Performance Evaluation of Parallel TCP Variants and Its Impact on Throughput and Fairness in Heterogeneous Networks Based on Test-Bed

BY

MOHAMED A. ALRSHAH

Thesis Submitted to

Faculty of Computer Science and Information Technology,

University Putra Malaysia,

In Fulfillment of the Requirements of the Degree of Master of Computer Science

June 2009

Abstract of thesis presented to the Senate of University Putra Malaysia in Partial fulfillment of the requirements for the Degree of Master of Science

Performance Evaluation of Parallel TCP Variants and Its Impact on Throughput and Fairness in Heterogeneous Networks Based on Test-Bed

By

MOHAMED A. ALRSHAH

June 2009

Supervisor: Associated Professor Dr Mohamed Othman

Faculty: Faculty of Computer Science and Information Technology

It has been argued that single Transport Control Protocol (TCP) connection with proper modification can emulate and capture the robustness of parallel TCP and can well replace it. In this work, a test bed experiment-based Comparison between Single-Based TCP and parallel TCP has been conducted to show the differences of their performance measurements such as throughput performance, loss ratio and TCP-Fairness.

In this experiment, Reno, Scalable, HSTCP, HTCP and CUBIC TCP variants, have been involved to show the differences in their performance and bandwidth utilization. On the other hand, the Link sharing Fairness has

been observed to show the impact of using Parallel TCP of TCP variants with the existing single-based TCP connections.

The results of this experiment reveal that: First: parallel-TCP strongly outperforms single-based TCP in terms of bandwidth utilization and fairness. Second: CUBIC-TCP achieved better performance than Reno, Scalable, Htcp and HStcp and for this reason it is (CUBIC) used as the default of Linux TCP variant in the latest versions of Linux like Fedora 10, 11 and openSUSE 11, 11.1.

Abstrak tesis yang dikemukakan kepada Senat Universiti Putra Malaysia bagi memenuhi keperluan ijazah Master Sains

Pernilaian ke atas Prestasi bagi Berlainan Jenis TCP yang Selari dan Pengaruhnya ke atas Prestasi dan Keadilan dalam Rangkaian yang Berlainan Berdasarkan Test-Bed

Oleh

MOHAMED A. ALRSHAH

June 2009

Pengerusi: Associated Professor Dr Mohamed Othman

Fakulti: Sains Komputer dan Teknologi Maklumat

Telah dipertikaikan bahawa penyambungan tunggal Transport Control Protocol(TCP) dengan modifikasi yang betul dapat mencontohi dan menawan keteguhan TCP selari serta dapat menggantikannya. Dalam thesis ini, suatu eksperimen berbentuk perbandingan diantara TCP berpangkalan tunggal dan TCP selari telah dijalankan untuk menunjukkan perbezaan ukuran prestasi seperti prestasi daya pemprosesan, nisbah kehilangan and keadilan TCP.

Dalam eksperimen ini, pembolehubah Reno, Scalable, HSTCP, HTCP dan CUBIC TCP telah digunakan untuk menunjukkan perbezaan prestasi

dan penggunanan lebar jalur. Disamping itu, perkongsian hubungan keadilan telah diperhatikan untuk menunjukkan impak penggunaan TCP selari dari pembolehubah TCP dengan wujudnya perhubungan TCP berpangkalan tunggal.

Hasil keputusan daripada eksperimen ini menunjukkan : Pertamanya, TCP selari dengan kuat mengatasi TCP berpangkalan tunggal dalam konteks penggunaan lebar jalur dan keadialn. Keduanya pula, CUBIC-YCP mencapai keputusan yang lebih berkesan daripada Reno, Scalable, HTCP dan HSTCP, oleh itu CUBIC digunakan sebagai keingkaran bagi pembolehubah LINUX TCP dalam versi terkini LINUX seperti Fedora10, 11 dan openSUSE 11, 11.1.

ACKNOWLEDGMENTS

First, Alhamdulillah, I would like to express my thanks and gratitude to Allah S.W.T, the most beneficent and the most merciful, whom granted me the ability to complete this thesis.

Thanks also to all my colleagues, lecturers and special thanks for my supervisor Associated Professor Dr Mohamed Othman for his continuous assistance, encouragement, advice and support to make this thesis project successful. He has been an ideal supervisor in every respect, both in terms of technical advice on my research and in terms of professional advice. Special dedications to all of my friends who have been supportive and understanding in ways that they only they knew how. Only God knows how to reciprocate them.

I owe a special debt of gratitude to my parents and family. They have, more than anyone else, been the reason I have been able to get this far. Words cannot express my gratitude to my parents, my brother and my sisters who give me their support and love from across the seas. They instilled in me the value of hard work and taught me how to overcome life's disappointments. My wife and my son (Jehad) give me their selfless support and love that make me want to excel. I am grateful to them for enriching my life. I thank you all so much.

APPROVAL SHEET

Thesis submitted to the Senate of University Putra Malaysia and has been accepted as fulfillment of the requirement for the degree of Master of Science.

……..

Associated Professor Dr Mohamed Othman

Dept of Communication Technology and Network

Faculty of Computer Science & Information Technology

University Putra Malaysia

Date:……

DECLARATION FORM

I hereby declare that the thesis titled “Performance Evaluation of Parallel TCP Variants and Its Impact on Throughput and Fairness in Heterogeneous Networks Based on Test-Bed” is based on my original work except for the quotations and citations have been acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at University Putra Malaysia or at other Institutions.

……..

MOHAMED A. ALRSHAH

Date:……

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATION

TCP	Transport Control Protocol
F	TCP Fairness Index
BDP	Bandwidth Delay Product
HTCP	Hamilton TCP
HSTCP	High Speed TCP
LFN	Low Fat Networks
ACK	TCP Acknowledgment
AIMD	Additive-Increase/Multiplicative-Decrease Algorithm
RFC	Request for Comment
UTP	Unshielded Twisted Pair

CHAPTER 1

INTRODUCTION

Achieving acceptable levels of TCP performance on high-speed wide area networks is very difficult. Poor TCP performance over wide area networks is caused by many factors that disrupt the mechanisms used by TCP to probe and utilize available network capacity. Over the years, significant resources have been invested to attempt to solve network performance problems. A common approach is to overprovision the network infrastructure to eliminate structural bottlenecks. In practice, however, simply eliminating potential sources of network congestion has failed to completely solve performance problems, and the resulting infrastructure often does not meet performance expectations.

The ability to quickly move large amounts of data over a shared wide area network is a necessity for many applications today. The Atlas project, for example, must be able to move many petabytes of data per year from a particle detector located at CERN in Switzerland to the United States. Multiuser collaborative environments that combine visualization, video conferencing, and remote application steering require low network latency and high throughput. The Optiputer project aims to build a distributed high performance computer using wide area optical networks as the system

backplane. Other tools such as GridFTP, bbcp, DPSS, and PSockets are used by applications that need to move large amounts of data over wide area networks [1].

Many of these applications use the Transmission Control Protocol (TCP) for accurate and reliable in-order transmission of data. TCP relies on the congestion avoidance algorithm to measure the capacity of a network path, fairly share bandwidth between competing TCP streams, and to maximize the effective use of the network.

In an attempt to solve performance problems, applications are increasingly relying on aggressive network protocols that can substantially improve throughput, but do so at the expense of unfairly appropriating network capacity from other applications. It is not clear that these aggressive protocols can successfully cooperate with existing network protocols to prevent congestion collapse or excessive levels of network congestion.

On a shared network, the rewards for aggressive behavior are not balanced with penalties for misbehavior that would encourage fair-sharing of network bandwidth. This creates a Tragedy of the Commons situation, in which one application's net gain results in a net loss borne by the community of users that choose to act cooperatively. Thus, the problem of providing

mechanisms for reliable high throughput transmission on shared networks that can overcome the limitations of TCP congestion avoidance and fairly share limited network resources is an important problem that needs to be solved.

