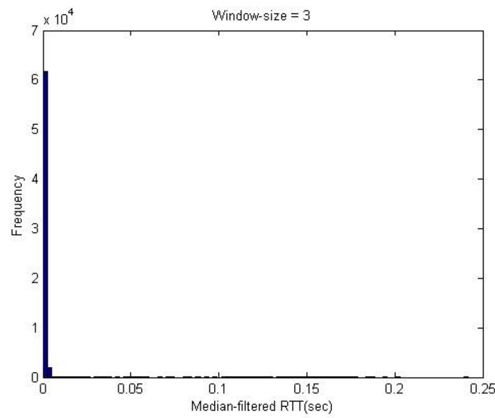

Fig. 15. Histogram of median-filtered RTT with window size equals to three.

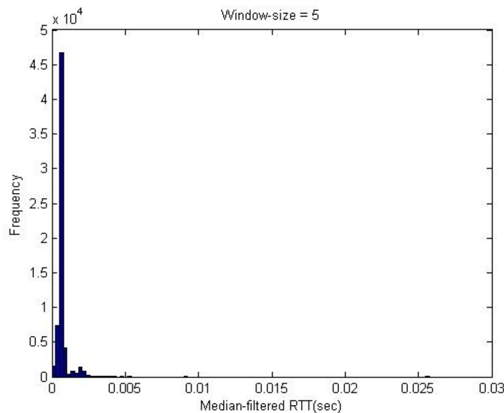

Fig. 16. Histogram of median-filtered RTT with window size equals to five.

From these observations, median filter can perform better than the weighted-average filter because consistent RTT and small RTO are obtained, which are desirable factors for high connection throughput.

VII. Computation and implementation consideration

The experimental results presented in this paper indicates that median filter is useful for RTO estimation yet it is hard to be efficiently implemented for real-time usage. This problem is due to the fact that a median number can only be found from a series of numbers which are sorted in ascending or descending order. Sorting is a time consuming process and the time to sort a series of number is directly proportional to the window size of the median filter [13].

As the bit rate of wireless networks is increasing, the time required for the median computation becomes critical. From the previous section, it is shown that a large window size is desirable. However, this also increases the amount of time in finding the median.

From Fig. 21, the time required for median filtering from 63886 data samples is obtained by using an Intel®Core™2 E6750 with vPro™technology running at 2.66GHz. Two 800MHz-1GB RAMs are used. The software that is being used to rub median filtering is MATLAB 7 (R14). The sorting technique being used is bubble sort.

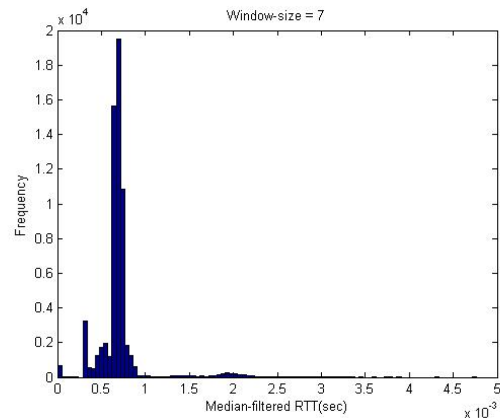

Fig. 17. Histogram of median-filtered RTT with window size equals to seven.

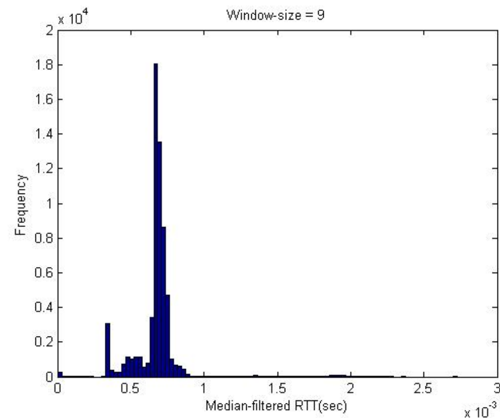

Fig. 18. Histogram of median-filtered RTT with window size equals to nine.

From [13], a fast two-dimensional median filtering algorithm was proposed, and the algorithm was proven to perform better than Quicksort [14]. The two-dimensional algorithm can be adapted for one-dimensional computation.

VIII. Conclusion

From the experimental results, median filter can perform better than the weighted-average filter because consistent RTT and small RTO are obtained, which are desirable factors for high connection throughput.

As the bit rate of wireless networks is increasing, the time required for the median computation becomes critical. From the previous section, it is shown that a large window size is desirable. However, this also increases the amount of time in finding the median. The time required for the median computation is examined for different window sizes, and it is found that the computation time is directly proportional to the window size.

IX. Further Development

The performance of median filter will be modeled by simulation using Network Simulator, version 2 (NS2). Median filter will also be implemented in Linux. Data from simulation and implementation will be analyzed. The fast two-dimensional median filtering algorithm will be adapted for one-dimensional median filtering to improve the computational efficiency.

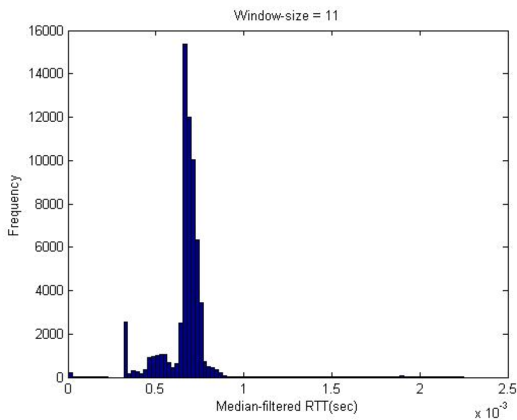

Fig. 19. Histogram of median-filtered RTT with window size equals to eleven.

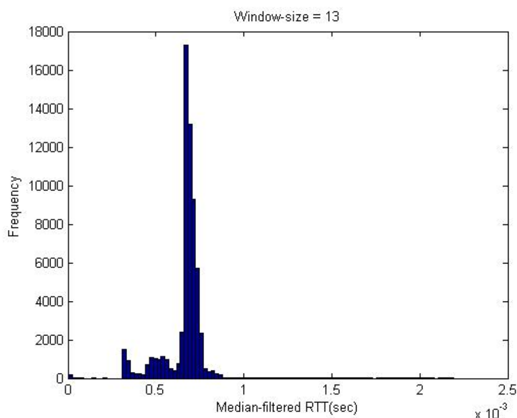

Fig. 20. Histogram of median-filtered RTT with window size equals to thirteen.

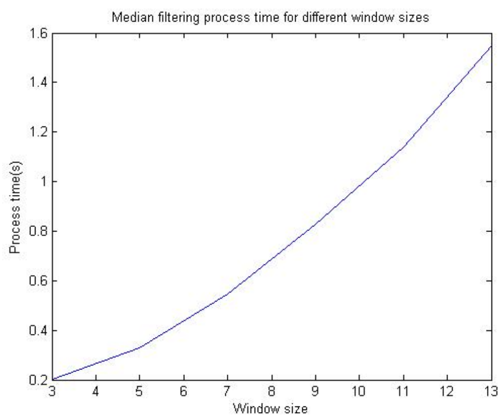

Fig. 21. Computational time against different window sizes.

References

[1] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 314 – 329, Aug 1988.

[2] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," *ACM SIGCOMM Computer Communication Review*, vol. 17, no. 5, pp. 2 – 7, Oct 1987.

[3] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397 – 413, Aug 1993.

[4] University of California at Berkeley. The Berkeley Snoop Protocol. Viewed on 07 May 2008. [Online]. Available: http://nms.lcs.mit.edu/~hari/papers/snoop.html

[5] U. Black, *TCP/IP and Related Protocols*, 2nd ed. McGraw-Hill, New York, 1995, pp. 2–3.

[6] U. D. Black, *Data Communications and Distributed Networks*, 3rd ed. Prentice Hall, New Jersey, 1993, pp. 256–257.

[7] D. E. Comer, *Internetworking with TCP/IP Volume I*, 5th ed. Prentice Hall, New Jersey, 2006, pp. 204–206.

[8] Median filter. Wikipedia, the free encyclopedia. Viewed on 11 May 2008. [Online]. Available: http://en.wikipedia.org/wiki/median_filter

[9] P. M. Clarkson and H. Stark, *Signal processing Methods for Audio, Images and Telecommunications*. Academic Press Ltd., Great Britain, 1995.

[10] A. D. Poularikas, *The Handbook of Formulas and Tables for Signal Processing*. CRC Press, United States, 1999.

[11] RFC 793 (rfc793). Information Sciences Institute (ISI), University of Southern California. Viewed on 10 March 2008. [Online]. Available: http://www.faqs.org/rfcs/rfc793.html

[12] F. Halsall, *Computer Networking and the Internet*, 5th ed. Addision Wesley, United States, 2005, pp. 457–459.

[13] T. Huang, G. Yang, and G. Tang, "A fast two-dimensional median filtering algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 27, no. 1, pp. 13 – 18, Feb 1979.

[14] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10 – 15, 1962.

[15] A. Patil. (2002, 01 Jun) A Snoop for every node. IBM developerWorks. Viewed on 06 May 2008. [Online]. Available: http://www.ibm.com/developerworks/library/wi-snoop/?n-wi-6202

[16] S. G. Glisic, *Advanced Wireless Networks*. John Wiley and Sons Ltd., England, 2006, pp. 279–282.

[17] D. E. Comer, *Internetworking with TCP/IP Volume I*, 5th ed. Prentice Hall, New Jersey, 2006, p. 217.

[18] G. Arce and R. Stevenson, "On the synthesis of median filter systems," *IEEE Transactions on Circuits and Systems*, vol. 34, no. 4, pp. 420 – 429, Apr 1987.

[19] T. Nodes and N. Gallagher Jr., "Median filters: Some modifications and their properties," *IEEE Transactions on Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing]*, vol. 30, no. 5, pp. 739 – 746, Oct 1982.

[20] S. Perreault and P. Hebert, "Median filtering in constant time," *IEEE Transactions on Image Processing*, vol. 16, no. 9, pp. 2389 – 2394, Sep 2007.

[21] Q. Li and D. L. Mills, "Jitter-based delay-boundary prediction of wide-area networks," *IEEE/ACM Transactions on Networking*, vol. 9, no. 5, pp. 578 – 590, Oct 2001.

[22] M. Allman and V. Paxson, "On estimating end-to-end network path properties," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 2, pp. 124 – 151, Apr 2001.

[23] L. P. Ma, K. E. Barner, and G. R. Arce, "TCP retransmission timeout algorithm using weighted medians," *IEEE/ACM Transactions on Networking*, vol. 11, no. 6, pp. 569 – 572, Jun 2004.

[24] L. P. Ma, G. R. Arce, and K. E. Barner, "Statistical analysis of TCP's retransmission timeout algorithm," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 383 – 396, Apr 2006.

[25] D. Loguinov and H. Radha, "Retransmission schemes for streaming internet multimedia: evaluation model and performance analysis," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 2, pp. 70 – 83, Apr 2002.

[26] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "Tcp westwood: Bandwidth estimation for enhanced transport over wireless links," *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pp. 287 – 297, Jul 2001.