1.1. TCP Congestion Avoidance Background

The TCP congestion avoidance algorithm was designed to operate as a distributed control system in which individual TCP streams have no knowledge of the state of other TCP streams, or knowledge of the state of the network over which it operates. It is arguably the most widely deployed and utilized distributed algorithm ever developed. The goals of the congestion avoidance algorithm are to prevent network congestion collapse and to fairly distribute limited network bandwidth resources to competing TCP streams. The TCP congestion avoidance algorithm operates by slowly increasing the transmission rate of packets to "probe" network capacity. The number of packets "in-flight" on the network between the sender and receiver is called the Congestion Window (or cwnd) of the TCP session. The only explicit feedback provided to the congestion avoidance control system is the detection of a lost packet by the TCP sender (packet drop), which indicates that a hop in the network path between the TCP sender and receiver is overloaded.

There are two probing phases in the congestion avoidance algorithm. The initial phase (named Slow Start) rapidly increases the number of packets in-flight by doubling cwnd every round trip time, or by one packet for every received packet acknowledgement (ACK). Once cwnd has reached a predetermined threshold (ssthresh), the algorithm enters the Linear Increase phase, in which the size of cwnd is increased by one packet every round trip time. The congestion avoidance algorithm reacts to a packet drop event by halving the number of packets it allows to be "in-flight" on the network path between the TCP sender and receiver. To recover from a lost packet, the congestion avoidance algorithm increases the congestion window by one packet for every new data packet acknowledged by the receiver [1].

Since this process is driven by the receipt of ACKs from the receiver, the rate of increase of cwnd (the number of packets allowed to be in-flight) is "clocked" by the time delay between the transmission of a data packet and the reception of the ACK for the packet. If the TCP sender and receiver are connected by a network path with a very short packet round trip time (RTT), the rate of increase of cwnd will be relatively high. However, in the case of transcontinental or global network paths, RTT is very high, which leads to a much slower rate of recovery from loss. Consequently, on wide area networks, packet loss has a significant impact on the overall throughput of a TCP session [1].

1.2. Relationship Between Packet Loss and TCP Performance

The implication of the relationship between packet loss and TCP performance is that, if the source of any packet loss is not due to network congestion, the number of non-congestion losses over a period of time could be large enough to adversely affect TCP performance. The Mathis and Padhye TCP bandwidth estimation equations state that, TCP throughput is inversely proportional to the square root of the packet loss rate. Because of this relationship, TCP performance over high-speed networks requires incredibly low non-congestion packet loss rates for the congestion avoidance algorithm to successfully probe network capacity. For example, using the Mathis equation, the packet loss rate must be $\leq 0.0018\% \simeq \frac{2}{100{,}000}$ packets to allow a TCP stream to utilize at least 2/3 of a 622 Mbps OC-12 ATM link. Floyd found that the maximum permitted IEEE bit error rate (BER) for a fiber optic line is large enough to prevent a TCP stream from ever making full use of a 10 Gbps Ethernet network over a transoceanic link [1].

The sensitivity of the congestion avoidance algorithm to non-congestion packet loss is due to several factors. First, there is no explicit flow control in the IP layer. If there was a mechanism by which routers could explicitly send a "slow down" signal back to a TCP sender, the congestion avoidance algorithm could react by reducing the transmission

rate. Explicit Congestion Notification (ECN) is a modification to the Internet Protocol (IP) proposed in 1994 to provide flow control for IP. ECN has not been widely deployed, and requires the universal deployment of a modified TCP sender that responds to ECN signals. Second, the congestion avoidance algorithm assumes that the rate of packet loss from causes other than congestion is very low. The third source of sensitivity to non-congestion packet loss is the fixed maximum packet frame size of 1500 bytes, which is the largest frame size supported by most network devices. Some gigabit ethernet equipment supports large "jumbo frame" packets of 9k, and there are efforts within the community to increase the maximum frame size to even larger values. Thus, the capacity and speed of modern networks has reached the point where non-congestion packet loss has become a significant factor in TCP performance [1].

1.3. Approaches to Solve TCP Performance Problems

An approach commonly used to solve TCP performance problems is to create multiple TCP streams to simultaneously transmit data over several sockets between an application server and client. Grossman developed an application library (PSockets) that can be used by an application to stripe data transmissions over a set of parallel TCP streams. The use of parallel TCP streams has also been adopted by GridFTP, MulTCP, bbcp, DPSS, and

other high performance data intensive applications. Parallel TCP is an aggressive approach that can overcome the effects of non-congestion loss, but it does so at the expense of unfairly appropriating bandwidth from competing TCP streams when there is limited available network capacity. Other aggressive approaches to improve performance have been proposed, but all of them suffer from the same problem: effectiveness is increased, but at the expense of fairness when the network is fully utilized [1].

1.4. Problem Statement

After the fast growth of communication devices and the increasing of network heterogeneity, standard TCP protocol becomes not able to fully utilize the high-speed network links (Bandwidth). Moreover, the variety of TCP protocols made some of confusion for system administrators, so, couple of test bed experiments has to be conducted on TCP variants to compare their performance measurements. TCP variants have to be examined in many environments such as wire and wireless networks, to show which TCP variant deserves to be used in high-speed networks.

It is feasible and valuable to build a network protocol based on parallel TCP sessions for data intensive applications, which effectively uses

unused network bottleneck bandwidth and maintains fairness over effectiveness when the network bottleneck is fully utilized.

1.5. Research Objectives

The goal of the work in this thesis is, to develop a new parallel TCP algorithm to solve TCP performance problems and to effectively utilize high-speed network links. Moreover, this algorithm will not unfairly appropriate bandwidth from other connections when the network is fully utilized. The main objectives in this research are itemized as following:

- Build a new network protocol in application level, based on multiple TCP sessions for data intensive applications.
- Develop a new scheme for test bed experiment to examine the proposed parallel TCP protocol using different TCP variants that are Reno, Scalable, HTCP, HSTCP and CUBIC to show its performance improvement.
- Show the impact of using parallel TCP on TCP fairness between the competitive single and parallel based TCP connections.

1.6. Research Scope

This research is only focuses on the following points:

- This work focuses only on wire networks while wireless networks are not considered.
- A single-bottleneck topology has been examined while multi-bottleneck has not been considered.
- This experiment is only focuses on these high-speed TCP variants that are Reno, Scalable, HTCP, HSTCP and CUBIC while the others have not considered.
- It focuses only on one type of traffic, which is standard Poisson traffic while the other types of traffic have not studied.

1.7. Organization of Thesis

This thesis organized into six chapters including the introductory chapter. The rest of the chapters in this thesis are as follows:

Chapter 2 gives a brief discussion about the solutions that have been used in the related work (Literature review). In addition, it explains the behaviour of parallel TCP algorithm and shows its unique merits.

Chapter 3 describes the methodology used in this research. The proposed algorithm and the work scheme then network topology, test-bed experiment configurations and parameters, and performance metrics.

Chapter 4 presents the proposed algorithm implementation. Moreover, it explains the way that used to collect and analyze the data, and the way to evaluate the performance of the algorithms that used in this work.

Chapter 5 presents the test-bed experiment results and analysis.

Chapter 6 concludes the overall study of this research, and future works are presented.

CHAPTER 2

LETERATURE REVIEW

2.1. Introduction

This chapter will explain TCP protocol and describes its variants, then, will explain parallel TCP and its existent implementations, finally will explain the main ideas (such as the utilization of multi-route, fast recovery and security issue) that will cause by TCP parallelization.

2.2. The Transmission Control Protocol

TCP is one of the core protocols of the Internet Protocol Suite. TCP was one of the two original components, with Internet Protocol (IP), of the suite, so that the entire suite is commonly referred to as TCP/IP. Whereas IP handles lower-level transmissions from computer to computer as a message makes its way across the Internet, TCP operates at a higher level, concerned only with the two end systems, for example, a Web browser and a Web server [2].

In particular, TCP provides reliable, ordered delivery of a stream of bytes from a program on one computer to another program on another computer. Besides the Web, other common applications of TCP include e-

mail and file transfer. Among its other management tasks, TCP controls message size, the rate at which messages are exchanged, and network traffic congestion.

2.3. TCP Variants

2.3.1. TCP Reno

To avoid congestion collapse, TCP uses a multi-faceted congestion control strategy. For each connection, TCP maintains a congestion window, limiting the total number of unacknowledged packets that may be in transit end-to-end. This is somewhat analogous to TCP's sliding window used for flow control. TCP uses a mechanism called slow start to increase the congestion window after a connection is initialized and after a timeout. It starts with a window of two times the maximum segment size [3, 4].