[27] DARPA Over the Years, August 1997. DARPA / ARPA –Defense / Advanced Research Project Agency. Viewed on 25 June 2008. [Online]. Available: http://www.livinginternet.com/i/ii_darpa.htm

[28] S. Vangala and M. A. Labrador. Performance of TCP over Wireless Networks with the Snoop Protocol. Department of Computer Science and Engineering, University of South Florida. Viewed on 25 June 2008. [Online]. Available: http://csdl2.computer.org/comp/proceedings/lcn/2002/1591/00/15910600.pdf

Median Filtering Simulation of Bursty Traffic

Auc Fai Chan, John Leis
Faculty of Engineering and Surveying
University of Southern Queensland
Toowoomba Queensland 4350

***Abstract*—The estimation of Retransmission Timeout (RTO) in Transmission Control Protocol (TCP) affects the throughput of the transmission link. If the RTO is just a little larger than the Round Trip Time (RTT), retransmissions will occur too often, and this increases congestion in the transmission link [1]. If the RTO is much larger than the RTT, the response to retransmit when a packet is lost will be too slow, and this will decrease the throughput in the transmission link. Currently, Jacobson's Algorithm [2] for estimation of RTO is implemented in TCP. He uses an Exponential Weighted Moving Average (EWMA) filter to estimate RTT and then determines RTO. The EWMA filter is good if RTT follows a Gaussian distribution. In reality, traffic on the Internet is bursty and follows a heavy-tailed distribution. Median filter has been recognized as a useful filter due to its edge preserving characteristic for image processing applications, and it performs well for heavy-tailed distributions. Thus the median filter is efficient for removing impulsive noise [3], [4]. In this paper, we demonstrate the applicability of the median filter in our simulations for timeout estimation in the presence of bursty traffic flows.**

***Index Terms*—median filter TCP congestion Pareto exponential**

I. Introduction

THE TCP specification that specifies the first original algorithm is RFC 793, which was published in 1981 [5]. The first original algorithm, after several years of implementation on the Internet, was found to have ambiguity in the measurement of Round Trip Time (RTT) that was used when the packets were retransmitted. In 1987, Phil Karn and Craig Partridge found that TCP was suffering from a problem they called retransmission ambiguity. They presented a novel and effective method to clarify this retransmission ambiguity problem, in their paper, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", [6]. Their method could mitigate congestion in Internet, but, it was Jacobson and Karels, who later addressed congestion on Internet more effectively. In October, 1986, Van Jacobson reported the first Internet congestion collapse [2]. The data throughput from Lawrence Berkeley Laboratory to University of California Berkeley, which are 400 yards apart, dropped from 32 Kbps to 40 bps. In 1988, Van Jacobson published his paper, "Congestion Avoidance and Control", [2]. Jacobson developed the congestion avoidance mechanisms that are now incorporated by almost all TCP implementations. All the TCP implementations are based on the four intertwined congestion control algorithm, slow start, congestion avoidance, fast retransmit and fast recovery. Currently, Jacobson Algorithm for estimation of Retransmission Timeout (RTO) is implemented in TCP. Jacobson Algorithm uses Exponential Weighted Moving Average (EWMA) filters to estimate RTT and then determine RTO. EWMA filter is good for signals that exhibit Gaussian distribution. In reality, RTT's are often impulsive, and follow the heavy tailed distribution. Using EWMA Filter to estimate heavy tailed distribution is inadequate. In digital image processing, Median filter is known to eliminate positive and negative impulses [3], [4]. Therefore, median filter can do better when dealing with heavy-tailed distribution. Thus we employ Median Filter in our simulations for timeout estimation in the presence of bursty traffic flows.

II. Problems with RTT Estimation

Jacobson Algorithm includes firstly finding the difference between the sample RTT and the old Estimated RTT.

$$Difference = M_{i-1} - A_{i-1} \tag{1}$$

A_{i-1} Old Estimated RTT
M_{i-1} Sample RTT [7], [8]

Secondly, find the new Estimated RTT

$$A_i = A_{i+1} + (G_1)(Difference) \tag{2}$$

$$= A_{i+1} + G_1(M_{i-1} - A_{i-1}) \tag{3}$$

A_i New Estimated RTT

G_1 is a constant, which is typically equal to 1/8 [7]–[9]. The last expression 3 above states that we make a new prediction (A_i), based on the old prediction (A_{i-1}), plus a fraction $(G_1 = 1/8)$ of the prediction error $(M_{i-1} - A_{i-1})$.

The prediction error is the sum of two components:

1) Error due to noise in the measurement, which is random and unpredictable, such as fluctuations in competing traffic. We denote this part by E_r.
2) Error due to a poor choice of A_{i-1} . We denote this part by E_e.

Then $A_i = A_{i-1} + G_1E_r + G_1E_e$

The G_1E_e term moves A_i in the correct direction while G_1E_r moves A_i in a random direction [2]. If A_i follows a Gaussian distribution (also called normal distribution),G_1E_r will cancel one another after a number of samples, and A_i will converge to the correct value.

However, in 1995, two researchers Paxson and Floyd found that FTP data connections had bursty arrival rate. In addition, the distribution of the number of bytes in each burst has a heavy right tail [10].

In statistic, heavy-tailed distributions are probability distributions whose tails are not exponentially bounded [11]. That is, they have heavier tails than the exponential distribution. One of the simplest heavy-tailed distributions is the Pareto distribution [12]. In the next section, we introduce the probability density functions (pdf) of exponential distribution and Pareto distribution and the similarity between them.

III. Probability Density Functions of Exponential and Pareto Distribution

One of the examples of exponential distribution is nucleus fission. Let α be the probability that a radioactive nucleus will decay in a specified time interval, for example one second. This decay probability, α, is a constant. That is, it does not change over time. Let N(x) be the number of nuclei under consideration at time x. Then the number of decays per second (dN/dx) will be equal to this probability of decay times the number of undecayed nuclei (-αN). The minus sign means that the number of undecayed nuclei, N, is going to decrease as nucleus fission goes on [13]. Hence,

$$\frac{dN(x)}{dx} = -\alpha N(x) \tag{4}$$

We solve this differential equation in 4, and let N = N_0 at time x = 0.

$$N(x) = \frac{N_0}{e^{\alpha x}} \tag{5}$$

In equation 5, N(x) is an exponential distribution. We plot N(x) in Fig.1.

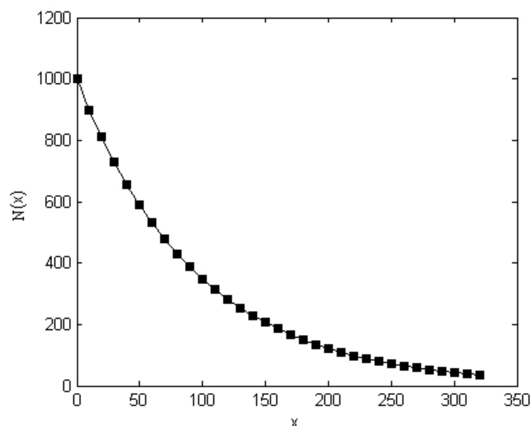

Fig. 1: Nuclear fission follows an exponential distribution [13].

In order to see the similarity between exponential distribution and Pareto distribution, we simplify equation 5 as follows:

- e = 2.7 (to one place of decimal)
- Let α = 1 (Probability can be 1)

Then, $N(x) = N_0/2.7^x$ where N_0, the number of nuclei at time x = 0, is a constant. Later, we show that one example of Pareto distribution is $f(x) = Constant/x^{2.7}$.

Now we introduce Pareto distribution. The definition of Pareto distribution [14] is:

$$f(x) = \begin{cases} \frac{\alpha k^\alpha}{x^{\alpha+1}} & \text{for x >k} \\ \\ 0 & \text{elsewhere} \end{cases} \tag{6}$$

Let us condsider the simple case where k = 1 and α = 1.7. We have

$$f(x) = \begin{cases} \frac{1.7}{x^{2.7}} & \text{for x >1} \\ \\ 0 & \text{elsewhere} \end{cases} \tag{7}$$

We see that there is similarity between exponential distribution and Pareto distribution. The parameter k specifies the minimum value of x. The parameter α determines the shape of f(x). In the simulations of Pareto traffic, we set the shape of Pareto, so we are actually setting the value of α. We plot exponential distribution and Pareto distribution in Fig.2 for comparison.

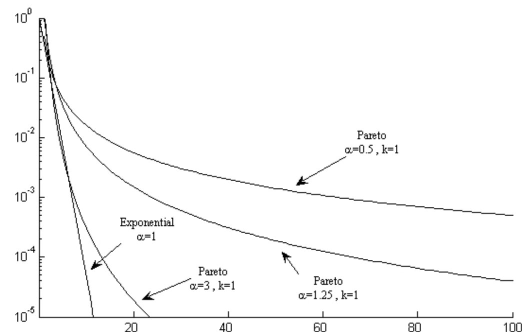

Fig. 2: Exponential and Pareto probability density functions [15].

In Fig.2, we see that Pareto distribution decreases much more slowly than exponential distribution, or we say that, Pareto distribution has a heavier tail than exponential distribution.

In 1997, Willinger *et al.* [16] generated an Ethernet traffic by superposition of many Pareto traffic sources, setting them ON and OFF. During ON period, the source transmits a burst of packets. During OFF period, the source is in idle and no packets are transmitted. We also set ON, OFF and shape of the Pareto traffic in our simulations.

Weighted-average is good if RTT follows a Gaussian distribution. In reality, traffic on Internet is bursty and follows a heavy-tailed distribution. Median filter is known in image processing to eliminate salt and pepper noise which is positive and negative impulses. Therefore, median filter can do better when dealing with heavy-tailed distribution.

IV. Simulation Environment

In both Case 1 and Case 2, dumbell topology is used with 6 pairs of nodes. All the links have the same bandwidth. The bottleneck link is 1Mb and the queue type is DropTail. For simplicity, we consider only TCP traffic which is a one-way flow between source nodes and destination nodes.

Case 1: The traffic source is exponential. There are 6 sources and 6 destinations. s1 sends packets to d1; s2 sends packets to d2, and so on. s1 through s6 are TCP agents. d1 through d6

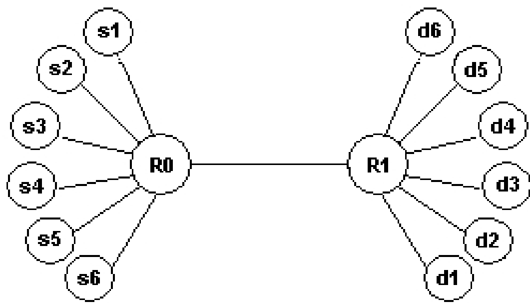

Fig. 3: Topology for Simulation Environment Case 1 and 2.

are TCP-sink agents. All links from sources to R0 are duplex links, with bandwidth of 10 Megabits, delay of 10 ms, and DropTail queues. R0R1 is a duplex link, with bandwidth of 1 Megabits, delay of 100 ms and Drop Tail queues. Queue size is set to 40.

Case 2: The traffic source is Pareto. The bandwidth and queue type are the same as Case 1. The traffic shape is 2.5 and the bursty time is 1 second.

V. Simulation Results

Case 1 : Exponential traffic After the simulation, the accumulative goodputs are plotted in Fig.4.

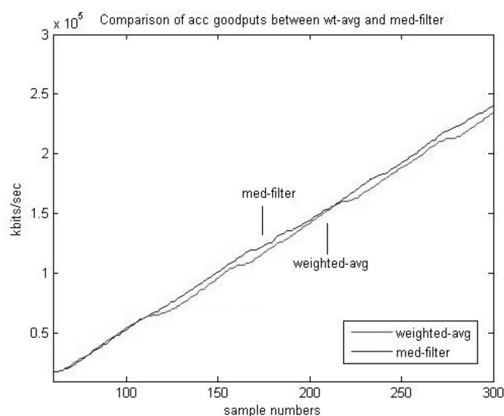

Fig. 4: Comparison of accumulative goodputs in exponential traffic.

Table I: Exponential Traffic Analysis.

	$throughput$	$goodput$	$loss\%$
Original TCP	753440	740960	1.66%
Median filter of size 5	760240	752160	1.06%

From Table I, two improvements made by median filter of size 5 are listed below:

1) The total goodputs from median filter of size 5 is 752160 while that from the original TCP is 740960. Therefore, median filter of size 5 can deliver more goodputs.
2) The loss percentage of median filter of size 5 is 1.06%, while that of the original TCP is 1.66%. Therefore, median filter of size 5 has lower loss percentage.

Next, RTT and RTO are plotted in the same figure for comparison. Fig.5 shows RTT and RTO from the original TCP, while Fig.6 shows RTT and RTO from median filter of size 5.

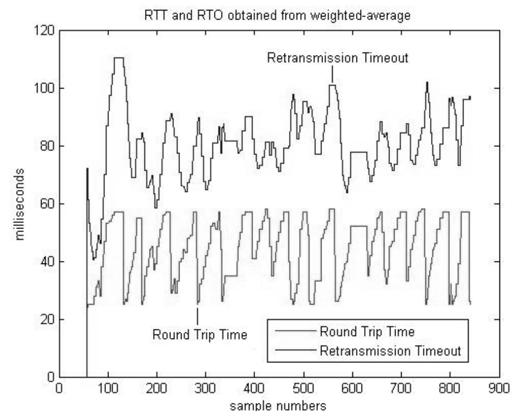

Fig. 5: RTT and RTO obtained from weighted-average in exponential traffic.

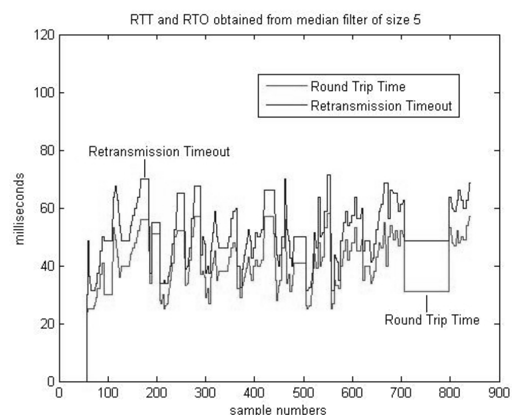

Fig. 6: RTT and RTO obtained from median filter of size 5 in exponential traffic.

In Fig.5, the RTO obtained from weighted average algorithm is much larger than RTT so the response to retransmit when a segment is lost will be too slow; and this will decrease the throughput of the transmission path [1].

If RTO is just a little larger than RTT, retransmission will be too often, and that increases congestion in the transmission path [1]. A desirable RTO is one which is close to, and always larger than the RTT. In Fig.6, the RTO estimated by median filter is possible to increase throughput and at the same time decrease congestion in the transmission path.

In order to have a good understanding of median filter working in exponential traffic, the RTO values obtained from median filters of size 7 and size 9 are shown in Fig.7 and Fig.8.

The loss percentage in the transmission is calculated by the following formula: loss percentage = (throughput - goodput) / throughput x 100%

The loss percentages of median filters of size 5, size 7, size 9 and the original TCP are shown in Table II.

The conclusion is that, all loss percentages of median filters

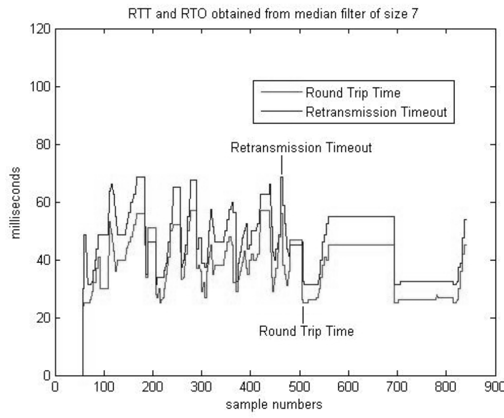

Fig. 7: RTT and RTO obtained from median filter of size 7 in exponential traffic.

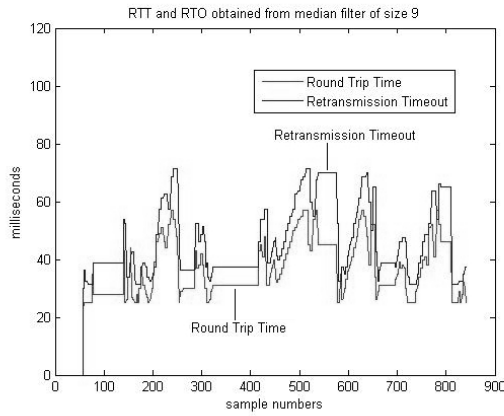

Fig. 8: RTT and RTO obtained from median filter of size 9 in exponential traffic.

Table II: Comparison of loss percentages in exponential traffic.

	$throughput$	$goodput$	$loss\%$
median filter size 5	760240	752160	1.06%
median filter size 7	653280	647360	0.91%
median filter size 9	728720	722480	0.86%
original TCP	753440	740960	1.66%

are lower than that of the original TCP.

Case 2 : Pareto traffic Similar to Case 1, the accumulative goodputs are plotted in Fig.9. In Fig.10 and Fig.11, RTT and RTO are plotted in the same figure for comparison.

Table III: Pareto Traffic Analysis.

	$throughput$	$goodput$	$loss\%$
Original TCP	725760	711760	1.93%
Median filter of size 5	760560	748560	1.58%

From Table 3, two features in the comparison of goodputs are:

1) The goodput from median filter of size five is 748560, while that from the original TCP is 711760. Therefore, median filter can deliver more goodputs.
2) The loss percentage of median filter of size five is 1.58%, while that of the original TCP is 1.93%. Therefore median filter has lower loss percentage.

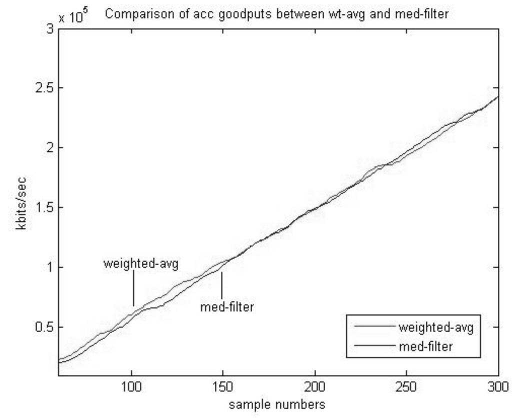

Fig. 9: Comparison of accumulative goodputs in Pareto traffic.

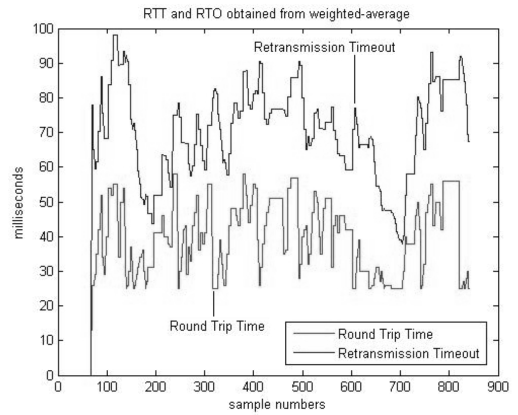

Fig. 10: RTT and RTO obtained from weighted-average in Pareto traffic.

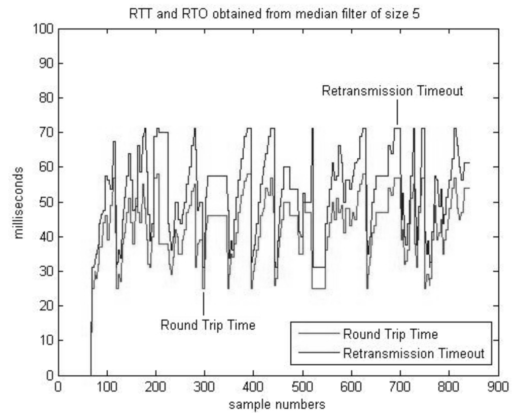

Fig. 11: RTT and RTO obtained from median filter of size 5 in Pareto traffic.

Fig.11 shows that the RTO from median filter of size 5, is better than the RTO in Fig.10 using weighted average estimation.

The RTOs obtained from median filters of size 7 and size 9 are shown in Fig.12 and Fig.13.

The loss percentages of median filters of size 5, size 7 and size 9 are shown in Table IV.

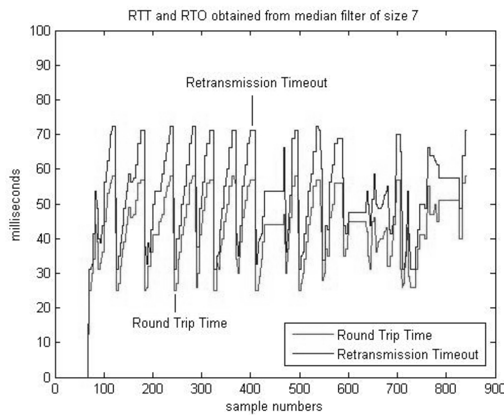

Fig. 12: RTT and RTO obtained from median filter of size 7 in Pareto traffic.

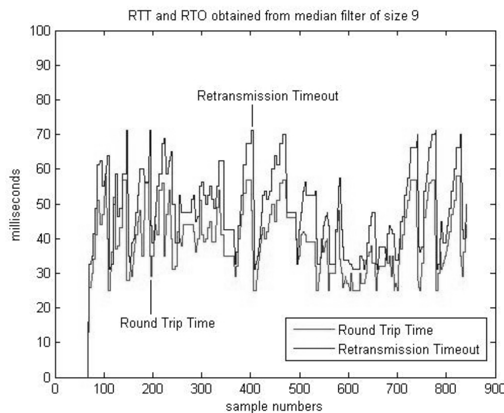

Fig. 13: RTT and RTO obtained from median filter of size 9 in Pareto traffic.

Table IV: Comparison of loss percentages in Pareto traffic.

	$throughput$	$goodput$	$loss\%$
median filter size 5	760560	748560	1.58%
median filter size 7	676640	667520	1.35%
median filter size 9	756080	744400	1.54%

The conclusion for loss percentages is that, all loss percentages from median filters of sizes 5, 7, 9 are lower than the loss percentage from the weighted average estimation, which is 1.93%.

When comparing goodputs among median filters of sizes 5, 7, and 9, the goodput from median filter of size 5 is the highest.

VI. Conclusion

The estimation of RTO based on median filter algorithm is proposed and analyzed in this paper. Median filter is a useful filter due to its preserving characteristic for image processing applications. From the experimental results, we find that it performs well for heavy-tailed distributions to eliminate impulses in bursty traffic flows. As a result, median filter can perform better than the weighted-average filter because consistent RTT and small RTO are obtained, which are desirable factors for high connection throughput to alleviate traffic congestion.

References

[1] D. E. Comer, *Internetworking with TCP/IP: Principles, Protocols, and Architecture Volume I*, 5th ed. Pearson Education Inc, New Jersey, 2006, pp. 204 – 210.

[2] V. Jacobson, “Congestion avoidance and control,” *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 314 – 329, Aug 1988.

[3] J. Astola and P. Kuosmanen, *Fundamental of Nonlinear Filtering*. CRC Press, United States, 1997.

[4] I. Pitas and A. N. Venetsanopoulos, *Nonlinear Digital Filters: Principles and Applications*. Kluwer Academic Publishers, Boca Raton, 1990.

[5] J. Postel. RFC793: Transmission Control Protocol. Viewed on 07 May 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[6] P. Karn and C. Partridge, “Improving round-trip time estimates in reliable transport protocols,” *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 1, pp. 67 – 74, Jan 1995.