Although the initial rate is low, the rate of increase is very rapid: for every packet acknowledged, the congestion window increases so that for every round trip time (RTT), the congestion window has doubled. When the congestion window exceeds a threshold ssthresh, the algorithm enters a new state, called congestion avoidance. In some implementations (e.g., Linux), the initial ssthresh is large, and so the first slow start usually ends after a loss.

However, ssthresh is updated at the end of each slow start, and will often affect subsequent slow starts triggered by timeouts [3, 4].

Congestion avoidance: As long as non-duplicate ACKs are received, the congestion window is additively increased by one MSS every round trip time. When a packet is lost, the likelihood of duplicate ACKs being received is very high (it is possible though unlikely that the stream just underwent extreme packet reordering, which would also prompt duplicate ACKs). The behaviour of Reno: If three duplicate ACKs are received (i.e., three ACKs acknowledging the same packet, which are not piggybacked on data, and do not change the receiver's advertised window), Reno will halve the congestion window, perform a "fast retransmit", and enter a phase called Fast Recovery. If an ACK times out, slow start is used [3, 4].

In the state of Fast Recovery, TCP Reno retransmits the missing packet that was signaled by three duplicate ACKs, and waits for an acknowledgment of the entire transmit window before returning to congestion avoidance. If there is no acknowledgment, TCP Reno experiences a timeout and enters the slow-start state. This algorithm reduces congestion window to one MSS on a timeout event [3, 4].

2.3.2. Scalable TCP

Scalable TCP is a simple change to the traditional TCP congestion control algorithm (RFC2581) which dramatically improves TCP performance in high-speed wide area networks. Scalable TCP changes the algorithm to update TCP's congestion window to the following:

$$cwnd = \begin{cases} cwnd + 0.01 & \text{for each ack received while not in loss recovery.} \\ 0.875 * cwnd & \text{on each loss event.} \end{cases}$$

Traditional TCP probing times are proportional to the sending rate and the round trip time. However, Scalable TCP probing times are proportional only to the round trip time making the scheme scalable to high-speed IP networks. Scalable TCP algorithm is only used for windows above a certain size. This allows Scalable TCP to be deployed incrementally [3].

2.3.3. Hamilton TCP (HTCP)

HTCP is another implementation of TCP with an optimized congestion control algorithm for high-speed networks with high latency (LFN). It has been created by researchers at Hamilton Institute in Ireland. It is an optional module in recent Linux 2.6 kernels. HTCP is a loss-based algorithm, using additive-increase/multiplicative-decrease (AIMD) to control TCP's congestion window. It is one of many TCP congestion avoidance

algorithms, which seeks to increase the aggressiveness of TCP on high bandwidth-delay product (BDP) paths, while maintaining "TCP friendliness" for small BDP paths [3].

HTCP increases its aggressiveness (in particular, the rate of additive increase) as the time since the previous loss increases. This avoids the problem encountered by HSTCP and BIC TCP of making flows more aggressive if their windows are already large. Thus, new flows can be expected to converge to fairness faster under HTCP than HSTCP and BIC TCP [3].

A side effect of increasing the rate of increase as the time since the last packet loss increases is that flows which happen not to lose a packet when other flows do, can then take an unfair portion of the bandwidth. Techniques to overcome this are currently in the research phase [3].

2.3.4. High Speed TCP (HSTCP)

HSTCP is a new congestion control algorithm protocol defined in RFC 3649 for TCP. Standard TCP performs poorly in networks with a large bandwidth delay product. It is unable to fully utilize available bandwidth. It

makes minor modifications to standard TCP's congestion control mechanism to overcome this limitation. When an ACK is received (in congestion avoidance), the window is increased by a(w)/w and when a loss is detected through triple duplicate acknowledgments, the window is decreased by (1 − b(w))w, where w is the current window size. When the congestion window is small, HSTCP behaves exactly like standard TCP so a(w) is 1 and b(w) is 0.5. When TCP's congestion window is beyond a certain threshold, a(w) and b(w) become functions of the current window size [3].

In this region, as the congestion window increases, the value of a(w) increases and the value of b(w) decreases. This means that HSTCP's window will grow faster than standard TCP and also recover from losses more quickly. This behavior allows HSTCP to be friendly to standard TCP flows in normal networks and also to quickly utilize available bandwidth in networks with large bandwidth delay products. In addition, its slow start and timeout behavior is exactly like standard TCP [3].

2.3.5. CUBIC

CUBIC is an implementation of TCP with an optimized congestion control algorithm for high-speed networks with high latency (LFN). It is a less aggressive and more systematic derivative of BIC TCP, in which the

window is a cubic function of time since the last congestion event, with the inflection point set to the window prior to the event. Being a cubic function, there are two components to window growth. The first is a concave portion where the window quickly ramps up to the window size before the last congestion event. Next is the convex growth were CUBIC probes for more bandwidth, slowly at first then very rapidly. CUBIC spends a lot of time at a plateau between the concave and convex growth region, which allows help the network stabilize before CUBIC begins looking for more bandwidth [3].

Another major difference between CUBIC and standard TCP flavors is that, it does not rely on the receipt of ACKs to increase the window size. CUBIC's window size is dependent only on the last congestion event. With standard TCP, flows with very short RTTs will receive ACKs faster and therefore have their congestion windows grow faster than other flows with longer RTTs. CUBIC allows for more fairness between flows since the window growth is independent of RTT. It is implemented and used by default in Linux kernels 2.6.19 and above [3].

2.4. Parallel TCP

The concept of Parallel TCP is not new and its original form is the use of a set of multiple standard TCP connections was used in late 80s and early/mid 90s to overcome the limitation on the TCP window size in high-speed networks and highly dynamic [5, 6, 7].

More recently, there has been a focus on improving the performance of data intensive applications, such as GridFTP [8, 9] and PSockets [10, 11]. These solutions focus on the use of multiple standard TCP connections to improve bandwidth utilization. However, these studies did not compare or identify the differences of the performance between the use of a set of multiple standard TCP connections and a single connection emulating a set of multiple standard TCP connections.

Some solutions use a set of parallel connections to outperform the single standard TCP connection in terms of congestion avoidance and maintaining fairness. The approaches described in [12, 13] adopt integrated congestion control across a set of parallel connections to make them as aggressive as a single standard TCP connection. It is shown that, the approaches are fair and effective to share bandwidth. In the references [14, 15], by using a fractional multiplier the aggregate window of the parallel

connections increases less than one packet per RTT. Only the involved connection halves its window when a packet loss is detected. Similar to Parallel TCP, pTCP uses multiple TCP Control Blocks and it shows that pTCP outperforms single connection or single TCP based approach such as MulTCP [16].

Moreover, single-based TCP was designed for connections that traverse a single path between the sender and receiver. However, multiple paths can be used by a connection simultaneously in several environments. They consider the problem of supporting striped connections that operate over multiple paths, by proposing an end-to-end transport layer protocol called pTCP that allows connections to enjoy the aggregate bandwidths offered by the multiple paths, irrespective of the individual characteristics of the paths [17].

On the other hand, some solutions have the capability of using a single TCP connection to emulate the behaviour of Parallel TCP. MulTCP [16] makes one logical connection behave like a set of multiple standard TCP connections to achieve weighted proportional fairness. The recent development on high performance TCP has resulted in TCP variants such as Scalable TCP [18], HSTCP [19], HTCP [20], BIC [21], CUBIC [22], and FAST

TCP [23]. All of these TCP variants have the effect of emulating a set of parallel TCP connections.

It has been discussed that, a single based TCP connection with proper modification can emulate the parallel TCP and thus can well replace Parallel TCP. The existing single-based TCP will not be able to achieve the same effects as the parallel TCP especially in heterogeneous networks combined with high-speed wireless access links where the packet losses prevail and are less predictable [24].

Kelly proposed Scalable TCP (STCP) [18]. The design objective of STCP is to make the recovery time from loss events be constant regardless of the window size. This is why it is called “Scalable”. Note that the recovery time of TCP-NewReno largely depends on the current window size. HighSpeed TCP (HSTCP) [25] uses a generalized AIMD where the linear increase factor and multiplicative decrease factor are adjusted by a convex fucntion of the current congestion window size. When the congestion window is less than some cutoff value, HSTCP uses the same factors as TCP. Most of high-speed TCP variants support this form of TCP compatibility, which is based on the window size. When the window grows beyond the cutoff point, the convex function increases the increase factor and reduces the decrease factor proportionally to the window size.

Hamilton TCP (HTCP) [20], like CUBIC, uses the elapsed time (Δ) since the last congestion event for calculating the current congestion window size. The window growth function of HTCP is a quadratic function of (Δ). HTCP is unique in that it adjusts the decrease factor by a function of RTTs which is engineered to estimate the queue size in the network path of the current flow. Thus, the decrease factor is adjusted to be proportional to the queue size.

However, the main problem of using a single connection emulating a set of multiple standard or modified TCP connections is when an ack received, the aggregated window size will increase by certain number of packets based on the TCP variant which is used, and it can quickly grow but when timeout detected the aggregated window size will be decreased to the half of the previous window this will affect the throughput.

While in parallel of multiple standard or modified TCP connections, each connection work separately from the concurrent connections, when one of them detects a timeout it will only decrease the window size of the involved connection while the other concurrent connections will keep their window increase until the timeouts detected. This highly improves the

throughput and makes the TCP connections behave fairly with each other as in single TCP.

This study concerns high-speed TCP variants, specifically those are implemented and widely available in Linux kernel. There is growing interest in more widespread uses of these TCP variants as parts of the user community begin to use more demanding applications (e.g. data-intensive Grid applications). Users and especially administrators, however, are concerned with the impact of these variants would have on the network. This is because they adopt different congestion control algorithms to normal standard TCP algorithms that are potentially more aggressive in their transmission behaviour. This is by design: the goal is to make use of available network capacity that standard TCP cannot utilize effectively due to its relatively conservative congestion control behaviour.

The use of congestion control in TCP has been the key to the Internet's stability, so any change to this behaviour merits investigation. In order to investigate this behaviour, researchers may naturally turn to simulation with ns-2 or other simulation tools in order to generate a set of experiments that are easily manageable, scalable, configurable and reproducible for their specific scenarios of interest, as reproducing and measuring those scenarios at scale in a test bed may not be feasible. For aforementioned reasons, this

kind of experiments has to be conducted to show the performance of TCP variants.

Parallel TCP uses a set of parallel (modified or standard) TCP connections to transfer data in an application process. With standard TCP connections, Parallel TCP has been used for effectively utilize bandwidth for data intensive applications over high bandwidth-delay product (BDP) networks. On the other hand, it has been argued that a single TCP connection with proper modification can emulate and capture the robustness of parallel TCP and thus can well replace it.

From the implementation of parallel TCP, it has been found that, the single-based TCP (such as HSTCP) may not be able to achieve the same effects as parallel TCP, especially in heterogeneous and highly dynamic networks. Parallel TCP achieves better throughput and performance than the single-connection based approach [24]. The applications that require good network performance often use parallel TCP streams and TCP modifications to improve the effectiveness of TCP. If the network bottleneck is fully utilized, this approach boosts throughput by unfairly stealing bandwidth from competing TCP streams. To improve the effectiveness of TCP is much more easier compared to improve the effectiveness while maintaining fairness [14].

2.5. Fast Recovery in Single-Based TCP

Assume that, the periodic loss event has been used; the evolution of congestion window for a single connection when Fast Recovery is taken into account will be as shown in figure 2.1 below. Which means that, after timeout detected; AIMD will halve its congestion window. This will affect the whole throughput of the connection and it will take a long time to reach the maximum congestion window again.

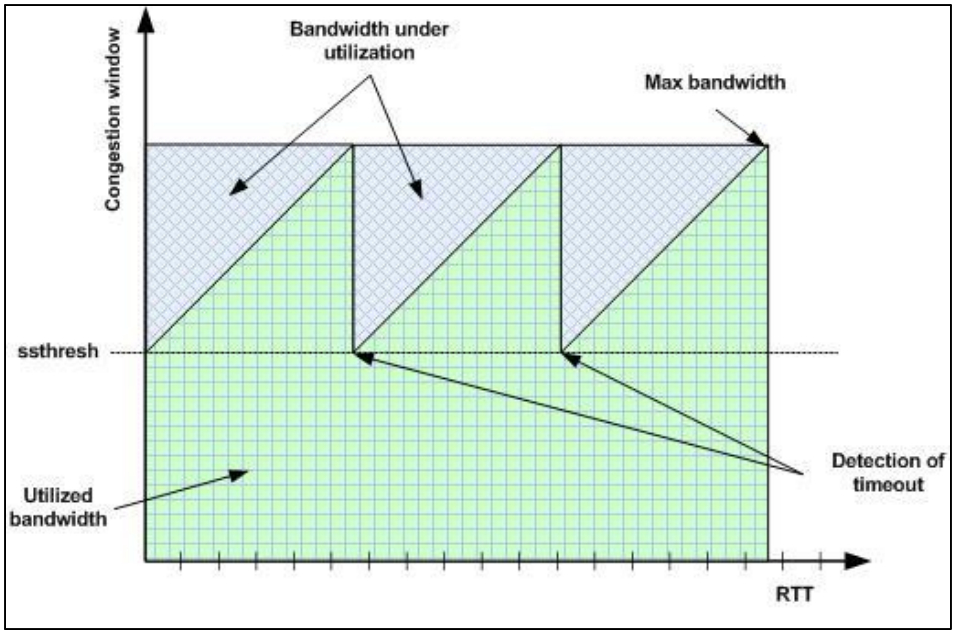

Figure 2.1: Evolution of congestion window in single-based TCP

As shown in figure 2.1 the green colored area reflects the throughput of the connection while the gray colored area reflects under utilization area. It is very clear that, detection of one timeout signal can reduce the congestion

window to 50%. This considered as a problem of waste of resources, this problem has been partially solved in parallel TCP and it will be explained in next section.

2.6. Fast Recovery in Parallel TCP

With the same assumptions in the previous section, Figure 2.2 shows the evolution of the congestion window for three parallel connections.

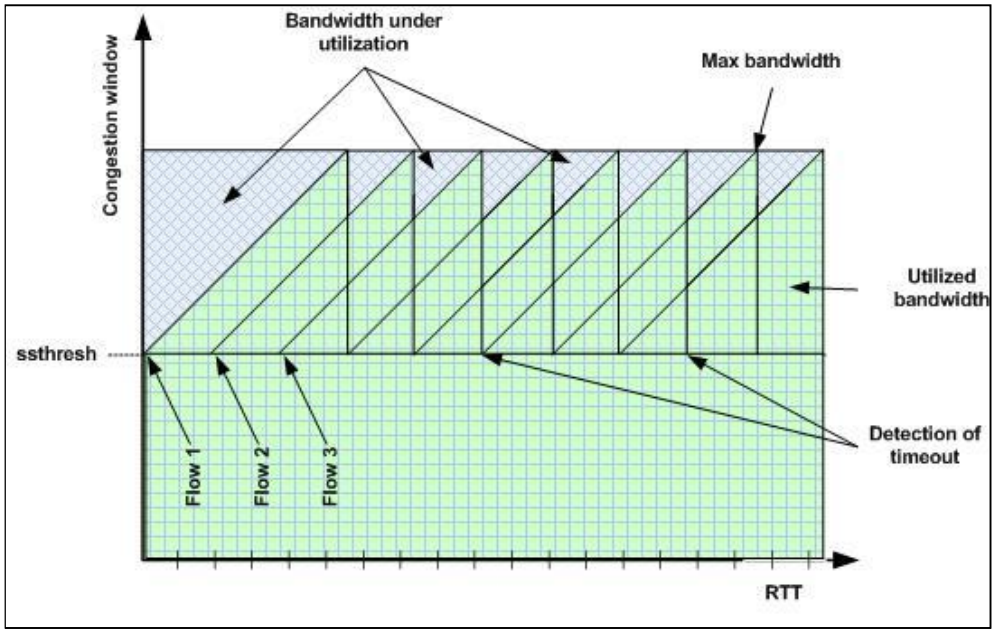

Figure 2.2: Evolution of congestion window in parallel TCP

It is well clear that, the unutilized area of the flow 1 filled by the utilized area of the other concurrent flows (flow 2 and flow 3). Means that, the detection of timeout in one connection will decrease the congestion

window of the involved connection only while the other concurrent connections will not be affected and they will continue in their congestion window increasing until the timeouts detected. The main two reasons to make parallel TCP behave in this way are the serialization of connections establishments, and the independency of parallel connections.