[7] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, 3rd ed. Pearson Education Inc, New Jersey, 2005, pp. 236 – 239.

[8] L. L. Peterson and B. S. Davie, *Computer Networks: A System Approach*, 4th ed. Morgan Kaufmann Publishers, San Francisco, 2000, pp. 204 – 206.

[9] L. P. Ma, K. E. Barner, and G. R. Arce, “TCP retransmission timeout algorithm using weighted medians,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 6, pp. 569 – 572, Jun 2004.

[10] V. Paxson and S. Floyd, “Wide area traffic: The failure of poisson modeling,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 6, pp. 569 – 572, Jun 2004.

[11] S. Asmussen, *Applied Probability and Queues*, 2nd ed. Springer, Berlin, 2003.

[12] Pareto distribution. Answers.com. Viewed on 26 December 2009. [Online]. Available: http://www.answers.com/topic/pareto-distribution

[13] D. H and V. V. Nuclear Power. Viewed on 12 January 2010. [Online]. Available: http://www.coloradocollege.edu/dept/ev/courses/EV212/Block5_2002/fission.htm

[14] J. E. Freund and R. E. Walpole, *Mathematical Statistics*, 4th ed. Prentice Hall, New Jersey, 1987, p. 218.

[15] X. J. Hei. Heavy-Tailed Distributions. Viewed on 12 January 2010. [Online]. Available: http://www.ee.ust.hk/~heixj/publication/comp660f/node4.html

[16] W. Winginger, M. S. Taqqu, R. Sherman, and D. Wilson, “Self-similarity through high variability: Statistical analysis of ethernet lan traffic at the source level,” *IEEE/ACM Transactions on Networking*, vol. 5, no. 1, pp. 71 – 86, Feb 1997.

Appendix B

Partition Function Example

The following example illustrates the Partition function.

A = [a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8]

where a_1= 6, a_2 = 2, a_3 = 4, a_4 = 7, a_5 = 1, a_6 = 3, a_7 = 8, a_8 = 5.

Therefore A can be written as:

A = [6, 2, 4, 7, 1, 3, 8, 5]

The following steps will show how the Partition function rearranges the Array A.

Step 1: Start

Line 1: PivotValue = list[first] = a_1 = 6

Line 2: PivotPoint = first = subscript of a_1 = 1

Line 3: for index = 1+1 to 8 do

Step 2: for index = 2 to 8 do

Line 4: if list[index] $<$ PivotValue then

if list[2] $<$ 6 then

if a_2 = 2 is less than 6 then

Line 5: PivotPoint = PivotPoint + 1 = 1 + 1 = 2

Line 6: Swop(list[PivotPoint], list[index])

Swop(list[2], list[2])

/*After swopping a_2 = 2*/

Step 3: for index = 3 to 8 do

Line 4: if list[index] $<$ PivotValue then

if list[3] $<$ 6 then

if $4 < 6$ then

Line 5: PivotPoint = PivotPoint + 1

PivotPoint = 2 + 1 = 3

Line 6: Swop(list[PivotPoint], list[index])

Swop(list[3], list[3])

/*After swopping $a_3 = 4$*/

Step 4: for index = 4 to 8 do

Line 4: if list[index] $<$ PivotValue then

if list[4] $<$ 6 then

if $7 < 6$ then

Step 5: for index = 5 to 8 do

Line 4: if list[5] $<$ 6 then

if $a_5 = 1$ is less than 6 then

Line 5: PivotPoint = PivotPoint + 1

PivotPoint = 3 + 1 = 4

Line 6: Swop(list[PivotPoint], list[index])

Swop(list[4], list[5])

/*After swopping $a_4 = 1$, $a_5 = 7$*/

/*Before swopping, A = [6, 2, 4, 7, 1, 3, 8, 5]*/

/*After swopping, A = [6, 2, 4, 1, 7, 3, 8, 5]*/

Step 6: for index = 6 to 8 do

Line 4: if list[6] $<$ 6 then

if $a_6 = 3$ is less than 6 then

Line 5: PivotPoint = 4 + 1 = 5

Line 6: Swop(list[5], list[6])

/*After swopping, A = [6, 2, 4, 1, 3, 7, 8, 5]*/

Step 7: for index = 7 to 8 do

Line 4: if list[7] $<$ 6 then

if $8 < 6$ then

Step 8: for index = 8 to 8 do

Line 4: if list[8] < 6 then

if 5 < 6 then

Line 5: PivotPoint = 5 + 1 = 6

Line 6: Swop(list[6], list[8])

/*After swopping, A = [6, 2, 4, 1, 3, 5, 8, 7]*/

Step 9:

Line 7: end if

Line 8: end for

Line 9: Swop(list[first], list[PivotPoint])

Swop(list[1], list[6])

/*After swopping, A = [5, 2, 4, 1, 3, 6, 8, 7]*/

Line 10: return Pivotpoint

/* Pivotpoint = 6 is returned*/

The following steps summarize how the Partition function rearranges Array A.

Step 1. Start

A = [6, 2, 4, 7, 1, 3, 8, 5]

Step 2. Index = 2, PivotPoint = 2

A = [6, 2, 4, 7, 1, 3, 8, 5]

Step 3. Index = 3, PivotPoint = 3

A = [6, 2, 4, 7, 1, 3, 8, 5]

Step 4. Index = 4, PivotPoint = 3

A = [6, 2, 4, 7, 1, 3, 8, 5]

Step 5. Index = 5, PivotPoint = 4

A = [6, 2, 4, 1, 7, 3, 8, 5]

Step 6. Index = 6, PivotPoint = 5

A = [6, 2, 4, 1, 3, 7, 8, 5]

Step 7. Index = 7, PivotPoint = 5

A = [6, 2, 4, 1, 3, 7, 8, 5]

Step 8. Index = 8, PivotPoint = 6

A = [6, 2, 4, 1, 3, 5, 8, 7]

Step 9. end if; end for; swop (list[1], list[6])

A = [5, 2, 4, 1, 3, 6, 8, 7]

The following example illustrates how Quicksort is recursively applied to Array A.

[5, 2, 4, 1, 3] + [6] + [8, 7]

Apply Quicksort to the left of PivotValue [6], i.e. [5, 2, 4, 1, 3] , and 5 is the PivotValue for this sub-array.

[2, 4, 1, 3] + [5] + [ ]

Apply Quicksort to the left of PivotValue [5], i.e. [2, 4, 1, 3] , and 2 is the PivotValue for this sub-array.

[1] + [2] + [4, 3]

Apply Quicksort to the right of PivotValue [2], i.e. [4, 3] , and 4 is the PivotValue for this sub-array.

[3] + [4] + [ ]

The left of PivotValue [6], after Quicksort is:

[1] [2] [3] [4] [5]

Apply Quicksort to the right of PivotValue [6], i.e. [8, 7] , and 8 is the PivotValue for this sub-array.

[8, 7]

[7] + [8] + [ ]

Array A after Quicksort is:

[1] [2] [3] [4] [5] [6] [7] [8]

It has just been shown that, after applying Partition function to

A = [6, 2, 4, 7, 1, 3, 8, 5], the correct outcome is

A = [5, 2, 4, 1, 3, 6, 8, 7]

Sometimes, for the purpose of illustrations of programs, it is simpler to write the outcome as A = [2, 4, 1, 3, 5, 6, 7, 8] by simply taking 6 as the pivot, putting those elements less than 6 to the left, and those larger than 6 to the right. Let us call this outcome as conceptual outcome.

Appendix C

Quicksort Algorithm Example

The following example illustrates the Quicksort algorithm, which is repeated here, with line number added for convenient reference.

Line 1:	Quicksort(list, first, last)
Line 2:	if first < last then
Line 3:	pivot = Partition(list, first, last)
Line 4:	Quicksort(list, first, pivot-1)
Line 5:	Quicksort(list, pivot+1, last)
Line 6:	end if

Input A = [6, 2, 4, 7, 1, 3, 8, 5]

Line 1: list is 6, 2, 4, 7, 1, 3, 8, 5

first is 1 (first is the index of the first element)

last is 8 (last is the index of the last element)

Line 2: if $1 < 8$ then

Line 3: pivot = Partition(list, first, last)

After calling Partition, the correct outcome is

A = [5, 2, 4, 1, 3, 6, 8, 7] and PivotPoint = 6 is returned.

Therefore pivot = 6

Line 4: Quicksort(list, first, pivot-1)

list is 5, 2, 4, 1, 3

first is 1

last is pivot-1 = 6-1 = 5

Loop in Line 4

Line 2: if $1 < 5$ then

Line 3: pivot = Partition (list, first, last)

After calling Partition, the conceptual outcome is

A = [2, 4, 1, 3, 5]

And PivotPoint = 5 is returned.

Therefore pivot = 5

⋮

Recursively, the conceptual outcome is

A = [1, 2, 3, 4, 5, 6, 8, 7]

End loop in Line 4.

Go to Line 5:

Line 5: Quicksort(list, pivot+1, last)

list is 8, 7

first is pivot+1=6+1=7

last is 8

Loop in Line 5

Line 2: if $7 < 8$ then

Line 3: pivot = Partition(list, first, last)

After calling Partition, the conceptual outcome is

A = [7, 8] and PivotPoint = 8 is returned.

Therefore pivot = 8

Line 5: Quicksort(list, pivot+1, last)

list is 7 (7 is element)

first is pivot + 1 = 8 + 1 = 9 (9 is index)

last is 1 (There is only one element in the list.)

Line 2: if 9 < 1 then what follows is not executed

End loop in Line 5. Go to Line 6.

Line 6: end if

Output A = [1, 2, 3, 4, 5, 6, 7, 8]

Appendix D

Shellsort Source Code

The Shellsort programme is presented as follows:

```
function [b]=shell_sort(a)
%SHELL_SORT Sort a one-dimension array by Shellsort.
%   [b]=shell_sort(a)
%   a = unsorted input
%   b = sorted ouput

if(nargin==0)
    error('No input.')
end

if(isempty(a))
    error('No empty array.')
end

a_size=size(a);
if((max(a_size))~=prod(a_size))
    error('This is not a one-dimesion array.')
end
```

```
sublist_position=[0 0];%sublist for holding indices                    20
length_a=length(a);
sublist_start_index=1;%sublist 1st starting index
number_of_sorting=1+floor(log2(length_a));%number of times
   to complete a sorting
                                                                        25
for index=1:number_of_sorting,
    interval=2^(number_of_sorting-index);%spacing for
       sublist
    while(length(sublist_position)~=1),%checking if
       sublist can be made                                              30
        sublist_position=[sublist_start_index:interval:...
        length_a];
        if(length(sublist_position)~=1)
            a(sublist_position)=...
            insertion_sort(a(sublist_position));                        35
        end
        sublist_start_index=sublist_start_index+1;
    end
    sublist_start_index=1;
    sublist_position=[0 0];                                             40
end

b=a;
```

```
function [b]=insertion_sort(a)
%INSERTION_SORT Sort a one-dimension array by insertion
   sort.
%   [b]=insertion_sort(a)
%   a = unsorted input                                        5
%   b = sorted ouput

if(nargin==0)
    error('No input.')
end                                                           10

if(isempty(a))
    error('No empty array.')
end
                                                              15
a_size=size(a);
if((max(a_size))~=prod(a_size))
    error('This is not a one-dimesion array.')
end
                                                              20
b=a(1);
insert_number=0;
less_than=0;
more_than_or_equal=0;
length_a=length(a);                                           25
less_than_logical_position=zeros(a_size);%an array to hold
   all the logical postion

for index=2:length_a,
    insert_number=a(index);                                   30
    less_than_logical_position=(b<insert_number);
    less_than=b(less_than_logical_position);%extract
       numbers less than insert_number
```

```
    if(isempty(less_than))%i.e. insert number is smaller
       than all numbers                                          35
        b=[insert_number,b];
    else
        more_than_or_equal=b(~less_than_logical_position);
        %extract numbers greater than or equal to
           insert_number                                         40

        b=[less_than,insert_number,more_than_or_equal];
    end
end
```

The above programme is written by following the literature review, Section 6.3 Shell- 45
sort.