Assume that, the available link bandwidth was only one Mbps, and there are three connections share this link; also assume that, the bandwidth was equally divided between the connections, which mean 0.33 Mbps for each connection. If a timeout detected on flow 1 it will decrease its window to the half, which is around 0.16 Mbps, while the congestion windows of the others will stay as they are (0.33 Mbps for each). The aggregation window for these three concurrent connections after packet loss detection will be as following:

$$\text{Aggregated window} = CW_{flow1} + CW_{flow2} + CW_{flow3} \qquad (2.1)$$

$CW_{flow1} \simeq 0.16\ Mbps$ After timeout detected.

$CW_{flow2} \simeq 0.33\ Mbps$

$CW_{flow3} \simeq 0.33\ Mbps$

$\text{Aggregated window} \simeq 0.16 + 0.33 + 0.33 \simeq 0.82$ Mbps

While the reduction of the congestion window in parallel of three connections will be as following:

$$\text{Reduction percentage} \simeq \frac{\text{Maximum ACW} - \text{Current ACW}}{\text{Maximum ACW}} \quad (2.2)$$

$$\text{Reduction percentage} \simeq \frac{1\ \text{Mbps} - 0.82\ \text{Mbps}}{1\ \text{Mbps}} \simeq \frac{0.18}{1} \simeq 18\%$$

While, ACW is Aggregated Congestion Window of parallel connections and CW is Congestion Window of single connection.

2.7. TCP Fairness

Fairness measure or metric is used in network engineering to determine whether users or applications are receiving a fair share of system resources or not. Congestion control mechanisms for new network transmission protocols or peer-to-peer applications must interact well with the existing TCP variants.

TCP fairness requires that, a new protocol receive no larger share of the network than a comparable TCP flows. This is important as TCP is the dominant transport protocol on the Internet, and if new protocols acquire unfair capacity, they tend to cause problems such as congestion collapse.

This was the case with the first versions of RealMedia's streaming protocol: it was based on UDP and was widely blocked at organizational firewalls until a TCP-based version was developed. There are several mathematical and conceptual definitions of fairness such as Jain's Fairness Index (JFI) [26, 27] as shown in the formula below which has been used in this experiment. Where x_i is the measured throughput for flow $\boldsymbol{i}$, for N flows in the system.

$$\boldsymbol{TCP\ Fairness\ (F)} = \frac{(\sum_{i=1}^{N} x_i)^2}{N(\sum_{i=1}^{N} x_i^2)} \qquad (2.3)$$

Max-min fairness states that small flows receive what they demand and larger flows share the remaining capacity equally. Bandwidth is allocated equally to all flows until one is satisfied, then bandwidth is equally increased among the remainder and so on until all flows are satisfied or bandwidth is exhausted [27].

2.8. Multi-Route Usage

In single-based TCP the connections does not have the ability to use more than one route at single time. During the connection establishment the intermediate routers will chose one route to be used by this connection but if route failure has been detected after certain amount of time, the intermediate routers will change to another route from the available routes. This will ensure the use of single route, this considered as waste of resources problem,

especially when multi-path infrastructure is available, because in some scenarios the chosen route limits the connection capability while there are an alternative routes are free or have not been fully utilized.

Contrarily, the proposed parallel TCP can utilize multi-routes without any modification this resulted by the independency of the parallel connections. Assume that, there are three parallel connections belongs to one application process, during the connections establishments the application will start with the first connection and the intermediate routers will chose one of the available routes to be used by this connection. Then the application will establish the next connection, and the intermediate routers may chose the same route (which already used by the first connection) or another one from the available routes as shown in figure 2.3. This selection of route will be rely on some criterions such as the link utilization, distance, link delay and link cost and so no. The use of multiple routes will increase the utilization of the available resources and thus will increase the throughput of these parallel TCP connections.

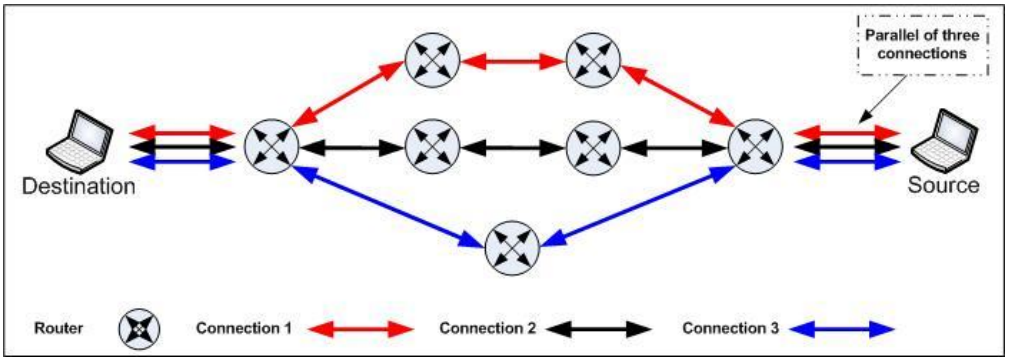

Figure 2.3: Multi-route Usage

2.9. Security Issue

While the using of multi-route in parallel TCP increases the resources utilization and it is better than using single route; both of single route and multi-route increases the security. Using the same example in figure 2.3 the whole of the data will be divided into three chunks, one chunk for each connection (red, black and blue). Each chunk of data will be transferred through independent connection with different characteristics for each, either through the same route or through different routes, so that will make the job of the network sniffers more difficult and will increase the network security.

2.10. Summary

From literature review, it can be summarized that, parallel TCP has its unique characteristics that cannot be emulated by single based TCP. These characteristics are the ability of fast recovery, the ability of using multiple paths to transfer data for single application process, high performance and high fairness.

CHAPTER 3

RESEARCH METHODOLOGY

3.1. Introduction

This chapter gives an overview of the test-bed experiment and its configurations such as network topology, experiment parameters. Then, it follows by description of hardware and software tools that used in the experiment and the structure chosen for the performance evaluation. As it known, there are three techniques for performance evaluation, which are analytical modeling, simulation and test-bed. In this work, a test-bed experiment has been conducted to compare between single-based TCP and the proposed parallel TCP using some high-speed TCP variants. Then it followed by performance metrics.

3.2. Parallel TCP Model

In this thesis, a new parallel TCP algorithm has been proposed and implemented to overcome the problems of the existent algorithms. Parallel TCP algorithms were suggested to increase the throughput of the standard TCP, because Standard TCP could not fully utilize the high-speed links.

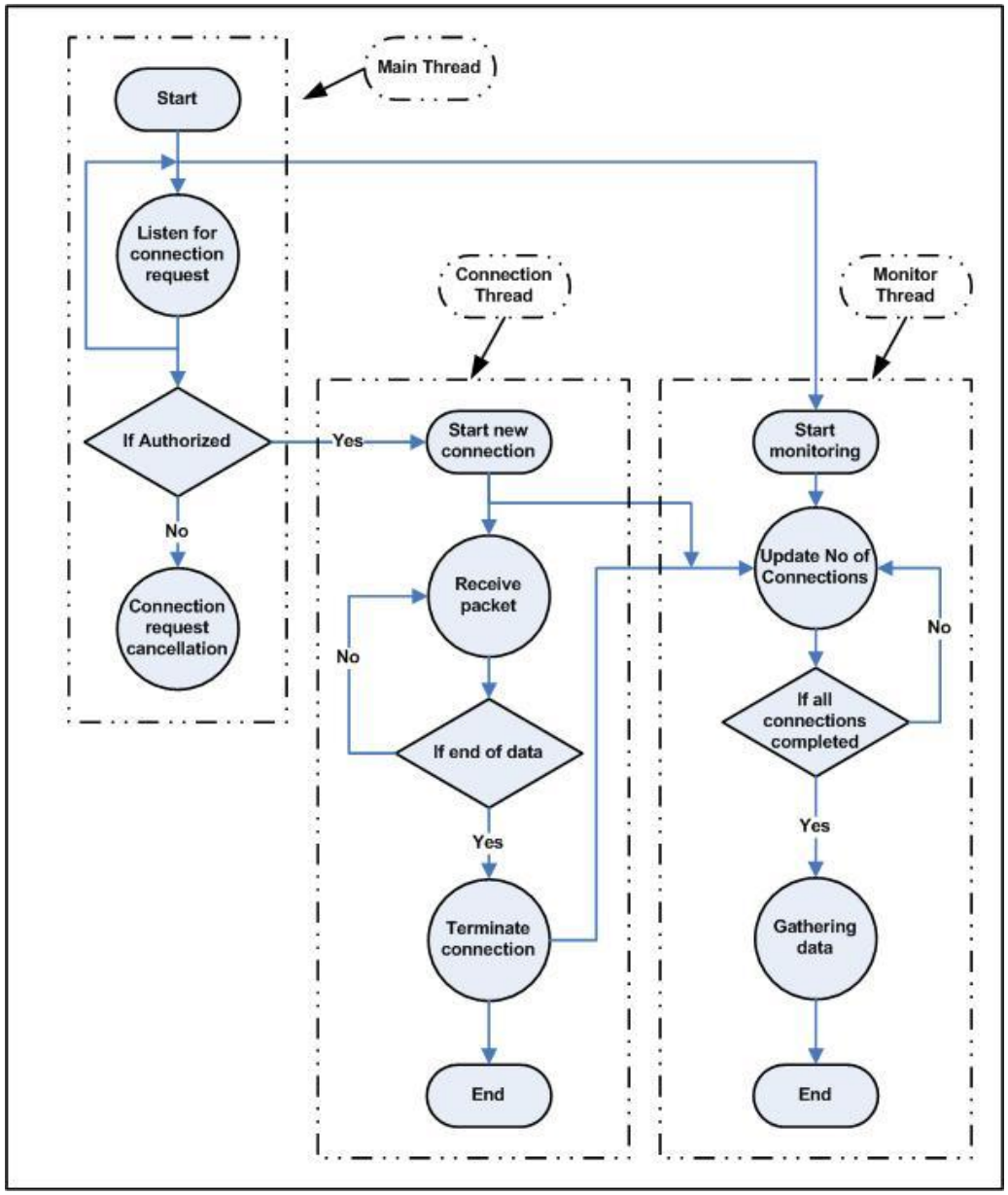

Figure 3.1: The Model of Receiver-Side of the Proposed Parallel TCP

However, after implementing these algorithms, a new problem has been arisen, which is unfairness problem. The proposed algorithm should be able to utilize the high-speed links while maintains fairness. This algorithm

has been implemented in both of sender and receiver sides as shown in figure 3.1 and figure 3.2. One shows the receiver-side model, and the other shows the sender-side model respectively.

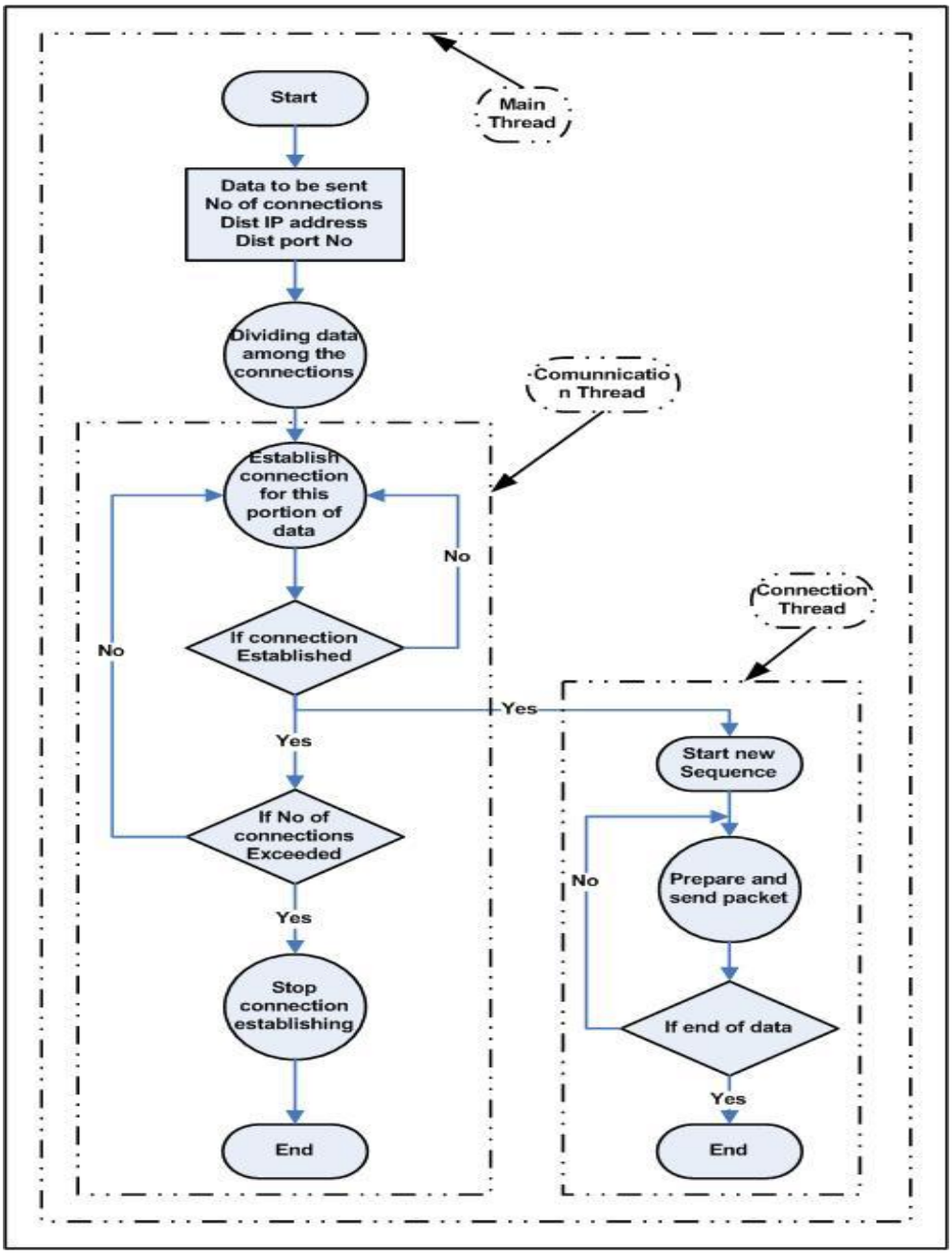

Figure 3.2: The Model of Sender-Side of the Proposed Parallel TCP

Figure 3.1 and figure 3.2 show the data and the control flows of the sender and the receiver algorithms, and they demonstrate that, the proposed algorithm implemented using multi-threading concepts to facilitate and accelerate the operations of the execution.

3.2.1. Receiver-Side

As shown in figure 3.1, the main thread will keep listening for any new connection request on the determined pair of IP address and port number. If any new connection request received and authorized, the main thread will send signal to connection thread manager to establish a new connection thread. The connection thread manager will create a new connection in separated thread and will send a signal to the monitor thread to start its job, which is monitoring the active connections and gathering the data when all the connections are being ended. Each connection thread will start receiving data from the involved sender until the end of the data, and then a termination of connection signal will be sent to the monitor thread to update its state. However, the monitor thread will keep working until it receives the termination signal of the last connection from the parallelized connections set, then it will start gathering and ordering data, consequently save this data into its final form.

3.2.2. Sender-Side

As shown in figure 3.2, the main thread of the sender will start working to send the data to the receiver. First, it will divide the data into chunks based on the number of connections that will be used to transfer this data. Then, it will send all the data chunks to the communication thread to start its job. Communication thread will start creating threads (connection threads) one thread for each chunk of data, and it will give different sequence numbers for each connection. Therefore, each connection thread will start data pipelining which means it will start creating packets and sending it to the receiver. When the end of the data chunk is reached, FIN packet will be sent to the receiver to start connection termination.

3.3. Parallel TCP Scheme

Figure 3.3 below shows the scheme of the proposed parallel TCP and emphasizes that; the application layer will be the first and the last responsible on parallelism in both sides (sender and receiver sides). Consequently, the lower layers do not have the ability to distinguish between the flows (connections) and do not know which flow belongs to which set of parallel connections, which means that, in the lower layers all connections is single-based TCPs.