Appendix E

Quicksort Source Code

The Quicksort programme is presented as below.

```
function [quick_sort_sublist]=...
quick_sort(list,first_element_index,last_element_index)
%QUICK_SORT Sort a one-dimension array by Quicksort.
%
   [quick_sort_sublist]=quick_sort(list,first_element_index,     5
%    last_element_index)
%    list = unsorted input
%    first_element_index = the index of the first element
   in an array
%    last_element_index = the index of the last element in      10
   an array
%    quick_sort_sublist = sorted ouput

if(nargin==0)
    error('No input.')                                           15
end

a_size=size(list);
if((max(a_size))~=prod(a_size))
```

```
    error('This is not a one-dimesion array.')                          20
end

if((first_element_index<last_element_index)&&...
    (length(list)>1))
    [pivot_point_index,partitioned_list]=partition(list,...             25
    first_element_index,last_element_index);

    quick_sort_sublist=quick_sort(partitioned_list,...
    first_element_index,pivot_point_index-1);
    %quick_sort_sublist_low=quick_sort_sublist%for testing               30
       use only
    partitioned_list=quick_sort_sublist;

    quick_sort_sublist=quick_sort(partitioned_list,...
    pivot_point_index+1,last_element_index);                            35
    %quick_sort_sublist_high=quick_sort_sublist
    %for testing use only
else
    quick_sort_sublist=list;
end                                                                     40
```

```
function [pivot_point_index,partitioned_list]=...
partition(list,first_element_index,last_element_index)
%PARTITION A sub-function used by Quicksort for sorting
   purpose.
%   [pivot_point_index,partitioned_list]=
%   partition(list,first_element_index,last_element_index)
%   list = unsorted input
%   first_element_index = the index of the first element
   in an array
%   last_element_index = the index of the last element in
   an array
%   pivot_point_index = the index of a pivot point in a
   partially sorted
%   array
%   partitioned_list = partially Quicksorted list

pivot_point_index=first_element_index;

if(length(list)>1)
    temp=0;
    pivot_value=list(first_element_index);

    for index=(first_element_index+1):last_element_index,
        if(list(index)<pivot_value)
            pivot_point_index=pivot_point_index+1;
            %The following 3 statements perform swopping
            temp=list(pivot_point_index);
            list(pivot_point_index)=list(index);
            list(index)=temp;
        end
    end
    %The following 3 statements perform swopping
    temp=list(first_element_index);
```

```
    list(first_element_index)=list(pivot_point_index);
    list(pivot_point_index)=temp;                            35
end

partitioned_list=list;
```

Appendix F

Fast 1D Median Filtering Source Code

The modified fast 1D median flitering programme follows:

```
function [median_array]=fast_median_1d(a,window_size)
%FAST_MEDIAN_1D Generate a set of median numbers for
   one-dimensional array.
%   FAST_MEDIAN_1D will only take elements from the 1st
   position to the
%   (length(a)-window_size+1)th positiom from array 'a'
   for median
%   calculation
%   [median_array]=fast_median_1d(a,window_size)
%   median_array = an array of median values
%   a = unsorted input
%   window_size = size of window (usually odd number)

if(nargin==0)
    error('No input.')
end
```

```
a_size=size(a);
if((max(a_size))~=prod(a_size))
    error('This is not an one-dimesion array.')                  20
end

length_a=length(a);
b=0;
c=zeros(1,length_a-1);                                             25
c_logical_index=logical(c);
less_than=0;
greater_than_or_equal=0;
sorted_sublist=bubble_sort(a(1:window_size));
half_size=length(sorted_sublist)/2;                                30
threshold=floor(half_size);
median_number=sorted_sublist(threshold+1);
median_array=zeros(1,(length_a-window_size+1));
median_array(1)=median_number;
less_than_median=threshold;                                        35
current_window=zeros(size(sorted_sublist));

for next_element_index=(window_size+1):length_a,
    sorted_sublist(min(find(sorted_sublist==...
    a(next_element_index-window_size))))=[];                      40
    c=sorted_sublist;
    b=a(next_element_index);

    %simplified insertion sort
    c_logical_index=(c<b);                                         45
    less_than=c(c_logical_index);
    greater_than_or_equal=c(~c_logical_index);
    current_window=[less_than,b,greater_than_or_equal];

    sorted_sublist=current_window;                                 50
```

```
    median_array(next_element_index-window_size+1)=...
    current_window(threshold+1);
end
```

Appendix G

Bubble Sort Source Code

The bubble sort programme is presented as follows:

```
function [b]=bubble_sort(a)
%BUBBLE_SORT Sort a one-dimension array by bubble sort.
%   [b]=bubble_sort(a)
%   a = unsorted input
%   b = sorted ouput

if(nargin==0)
    error('No input.')
end

if(isempty(a))
    error('No empty array.')
end

a_size=size(a);
if((max(a_size))~=prod(a_size))
    error('This is not a one-dimesion array.')
end
```

```
k=1;                                                    20
temp=0;
logic_true=1;
logic_false=0;
switching=logic_true;
lower_index=1;                                          25
upper_index=length(a);

while(switching),
    switching=logic_false;
    for index=lower_index:(upper_index-k),              30
        if(a(index)>a(index+1))
            temp=a(index+1);
            a(index+1)=a(index);
            a(index)=temp;
            switching=logic_true;                       35
        end
    end
    k=k+1;
end
                                                        40
b=a;
```

Appendix H

The Median Filter Algorithm in TCP Tahoe

```
/* This has been modified to use the tahoe code. */
void TcpAgent::rtt_update(double tao)
{
        static TracedInt store_1 = 1, store_2 = 1,
                         store_3 = 1, store_4 = 1,                5
                         store_5 = 1;
        TracedInt med_ian, temp;
        TracedInt ara[5];
        int max, ctr1, ctr2, j;
                                                                  10
        max = 5;
   double now = Scheduler::instance().clock();
   if (ts_option_)
      t_rtt_ = int(tao /tcp_tick_ + 0.5);
   else {                                                         15
      double sendtime = now - tao;
      sendtime += boot_time_;
      double tickoff = fmod(sendtime, tcp_tick_);
      t_rtt_ = int((tao + tickoff) / tcp_tick_);
```

```
}                                                                   20
if (t_rtt_ < 1)
   t_rtt_ = 1;
//
// t_srtt_ has 3 bits to the right of the binary point
// t_rttvar_ has 2                                                  25
     // Thus "t_srtt_ >> T_SRTT_BITS" is the actual
        srtt,
//   and "t_srtt_" is 8*srtt.
// Similarly, "t_rttvar_ >> T_RTTVAR_BITS" is the
//   actual rttvar, and "t_rttvar_" is 4*rttvar.                    30
//
     if (t_srtt_ != 0) {
   register short delta;
   delta = t_rtt_ - (t_srtt_ >> T_SRTT_BITS);
             // d = (m - a0)                                        35

             store_1 = store_2;
             store_2 = store_3;
             store_3 = store_4;
             store_4 = store_5;                                     40
             store_5 = t_rtt_;

             ara[0] = store_1;
             ara[1] = store_2;
             ara[2] = store_3;                                      45
             ara[3] = store_4;
             ara[4] = store_5;

             for (ctr1 = 0; ctr1 < (max - 1); ctr1++)
                { for (ctr2 = (ctr1 + 1); ctr2 < max;              50
                   ctr2++)
                     {if (ara[ctr1] > ara[ctr2])
```

```
                                {temp = ara[ctr1];
                                 ara[ctr1] = ara[ctr2];
                                 ara[ctr2] = temp;                  55
                                }

                             }
                       }
                                                                    60
                    j = max / 2;
                    med_ian = ara[j];
                    t_srtt_ = med_ian << T_SRTT_BITS;

          if (t_srtt_  <= 0)    // a1 = 7/8 a0 + 1/8 m              65
             t_srtt_ = 1;
          if (delta < 0)
             delta = -delta;
          delta -= (t_rttvar_ >> T_RTTVAR_BITS);
          if ((t_rttvar_ += delta) <= 0)                            70
             t_rttvar_ = 1;
       } else {
          t_srtt_ = t_rtt_ << T_SRTT_BITS;
                    // srtt = rtt
          t_rttvar_ = t_rtt_ << (T_RTTVAR_BITS-1);                  75
                    // rttvar = rtt / 2
       }
```