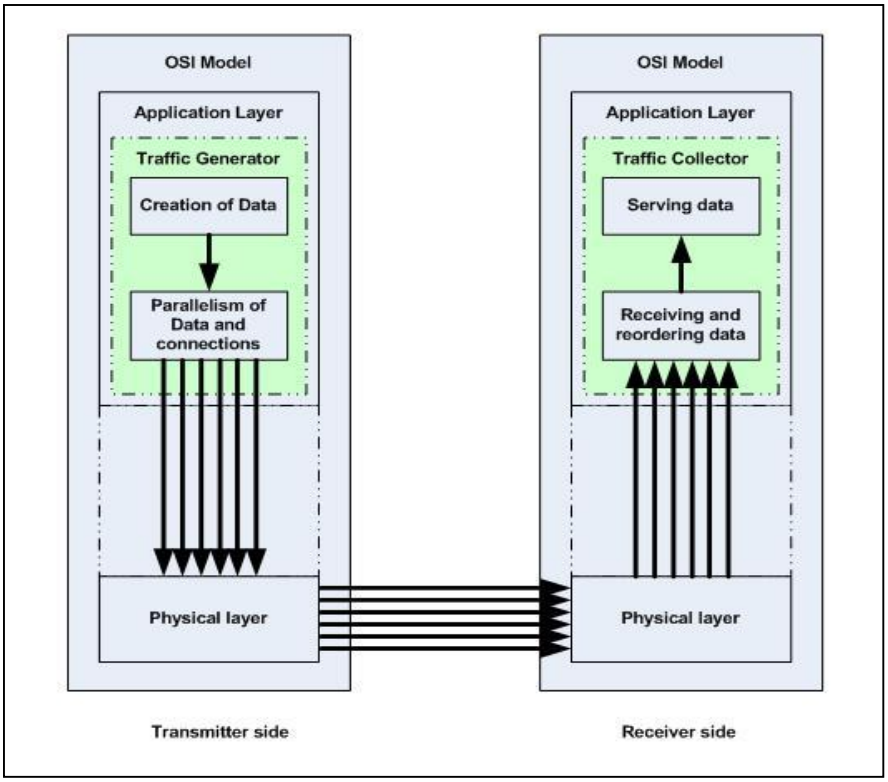

Figure 3.3: diagram of parallel TCP scheme

This isolation of use will not negatively affect the performance of the proposed parallel TCP such as throughput and Fast Recovery but contrarily it will improve them as well as it will not affect the fairness among the competed flows, because it will rely on the behaviour of single-based TCP. Consequently, when the application data being divided into chunks and traveled separately maybe on the same link or on different links, with different information of TCP headers, it will not be easily detected and reassembled except at the end node (destination) which will rely on the agreement of the applications on senders and receivers at the parallelism level.

3.4. Network Topology

The network topology that has been used in this experiment is typical single bottleneck ‘‘***dumbbell***’’ as shown in figure 3.4. This topology has been implemented using six PCs, two of them were used as PC routers, and two as senders while the others was used as receivers.

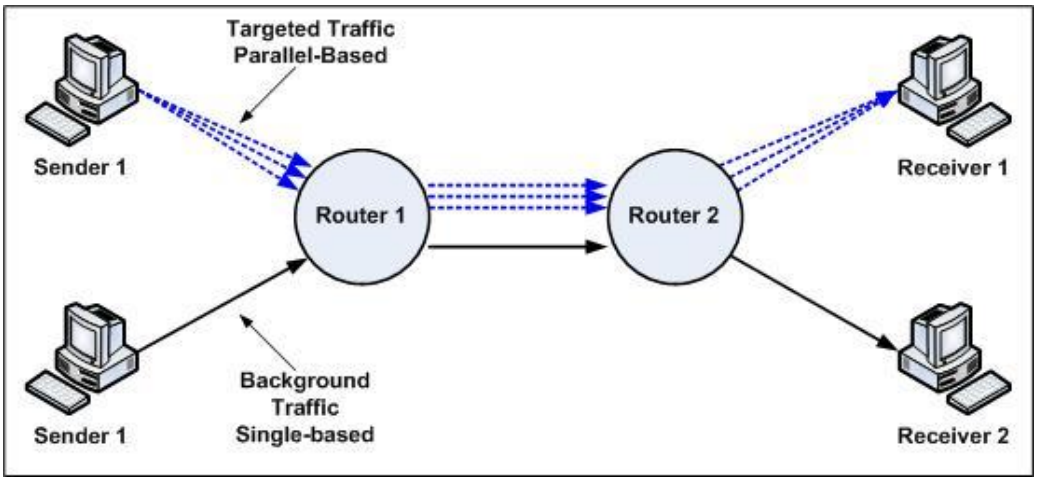

Figure 3.4: Network Topology

The capacity of the links between senders and the router1, and between receivers and Router2 is 100 Mbps, while the capacity of the link that connects the bottleneck routers is 10 Mbps. Moreover, all of the links in this topology are Full-Duplex links, and the traffic type that has been used in this experiment is Standard Poisson distribution [24] and the flow control algorithm that applied on the bottleneck routers is Random Early Drop (RED) algorithm. The packet size in this experiment was 1000 bytes and the end-to-end

buffers were 300 packets. Additionally, next section will shows the experiment parameters as they have been set in this experiment.

3.5. Experiment Parameters

Table 3.1 below shows the test-bed experiment parameters and their values as used in the experiment. This table covers only the important parameters such as the TCP schemes, the traffic type that used in the experiment, the flow control algorithm that used in the bottleneck routers, links capacities, links delay, packet size, buffer size, and the experiment time.

No.	Parameter	Value
1.	TCP Scheme	Reno/Scalable/Htcp/HStcp/CUBIC
2.	Flow Control Algorithm	Random Early Drop (RED)
3.	Link capacity	100 Mbps for nodes and 10 Mbps for bottleneck
4.	Link delay	100 milliseconds
5.	Packet size	1000 bytes
6.	Buffer size	300 packets
7.	Traffic type	Standard Poisson distribution
8.	Experiment time	1000 seconds

Table 3.1: Experiment Parameters

3.6. Hardware and Software Requirements

This section will give a brief overview on hardware software tools that have been exploited in this experiment to provide the desired environment of the experiment with brief explanation about the functionality of each one.

3.6.1. The Hardware

HP Compaq DC7100 computers with Intel Pentium4 3.2 GHz dual core processor and 1GB memory has been used in this experiment to represent the senders, receivers and PC routers. Cat5 UTP cables have been used to connect these computers with each other to represent the aforementioned network topology.

3.6.2. The Operating System

The latest version 11.1 of Linux openSUSE has been used on the nodes (senders and receivers) with Linux kernel version 2.6.27. This operating system uses TCP CUBIC as default TCP variant, while thirteen TCP variants have been implemented in this version of Linux kernel, these TCP variants are Reno, Scalable, HTCP, HSTCP, CUBIC, BIC, Westwood, Vegas, Hybla, Yeah, Illinois, TCP-LP (Low Priority) and Veno.

3.6.3. MONO Framework

MONO framework version 2.0.2 is an open source project supported by NOVELL with the help of Microsoft. Earlier version of this framework had been released to represent the same functionality of Microsoft Dot Net framework. However, the major difference between them is that, Microsoft framework can only be installed on Windows platform, while MONO can be installed on a variety of famous platforms in the world such as Windows, Macintosh, Linux, UNIX, and Solaris. Whatever, this framework was used instead of Microsoft Dot Net framework to provide the libraries which needed by the sender and receiver side applications on Linux openSUSE.

3.6.4. MikroTik Router OS

MikroTik Router OS is Linux based software for router system, which makes it easy for any ISP or network administrator to build low cost PC router for local area, wireless, or wide area networks. It provides a high level of control; by using this software, the data flow control and the routing algorithms are configurable. MikroTik Router OS provides some of flow control algorithms such as RED, DropTail and FIFO. In addition, it provides a variety of routing algorithms such as RIP, OSPF and BGP. Besides, the bandwidth of the installed network interfaces and the service queues of it can be highly controlled. To control this kind of routers, there are three ways:

First, through local access using Linux Sell commands on the PC router itself. Second, through remote access using WinBox tool, which is Windows based software with very nice graphical user interface. Third, through its web based configuration page. However, MikroTik Router OS version 2.9 has been used in this experiment to configure the PC routers to provide a high level of control.