Appendix I

Simulation Script testing.tcl

```
# myeg10.tcl
# makes link lossy and samples TCP parameters
# also allows setting of traffic type (CBR/exp/pareto)
# use redmon.m to plot queue length data files
# use tcpmon.m to plot TCP parameter data files                5
#-----------------------------------------------
set donam          1
set dotrace        0

set starttime      1                                           10
set endtime        85

set qlensampletime 0.01
set tputsampletime 0.1
set tcpsampletime  0.1                                         15

set qsize          40

set lossstarttime  $starttime
set lossendtime    $endtime                                    20
set initlossrate   0.01
```

```
set lossrate       $initlossrate

set grandthrough   0
set grandgoodput   0                                        25
set doappendresult 1

#If simulationmode=0, TCP Tahoe (normal tcp.cc) is used.
#If simulationmode=1, TCP Tahoe with median filter
#(modified tcp.cc) is used.                                 30
#Whenever tcp.cc is modified, go to the directory
#ns-2.xx and issue the 'make' command. Otherwise, this
#Tcl script will not produce correct results.

set simulationmode 1                                        35

if { $simulationmode == 0 } {
    set mode nar
}
                                                            40
if { $simulationmode == 1 } {
    set medianwindowsize  5

    set mode med$medianwindowsize
}                                                           45

# traffic type
#set traftype        "CBR"
#set traftype        "Exponential"
set traftype        "Pareto"                                50
#-----------------------------------------------

#-----------------------------------------------
switch $traftype {
```

```
"CBR"  {                                                        55
    #------------------------------------------------
    puts "Traffic type CBR"
    # packet size (default 210)
    set tspktsize     210
                                                                60
    # send rate during 'on' times (default 448000)
    set tsrate        448000
    #------------------------------------------------
}
                                                                65
"Exponential" {
    puts "Traffic type Exponential"
    #------------------------------------------------
    # Application/Traffic/Exponential
    # exponential on/off traffic source parameters               70
    # see P321 of ns manual

    # packet size
    set tspktsize     1000
                                                                75
    # average 'on' time
    set tsbursttime   1

    # average 'idle' time
    set tsidletime    2                                         80

    # send rate during 'on' times
    set tsrate        448000
    #------------------------------------------------
}                                                               85

"Pareto" {
```

```
        puts "Traffic type Pareto"
        #-----------------------------------------------
        # Application/Traffic/Pareto                      90
        # Pareto on/off traffic source parameters
        # see P321-2 of ns manual

        # packet size
        set tspktsize     1000                            95

        # average 'on' time
        set tsbursttime   1

        # average 'idle' time                             100
        set tsidletime    2

        # send rate during 'on' times
        set tsrate        448000
                                                          105
        # pareto shape param
        set tsshape       2.5
        #-----------------------------------------------
    }
                                                          110
    default {
        puts "Unknown traffic type"
        exit
    }
}                                                         115
#exit
#-----------------------------------------------

#-----------------------------------------------
#puts "argc=$argc argv=$argv"                             120
```

```
# defaults
set npeers    6
set qtype "DropTail"
                                                              125
# if 1 arg given, it is number of sources
if { $argc == 1 } {
    set npeers [lindex $argv 0]
}
                                                              130
# if 2 args given, it is number of peers & queue type
if { $argc == 2 } {
    set npeers [lindex $argv 0]
    set qtype  [lindex $argv 1]
}                                                             135

puts "npeers $npeers"
puts "qtype $qtype"

#exit                                                         140
#----------------------------------------------

#----------------------------------------------
switch $qtype {
    "DropTail"  {                                             145
        puts "testing with DropTail queue"
    }

    "RED" {
        puts "testing with RED queue"                         150
    }

    default {
```

```
        puts "unknown queue type '$qtype'"
        exit                                                        155
    }
}
#exit
#-----------------------------------------------
                                                                    160
#-----------------------------------------------
set ns [new Simulator]
#-----------------------------------------------

#-----------------------------------------------                    165
#Define different colors for data flows
set colorlist { blue red magenta orange yellow black gold
   green pink purple }
set ncolors [llength $colorlist]
#puts "ncolors=$ncolors"                                            170
for {set n 1} {$n <= $npeers} {incr n} {
    # array indexes start at 0 but loop var starts at 1
    set coloridx [expr (($n-1) % $ncolors)]
    set colorname [lindex $colorlist $coloridx]
    #puts "color $coloridx for flow $n = $colorname"                175
    $ns color $n $colorname
}
#-----------------------------------------------

#-----------------------------------------------                    180
set qlenfilename \
qlen-$mode-$npeers-$qtype-$traftype-$lossrate.dat
set tpfilename   \
tput-$mode-$npeers-$qtype-$traftype-$lossrate.dat
set gpfilename   \                                                  185
gput-$mode-$npeers-$qtype-$traftype-$lossrate.dat
```

```
set tcpfilename  \
tcp-$mode-$npeers-$qtype-$traftype-$lossrate.dat
set outnamname   \
out-$mode-$npeers-$qtype-$traftype-$lossrate.nam
set tracname     \
trac-$mode-$npeers-$qtype-$traftype-$lossrate.tr

if { $donam == 1 } {
    set nf [open $outnamname w]
    $ns namtrace-all $nf
}

if { $dotrace == 1 } {
    set tf [open $tracname w]
    $ns trace-all $tf
}

if { $doappendresult == 1 } {
    set resultname \
    result-$mode-$npeers-$qtype-$traftype.dat
    set rf [open $resultname a]
}
set qlenfile [open $qlenfilename w]
set tpfile   [open $tpfilename   w]
set gpfile   [open $gpfilename   w]
set tcpfile  [open $tcpfilename  w]
#-----------------------------------------------

#-----------------------------------------------
#set S0 [$ns node]
#set S1 [$ns node]

for {set n 1} {$n <= $npeers} {incr n} {
```

```
    puts "create src-dest $n"                                          220
    set src($n) [$ns node]
    set dest($n) [$ns node]
}
#-----------------------------------------------
                                                                        225
#-----------------------------------------------
set R0 [$ns node]
set R1 [$ns node]
#-----------------------------------------------
                                                                        230
#-----------------------------------------------
#$ns duplex-link $S0 $R0 10Mb 1ms DropTail
#$ns duplex-link $S1 $R0 10Mb 1ms DropTail

#$ns duplex-link $R1 $D0 10Mb 1ms DropTail                             235
#$ns duplex-link $R1 $D1 10Mb 1ms DropTail

for {set n 1} {$n <= $npeers} {incr n} {
    puts "create link $n"
                                                                        240
    # links to routers are droptail
    $ns duplex-link $src($n) $R0      10Mb 10ms  DropTail
    $ns duplex-link $R1      $dest($n) 10Mb 10ms  DropTail
}
                                                                        245
#$ns duplex-link $R0 $R1 1M 100ms DropTail
$ns duplex-link $R0 $R1 1M 100ms $qtype

$ns duplex-link-op $R0 $R1 queuePos 0.5
$ns queue-limit $R0 $R1 $qsize                                         250
#-----------------------------------------------
```

```
#-----------------------------------------------
# NAM
$ns duplex-link-op $R0 $R1 orient right                              255

for {set n 1} {$n <= $npeers} {incr n} {
    #set a [expr 270+20*$i]

    # this gives nodes going from 90 to 270 degrees                   260
    #set a [expr 270 + 180/($npeers-1)*($n-1)]

    # this gives slightly offset (20 deg off each axis)
    set a [expr 270 + 20 + 140/($npeers-1)*($n-1)]
                                                                      265
    if {$a > 360} {
        set a [expr $a-360]
    }

    $ns duplex-link-op $src($n) $R0       orient [expr                270
       $a]deg
    $ns duplex-link-op $R1      $dest($n) orient [expr
       $a]deg
}
#-----------------------------------------------                      275

#-----------------------------------------------
#set tcp0 [new Agent/TCP]
#$ns attach-agent $S0 $tcp0
                                                                      280
#set tcp1 [new Agent/TCP]
#$ns attach-agent $S1 $tcp1

#set sink0 [new Agent/TCPSink]
#$ns attach-agent $D0 $sink0                                          285
```

```
#set sink1 [new Agent/TCPSink]
#$ns attach-agent $D1 $sink1

#$ns connect $tcp0 $sink0                                         290
#$ns connect $tcp1 $sink1
#-----------------------------------------------

#-----------------------------------------------
#$ts0 attach-agent $tcp0                                          295
#$ts1 attach-agent $tcp1
#-----------------------------------------------

#-----------------------------------------------
for {set n 1} {$n <= $npeers} {incr n} {                          300
    puts "creating src $n"
    set srctcp($n) [new Agent/TCP]
    $ns attach-agent $src($n) $srctcp($n)
    $srctcp($n) set fid_ $n
}                                                                 305

for {set n 1} {$n <= $npeers} {incr n} {
    puts "creating dest $n"

    set desttcp($n) [new Agent/TCPSink]                           310
    $ns attach-agent $dest($n) $desttcp($n)
    $desttcp($n) set fid_ $n
}
#-----------------------------------------------
                                                                  315
#-----------------------------------------------
# define traffic sources
```

```
for {set n 1} {$n <= $npeers} {incr n} {
    puts "creating traffic data source $n"                    320

    #----------------------------------------------
    switch $traftype {
        "CBR"  {
            #----------------------------------------------    325
            # constant bitrate
            set datasrc($n) [new Application/Traffic/CBR]

            $datasrc($n) set packetSize_ $tspktsize
            $datasrc($n) set rate_       $tsrate                330
            #----------------------------------------------
        }

        "Exponential" {
            #----------------------------------------------    335
            # exponential on/off
            set datasrc($n) [new
               Application/Traffic/Exponential]

            $datasrc($n) set packetSize_ $tspktsize             340
            $datasrc($n) set burst_time_ $tsbursttime
            $datasrc($n) set idle_time_  $tsidletime
            $datasrc($n) set rate_       $tsrate
            #----------------------------------------------
        }                                                      345

        "Pareto" {
            #----------------------------------------------
            # pareto on/off
            set datasrc($n) [new                               350
               Application/Traffic/Pareto]
```

```
            $datasrc($n) set packetSize_ $tspktsize
            $datasrc($n) set burst_time_ $tsbursttime
            $datasrc($n) set idle_time_  $tsidletime         355
            $datasrc($n) set rate_       $tsrate
            $datasrc($n) set shape_      $tsshape
            #-----------------------------------------------
        }
    }                                                        360
    #-----------------------------------------------

    $datasrc($n) attach-agent $srctcp($n)
}
#-----------------------------------------------             365

#-----------------------------------------------
# connect TCP's
for {set n 1} {$n <= $npeers} {incr n} {
    $ns connect $srctcp($n) $desttcp($n)                     370
}
#-----------------------------------------------

#-----------------------------------------------
set monqueue [$ns monitor-queue $R0 $R1 stdout]             375

#set tracedqueue [[$ns link $R0 $R1] queue]
set qmonlink [$ns link $R0 $R1]
set tracedqueue [$qmonlink queue]
                                                             380
set numsamplestp  0
set numsamplesql  0
#-----------------------------------------------
```

```
#-----------------------------------------------                 385
# error/lossy link
set lossModel [new ErrorModel]

# loss rate
$lossModel set rate_ $initlossrate                               390

$lossModel unit packet
$lossModel drop-target [new Agent/Null]
set lossmonlink [$ns link $R0 $R1]
$lossmonlink install-error $lossModel                            395
#-----------------------------------------------

#-----------------------------------------------
proc finish { } {
    global ns nf tf rf                                           400
    global qlenfilename tpfilename gpfilename tcpfilename
    global outnamname tracname
    global qlenfile tpfile gpfile tcpfile
    global donam dotrace doappendresult
    global grandthrough grandgoodput                             405
    global initlossrate

    close $qlenfile
    close $tpfile
    close $gpfile                                                410
    close $tcpfile

    if { $donam == 1 } {
        close $nf
    }                                                            415

    if { $dotrace == 1 } {
```

```
        $ns flush-trace
        close $tf
    }                                                           420

    if { $doappendresult == 1 } {
        puts $rf "$initlossrate $grandthrough
           $grandgoodput"
        close $rf                                               425
    }

    puts "Output files:"
    puts "queue length file: $qlenfilename"
    puts "throughput file:   $tpfilename"                       430
    puts "goodput file:      $gpfilename"
    puts "TCP param file:    $tcpfilename"
    puts "Grand Throughput:  $grandthrough"
    puts "Grand Goodput:     $grandgoodput"
                                                                435
    if { $donam == 1 } {
        exec nam $outnamname &
    }

    exit 0                                                      440
}
#-----------------------------------------------


#-----------------------------------------------
proc samplethroughput { } {                                     445
    global ns npeers tputsampletime
    global tracedqueue #tcp0 tcp1
    global tpfile gpfile
    global srctcp    # array
    global desttcp   # array                                    450
```

```
global numsamplestp
global grandthrough grandgoodput

set now [$ns now]
set numsamplestp [expr $numsamplestp+1]                      455

puts "throughput:"
set ndbytes   0
set nrexmitbytes 0
                                                             460
for {set n 1} {$n <= $npeers} {incr n} {
    #puts "tcp $n"

    set ndbsrctcp($n) [$srctcp($n) set ndatabytes_ ]
    #puts "ndatabytes for src $n = $ndbsrctcp($n)"            465
    set ndbytes [expr $ndbytes+$ndbsrctcp($n)]

    set nrexmitsrctcp($n) [$srctcp($n) set
       nrexmitbytes_]
    #puts "nrexmit databytes for src $n =                      470
       $nrexmitsrctcp($n)"

    set nrexmitbytes [expr
       $nrexmitbytes+$nrexmitsrctcp($n)]
}                                                            475
puts "totals: $ndbytes $nrexmitbytes"

set kbps [expr
   double($ndbytes*8/($tputsampletime*1000))]
puts "kbps = $kbps"                                          480

set goodkbps [expr double(($ndbytes -
   $nrexmitbytes)*8/($tputsampletime*1000))]
```

```
    puts "goodkbps = $goodkbps"
                                                                    485
    if { $numsamplestp > 1 } {
        # first sample should not be used
        puts $tpfile "$kbps"
        puts $gpfile "$goodkbps"
        set grandthrough [expr $grandthrough+$kbps]                 490
        set grandgoodput [expr $grandgoodput+$goodkbps]
    }

    for {set n 1} {$n <= $npeers} {incr n} {
        $srctcp($n) set ndatabytes_ 0                               495
        $srctcp($n) set nrexmitbytes_ 0
    }

    $ns at [expr $now+$tputsampletime] "samplethroughput"
                                                                    500
    # only works for RED queues
    #set currqueue [$tracedqueue set curq_]
    #set avqueue [$tracedqueue set ave_]
    #puts "current queue=$currqueue  average
      queue=$avqueue"                                               505
}
#-----------------------------------------------


#-----------------------------------------------
#proc samplequeuelen { sum number } {                               510
proc samplequeuelen { } {
    global ns monqueue qlensampletime
    global qlenfile
    global numsamplesql
                                                                    515
    set now [$ns now]
```

```
    set numsamplesql [expr $numsamplesql+1]

    set qlen [$monqueue set pkts_]                                520
    #puts "LEN=$qlen"

    #set sum [expr $sum+$len]
    #set number [expr $number+1]
    #set avqlen [expr 1.0*$sum/$number]                           525
    #puts "Av queue length: $avqlen"

    if { $numsamplesql > 1 } {
        # first sample should not be used
        #puts $qlenfile "$avqlen"                                  530
        puts $qlenfile "$qlen"
    }

    #$ns at [expr $now+$qlensampletime] "samplequeuelen
       $sum $number"                                               535
    $ns at [expr $now+$qlensampletime] "samplequeuelen"
}
#-----------------------------------------------


#-----------------------------------------------                   540
proc sampletcpparams { tcpsrc } {
    global ns tcpsampletime
    global tcpfile

    set now [$ns now]                                              545

    # see ns-default.tcl for parameter names
    set currwindow     [$tcpsrc set window_]
    set currcwnd       [$tcpsrc set cwnd_]
```

```
set currawnd       [$tcpsrc set awnd_]                                    550
set currmaxcwnd    [$tcpsrc set maxcwnd_]
set currseqno      [$tcpsrc set seqno_]
set currtseqno     [$tcpsrc set t_seqno_]
set currpacketsize [$tcpsrc set packetSize_]
                                                                          555
set currack      [$tcpsrc set ack_]
set currdupacks  [$tcpsrc set dupacks_]
set currssthresh [$tcpsrc set ssthresh_]
set currrtt      [$tcpsrc set rtt_]
set currsrtt     [$tcpsrc set srtt_]                                      560
set currrttvar   [$tcpsrc set rttvar_]
set currmaxrto   [$tcpsrc set maxrto_]
set currminrto   [$tcpsrc set minrto_]

puts "sample TCP: time=$now"                                              565
puts "window = $currwindow"
puts "cwnd = $currcwnd"
puts "awnd = $currawnd"
puts "ack = $currack"
puts "dupacks = $currdupacks"                                             570
puts "maxcwnd = $currmaxcwnd"
puts "seqno = $currseqno"
puts "tseqno = $currtseqno"
puts "packetsize = $currpacketsize"
                                                                          575
puts "ssthresh = $currssthresh"
puts "rtt = $currrtt"
puts "rttvar = $currrttvar"
puts "maxrto = $currmaxrto"
puts "minrto = $currminrto"                                               580

# write tcp params to file
```

```
    # whichever of the above are necessary
    puts $tcpfile "$currwindow $currcwnd \
                  $currawnd $curack \
                  $currdupacks $currmaxcwnd \
                  $currseqno $currpacketsize \
                  $currssthresh $currrtt $currrttvar \
                  $currmaxrto $currminrto $currsrtt"

    $ns at [expr $now+$tcpsampletime] "sampletcpparams
       $tcpsrc"
}
#----------------------------------------------

#----------------------------------------------
proc setlossrate { lossrate } {
    global ns lossModel

    set now [$ns now]

    puts "**** setting loss rate $lossrate at time $now"
    $lossModel set rate_ $lossrate
}
#----------------------------------------------

#----------------------------------------------
# start sources
#$ns at 0 "$ts0 start"
#$ns at 0 "$ts1 start"

for {set n 1} {$n <= $npeers} {incr n} {
    puts "start $n"

    $ns at 0 "$datasrc($n) start"
```

```
}
#--------------------------------------------------


#--------------------------------------------------
# start sampling                                              620
#$ns at $qlensampletime "samplequeuelen 0 0"
#$ns at $tputsampletime "samplethroughput"
#$ns at $sampletime "sampletcp0"

#$ns at $starttime "samplequeuelen 0 0"                       625
$ns at $starttime "samplequeuelen"
$ns at $starttime "samplethroughput"
$ns at $starttime "sampletcpparams $srctcp(1)"
#--------------------------------------------------
                                                              630
#--------------------------------------------------
# introduce errors on link
$ns at $lossstarttime "setlossrate $lossrate"
$ns at $lossendtime   "setlossrate $lossrate"
#--------------------------------------------------           635


#--------------------------------------------------
$ns at $endtime "finish"
#--------------------------------------------------
                                                              640
#--------------------------------------------------
# run simulation
$ns run
#--------------------------------------------------
```

Appendix J

Simulation Script wirelessa.tcl

```
# Modified from wireless2.tcl in Marc Greis' Tutorial
# simulation of a wireless scenario consisting of 2
# wireless nodes. Both nodes are not moving.
#
# ========================================================
# Define options
# ========================================================
# Refer to directories /ns/mobile; /ns/mac
# Refer to mtp.tcl in /ns/tcl/ex/wireless-scripts/
# channel type
set opt(chan)           Channel/WirelessChannel
# radio-propagation model
set opt(prop)           Propagation/Shadowing
# network interface type
set opt(netif)          Phy/WirelessPhy
# MAC type
set opt(mac)            Mac/802_11
# interface queue type
set opt(ifq)            Queue/DropTail
# link layer type
set opt(ll)             LL
```

```
# antenna model
set opt(ant)            Antenna/OmniAntenna
# max packet in ifq
set opt(ifqlen)         100                                 25
# number of mobilenodes
set opt(nn)             2
# routing protocol
set opt(adhocRouting)   DSDV
                                                            30
# connection pattern file
set opt(cp)             ""
# node movement file
set opt(sc)             ""
                                                            35
# x coordinate of topology
set opt(x)              670
# y coordinate of topology
set opt(y)              670
# seed for random number gen                                40
set opt(seed)           0.0
# time to stop simulation
set opt(stop)           100.0

set opt(ftp1-start)     2.0                                 45

set lossrate            0.000
set tcptype             "Tahoe"

#Configuration for wireless link configuration              50
#Refer to the chapter on Radio Propagation Models
#in ns-Manual
set wirelessbandwidth   2Mb
set wirelessdelay       1ms
```

```
Phy/WirelessPhy set bandwidth_ $wirelessbandwidth                     55
# 2.4GHz; can be the exact channel carrier frequency
Phy/WirelessPhy set freq_ 2.4e9
Mac/802_11 set dataRate_ $wirelessbandwidth
Mac/802_11 set basicRate_ $wirelessbandwidth
LL set bandwidth_ $wirelessbandwidth                                  60
LL set delay_ $wirelessdelay
ErrorModel set bandwidth_ $wirelessbandwidth
ErrorModel set delay_ $wirelessdelay

# path loss exponent                                                  65
Propagation/Shadowing set pathlossExp_ 2
# shadowing deviation (dB)
Propagation/Shadowing set std_db_ 4
# seed for random number generator
Propagation/Shadowing set seed_ 0                                     70
#Receiving threshold in the wireless network interface
#This value can be found from the specification sheet
#of the wireless device being used for the simulation
#Otherwise, this value is calculated by using threshold.cc
Phy/WirelessPhy set RXThresh_ 2.45199e-9 ; #(for 50m)                 75

# Read command line arguments
if {$argc > 1} {
    set tcptype  [lindex $argv 0]
    set lossrate [lindex $argv 1]                                     80
} else {
    puts "Usage: ns tcl_script_name.tcl <protocol>\
    <error rate>"
    puts " "
    puts "<protocol> is Reno, Newreno, Tahoe, Sack, Vegas"            85
    puts "<error rate> is the link error rate (0.001\
    = 0.1%). Use 0 for no errors"
```

```
}

set sinktype TCPSink                                        90
if {$tcptype == "Tahoe"} {
   set sourcetype TCP
} elseif {$tcptype == "Reno"} {
   set sourcetype TCP/Reno
} elseif {$tcptype == "Newreno"} {                          95
   set sourcetype TCP/Newreno
} elseif {$tcptype == "Sack"} {
   set sourcetype TCP/Sack1
   set sinktype TCPSink/Sack1
} elseif {$tcptype == "Vegas"} {                            100
   set sourcetype TCP/Vegas
} else {
   puts "TCP type is not defined in ns\n"
   exit
}                                                           105

proc stop {} {
    global ns_ tracefd namtrace
    #$ns_ flush-trace
    close $tracefd                                          110
    close $namtrace
}

proc UniformErr {} {
    global ns_ err lossrate                                 115
    set err [new ErrorModel]
    $err set rate_ $lossrate
    $err unit pkt
    $err ranvar [new RandomVariable/Uniform]
    #Drop-target is not defined in this case. Errors will   120
```

```
    #be handled by the agents
    return $err
}

# ======================================================          125
# check for boundary parameters and random seed
# ======================================================
if { $opt(x) == 0 || $opt(y) == 0 } {
   puts "No X-Y boundary values given for wireless\
            topology\n"                                            130
}
if {$opt(seed) > 0} {
   puts "Seeding Random number generator with\
            $opt(seed)\n"
   ns-random $opt(seed)                                            135
}

# create simulator instance
set ns_   [new Simulator]
                                                                   140
set tracefd  [open wirelessa-out.tr w]
$ns_ trace-all $tracefd

# Create topography object
set topo   [new Topography]                                        145

# define topology
$topo load_flatgrid $opt(x) $opt(y)

# create God                                                       150
create-god $opt(nn)

#wiredRouting IS "OFF"
```

```
$ns_ node-config -adhocRouting $opt(adhocRouting) \
                 -llType $opt(ll) \                                155
                 -macType $opt(mac) \
                 -ifqType $opt(ifq) \
                 -ifqLen $opt(ifqlen) \
                 -antType $opt(ant) \
                 -propType $opt(prop) \                            160
                 -phyType $opt(netif) \
                 -channel [new $opt(chan)] \
                 -topoInstance $topo \
                 -wiredRouting OFF \
                 -agentTrace ON \                                  165
                 -routerTrace OFF \
                 -macTrace OFF

# note the position and movement of mobilenodes is as
# defined in $opt(sc)                                              170

#configure for mobilenodes
$ns_ node-config -IncomingErrProc UniformErr \
                 -OutcomingErrProc UniformErr
                                                                   175
for {set j 0} {$j < $opt(nn)} {incr j} {
    set node_($j) [$ns_ node]
    $node_($j) random-motion 0 ;# disable random motion
}
                                                                   180
$node_(0) set X_ 335.0
$node_(0) set Y_ 335.0
$node_(0) set Z_ 0.0

$node_(1) set X_ 385.0                                             185
$node_(1) set Y_ 335.0
```

```
$node_(1) set Z_ 0.0

set tcp1 [new Agent/$sourcetype]
$tcp1 set fid_ 1                                                    190
set sink1 [new Agent/$sinktype]
$ns_ attach-agent $node_(0) $tcp1
$ns_ attach-agent $node_(1) $sink1
$ns_ connect $tcp1 $sink1
set ftp1 [new Application/FTP]                                      195
$ftp1 attach-agent $tcp1
$ns_ at $opt(ftp1-start) "$ftp1 start"

# source connection-pattern and node-movement scripts
if { $opt(cp) == "" } {                                             200
   puts "*** NOTE: Connection pattern in the Tcl\
      script will be used."
      set opt(cp) "none"
} else {
   puts "Loading connection pattern..."                             205
   source $opt(cp)
}
if { $opt(sc) == "" } {
   puts "*** NOTE: Scenario setting in the Tcl\
      script will be used."                                         210
      set opt(sc) "none"
} else {
   puts "Loading scenario file..."
   source $opt(sc)
   puts "Load complete..."                                          215
}

# Define initial node position in nam
```

```
for {set i 0} {$i < $opt(nn)} {incr i} {                    220

    # 20 defines the node size in nam, must adjust it
    # according to your scenario
    # The function must be called after mobility model is
    # defined                                                225

    $ns_ initial_node_pos $node_($i) 20
}

# Tell all nodes when the simulation ends                    230
for {set i } {$i < $opt(nn) } {incr i} {
    $ns_ at $opt(stop).0 "$node_($i) reset";
}

$ns_ at $opt(stop).0002 "puts \"NS EXITING...\" ;\          235
                        $ns_ halt"
$ns_ at $opt(stop).0001 "stop"

# informative headers for CMUTracefile
puts $tracefd "M 0.0 nn $opt(nn) x $opt(x) y $opt(y) rp\    240
              $opt(adhocRouting)"
puts $tracefd "M 0.0 sc $opt(sc) cp $opt(cp) seed\
              $opt(seed)"
puts $tracefd "M 0.0 prop $opt(prop) ant $opt(ant)"
                                                             245
puts "Starting Simulation..."
$ns_ run
```

Appendix K

Simulation Script wirelessb.tcl

```
# Modified from wireless2.tcl in Marc Greis' Tutorial
# simulation of a wired-cum-wireless scenario consisting
# of 1 wireless node, 1 wired node and 1 base station.
# Wireless node is moving towards the base station.
# =========================================================    5
# Define options
# =========================================================
# Refer to directories /ns/mobile; /ns/mac
# Refer to mtp.tcl in /ns/tcl/ex/wireless-scripts/
# channel type                                                 10
set opt(chan)           Channel/WirelessChannel
# radio-propagation model
set opt(prop)           Propagation/Shadowing
# network interface type
set opt(netif)          Phy/WirelessPhy                        15
# MAC type
set opt(mac)            Mac/802_11
# interface queue type
set opt(ifq)            Queue/DropTail
# link layer type                                              20
set opt(ll)             LL
```

```
# antenna model
set opt(ant)            Antenna/OmniAntenna
# max packet in ifq
set opt(ifqlen)         100
# number of mobilenodes
set opt(nn)             1
# routing protocol
set opt(adhocRouting)   DSDV

# connection pattern file
set opt(cp)             ""
# node movement file
set opt(sc)             ""

# x coordinate of topology
set opt(x)              670
# y coordinate of topology
set opt(y)              670
# seed for random number gen
set opt(seed)           0.0
# time to stop simulation
set opt(stop)           100.0

set opt(ftp1-start)     2.0

set num_wired_nodes     1
set num_bs_nodes        1

set lossrate            0.000
set tcptype             "Tahoe"

#Configuration for wireless link configuration
#Refer to the chapter on Radio Propagation Models
```

```
#in ns-Manual                                                          55
set wirelessbandwidth   2Mb
set wirelessdelay       1ms
Phy/WirelessPhy set bandwidth_ $wirelessbandwidth
# 2.4GHz; can be the exact channel carrier frequency
Phy/WirelessPhy set freq_ 2.4e9                                        60
Mac/802_11 set dataRate_ $wirelessbandwidth
Mac/802_11 set basicRate_ $wirelessbandwidth
LL set bandwidth_ $wirelessbandwidth
LL set delay_ $wirelessdelay
ErrorModel set bandwidth_ $wirelessbandwidth                           65
ErrorModel set delay_ $wirelessdelay

# path loss exponent
Propagation/Shadowing set pathlossExp_ 2
# shadowing deviation (dB)                                             70
Propagation/Shadowing set std_db_ 4
# seed for random number generator
Propagation/Shadowing set seed_ 0
#Receiving threshold in the wireless network interface
#This value can be found from the specification sheet                  75
#of the wireless device being used for the simulation
#Otherwise, this value is calculated by using threshold.cc
Phy/WirelessPhy set RXThresh_ 2.45199e-9 ; #(for 50m)

# Read command line arguments                                          80
if {$argc > 1} {
    set tcptype  [lindex $argv 0]
    set lossrate [lindex $argv 1]
} else {
    puts "Usage: ns tcl_script_name.tcl <protocol>\                    85
    <error rate>"
    puts " "
```

```
    puts "<protocol> is Reno, Newreno, Tahoe, Sack, Vegas"
    puts "<error rate> is the link error rate (0.001\
    = 0.1%). Use 0 for no errors"                                90
}

set sinktype TCPSink
if {$tcptype == "Tahoe"} {
   set sourcetype TCP                                            95
} elseif {$tcptype == "Reno"} {
   set sourcetype TCP/Reno
} elseif {$tcptype == "Newreno"} {
   set sourcetype TCP/Newreno
} elseif {$tcptype == "Sack"} {                                  100
   set sourcetype TCP/Sack1
   set sinktype TCPSink/Sack1
} elseif {$tcptype == "Vegas"} {
   set sourcetype TCP/Vegas
} else {                                                         105
   puts "TCP type is not defined in ns\n"
   exit
}

proc stop {} {                                                   110
    global ns_ tracefd namtrace
    #$ns_ flush-trace
    close $tracefd
    close $namtrace
}                                                                115

proc UniformErr {} {
    global ns_ err lossrate
    set err [new ErrorModel]
    $err set rate_ $lossrate                                     120
```

```
    $err unit pkt
    $err ranvar [new RandomVariable/Uniform]
    #Drop-target is not defined in this case. Errors will
    #be handled by the agents
    return $err                                              125
}

# =====================================================
# check for boundary parameters and random seed
# =====================================================      130
if { $opt(x) == 0 || $opt(y) == 0 } {
   puts "No X-Y boundary values given for wireless\
      topology\n"
}
if {$opt(seed) > 0} {                                        135
   puts "Seeding Random number generator with\
      $opt(seed)\n"
   ns-random $opt(seed)
}
                                                             140
# create simulator instance
set ns_   [new Simulator]

# set up for hierarchical routing
$ns_ node-config -addressType hierarchical                   145
# number of domains
AddrParams set domain_num_ 2
# number of clusters in each domain
lappend cluster_num 1 1
AddrParams set cluster_num_ $cluster_num                     150
# number of nodes in each cluster of each domain
lappend eilastlevel 1 2
AddrParams set nodes_num_ $eilastlevel
```

```
set tracefd  [open wirelessb-out.tr w]                                155
$ns_ trace-all $tracefd

# Create topography object
set topo   [new Topography]
                                                                      160
# define topology
$topo load_flatgrid $opt(x) $opt(y)

# create God
create-god [expr $opt(nn) + $num_bs_nodes]                            165

#create wired nodes
# hierarchical addresses for wired domain
set temp {0.0.0 0.1.0}
for {set i 0} {$i < $num_wired_nodes} {incr i} {                      170
    set W($i) [$ns_ node [lindex $temp $i]]
}

#wiredRouting IS "ON"
# configure for base-station node                                     175
$ns_ node-config -adhocRouting $opt(adhocRouting) \
                 -llType $opt(ll) \
                 -macType $opt(mac) \
                 -ifqType $opt(ifq) \
                 -ifqLen $opt(ifqlen) \                               180
                 -antType $opt(ant) \
                 -propType $opt(prop) \
                 -phyType $opt(netif) \
                 -channel [new $opt(chan)] \
                 -topoInstance $topo \                                185
                 -wiredRouting ON \
```

```
                 -agentTrace ON \
                 -routerTrace OFF \
                 -macTrace OFF

#create base-station node
# hier address to be used for wireless domain
set temp {1.0.0 1.0.1 1.0.2 1.0.3}

set BS(0) [$ns_ node [lindex $temp 0]]
# disable random motion
$BS(0) random-motion 0

#provide some co-ord (fixed) to base station node
$BS(0) set X_ 335.0
$BS(0) set Y_ 335.0
$BS(0) set Z_ 0.0

# create mobilenodes in the same domain as BS(0)
# note the position and movement of mobilenodes is as
# defined in $opt(sc)

#wiredRouting IS "OFF"
#configure for mobilenodes
$ns_ node-config -wiredRouting OFF
$ns_ node-config -IncomingErrProc UniformErr
                 -OutcomingErrProc UniformErr

  for {set j 0} {$j < $opt(nn)} {incr j} {
    set node_($j) [ $ns_ node [lindex $temp \
       [expr $j+1]] ]
    $node_($j) base-station [AddrParams addr2id \
       [$BS(0) node-addr]]
}
```

```
#
# nodes: 1, pause: 0.00, max speed: 0.5, max x: 670.00,
# max y: 670.00
#
$node_(0) set X_ 385.0
$node_(0) set Y_ 335.0
$node_(0) set Z_ 0.0
$ns_ at 0.0 "$node_(0) setdest 335.0 335.0 0.5"
#
# Destination Unreachables: 0
#
# Route Changes: 0
#
# Link Changes: 0
#
# Node | Route Changes | Link Changes
#    0 |             0 |            0
#

#create links between wired and BS nodes

$ns_ duplex-link $W(0) $BS(0) 10Mb 45ms DropTail
$ns_ queue-limit $W(0) $BS(0) 500

$ns_ duplex-link-op $W(0) $BS(0) orient right
$ns_ duplex-link-op $W(0) $BS(0) queuePos 0.5

set tcp1 [new Agent/$sourcetype]
$tcp1 set fid_ 1
set sink1 [new Agent/$sinktype]
$ns_ attach-agent $W(0) $tcp1
$ns_ attach-agent $node_(0) $sink1
```

```
$ns_ connect $tcp1 $sink1
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1                                        255
$ns_ at $opt(ftp1-start) "$ftp1 start"

# source connection-pattern and node-movement scripts
if { $opt(cp) == "" } {
   puts "*** NOTE: Connection pattern in the Tcl\               260
      script will be used."
      set opt(cp) "none"
} else {
   puts "Loading connection pattern..."
   source $opt(cp)                                              265
}
if { $opt(sc) == "" } {
   puts "*** NOTE: Scenario setting in the Tcl\
      script will be used."
      set opt(sc) "none"                                        270
} else {
   puts "Loading scenario file..."
   source $opt(sc)
   puts "Load complete..."
}                                                               275

# Define initial node position in nam

for {set i 0} {$i < $opt(nn)} {incr i} {
                                                                280
    # 20 defines the node size in nam, must adjust it
    # according to your scenario
    # The function must be called after mobility model is
    # defined
                                                                285
```

```
    $ns_ initial_node_pos $node_($i) 20
}

# Tell all nodes when the simulation ends
for {set i } {$i < $opt(nn) } {incr i} {                                290
    $ns_ at $opt(stop).0 "$node_($i) reset";
}
$ns_ at $opt(stop).0 "$BS(0) reset";

$ns_ at $opt(stop).0002 "puts \"NS EXITING...\" ;\                      295
                        $ns_ halt"
$ns_ at $opt(stop).0001 "stop"

# informative headers for CMUTracefile
puts $tracefd "M 0.0 nn $opt(nn) x $opt(x) y $opt(y) rp\                300
              $opt(adhocRouting)"
puts $tracefd "M 0.0 sc $opt(sc) cp $opt(cp) seed\
              $opt(seed)"
puts $tracefd "M 0.0 prop $opt(prop) ant $opt(ant)"
                                                                        305
puts "Starting Simulation..."
$ns_ run
```