3.6.5. Other Software Tools

TCPdump version 4.0 has been used to monitor the network traffic, then to collect the data and save it into trace files to be analyzed using AWK programming language version 3.1.6. Both of TCPdump and AWK are Linux-based and free tools.

3.7. Performance Metrics

The main point of interesting in this work is to evaluate the performance measurements of TCP variants that are performance throughput, throughput ratio, loss ratio and TCP fairness index (JFI); and to compare between single-based TCP and the proposed parallel TCP through the topology discussed in section 3.4. Following the definitions of these performance metrics to give the readers more understanding:

- Performance throughput: is the amount of data moved successfully from one place to another in a given time period over a physical or logical link.

- Throughput ratio: is the amount of data which have been moved successfully from one source to destination in a given time period over a physical or logical link.

- Loss Ratio: is the amount of retransmitted data from source to destination in a given time period over a physical or logical link.

- Fairness: it is known as Fairness Index which used to determine whether the applications that use the same link or connection are receiving a fair share of bandwidth or not. The Fairness Index is denoted as F, while F is a decimal value resides between zero and one, which represents the percentage of fairness.

3.8. Summary

In this chapter, a new Parallel TCP algorithm has been proposed to solve the problems of the existent algorithms, so the proposed algorithm will be able to fully utilize high-speed network links in both of homogeneous and heterogeneous networks while it will take care and maintain fairness. This algorithm will be evaluated by using test-bed experiment to compare among TCP variants in both cases of single and parallel.

CHAPTER 4

DESIGN AND IMPLEMENTATION

This chapter will give the reader a brief explanation on the experiment design and implementation. It will start by the implementation of the proposed algorithm and how it has been developed, then it will explain how to tune the operating system kernel (Linux openSUSE) to play with the congestion control algorithms and how to collect the trace files and how to analyze it.

4.1 The Implementation of the Proposed Algorithm

Traffic Generator (TG) and Traffic Collector (TC) have been built from scratch using Microsoft C# and Microsoft DotNet framework v3.5. Figure 4.1 shows the Traffic Collector (TC), which has been used to collect the traffic that has been generated by Traffic Generator (TG), while figure 4.2 shows the Traffic Generator (TG), which has been used to provide a traffic. Refer to Appendix A and B for the source code.

To start the operation of generating traffic, first, TC has to be started and then TG can generate traffic and thus send it to TC. To start TC, the local IP address of the computer where TC is installed and the port number have

to be set through the user interface, this parameters will be used by TC to listen for the data that arrives from TG to this pair of IP address and port number.

After IP address and port number being setup, “Start Listening” button has to be pressed to start TC which is the receiver. As noted above, there is no setting for the number of connections that will be established between TC and TG because it is not the responsibility of TC. TC must be ready for accepting any number of connections if it is not exceeded the maximum number of allowed connections.

While TC waiting for the new connections requests from TGs, there are some parameters have to be setup on TG user interface such as the IP address and port number of TC and the number of connections has to be set as well then just press “Go” button to establish the connections and thus send the data to TC.

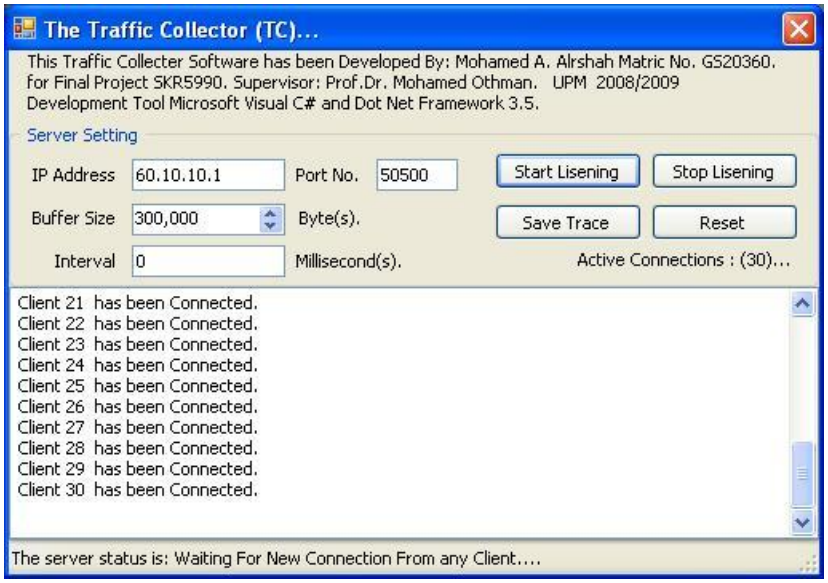

Figure 4.1: Traffic Collector

To start using TC and TG just go to terminal and run the proper command which is either “mono MyServer.exe” to run TC program, or “mono MyClient.exe” to run TG program. Otherwise, these programs cannot run especially if mono does not installed on that computers.

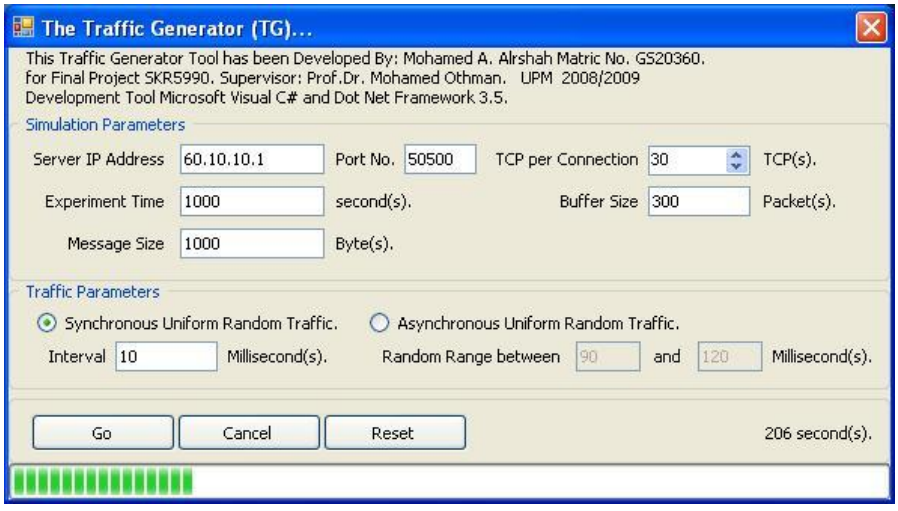

Figure 4.2: Traffic Generator

4.2 Tuning of The Operating System

To tune the Linux kernel at run time, the parameters in this folder /proc/sys/net/ipv4/ have to be modified. In this folder, there are many files each file named by the relative parameter name. To read the value of any parameter the following Shell command has to be run in the terminal:

```
[root@localhost ipv4]# cat tcp_congestion_control
CUBIC
[root@localhost ipv4]#
```

While "***tcp_congestion_control***" is the parameter name (file name) and "***cat***" is a Shell command means read. After running this command the result that is the current value of the selected parameter will be printed directly which is "***CUBIC***" in this example. Likewise, to set any value to any parameter just use "***echo***" command followed by valid value as shown in the command below:

```
[root@localhost ipv4]# echo "bic" > tcp_congestion_control
[root@localhost ipv4]#
```

The problem of this way is that, the sitting will be reset by the default values after system rebooting operation, because the system will update all of these parameters values during the startup of the system. However to avoid this problem from happening, sysctl.conf file should be used to fix the parameters values, thus these parameters values will not change even after system rebooting unless they are changed in this file. This file contents will

be uploaded each time the system starts up. To play with this file, just go to this folder /etc/ and write the following command:

```
[root@localhost etc]# gedit sysctl.conf
[root@localhost etc]#
```

This command will start Linux "gedit" text editor tool to show the contents of this file, this tool (gedit) is a traditional text editor comes with the standard package of Linux. Nevertheless, if it has not been selected to be installed during the installation of the operating system, an error message will appear to tell that, the application (gedit) is not found. Consequently, the user must try to use any other text editor (such as KWrite or DocWriter). Figure 4.3 shows the contents of sysctl.config file, and it can be carefully modified as shown in figure 4.4.

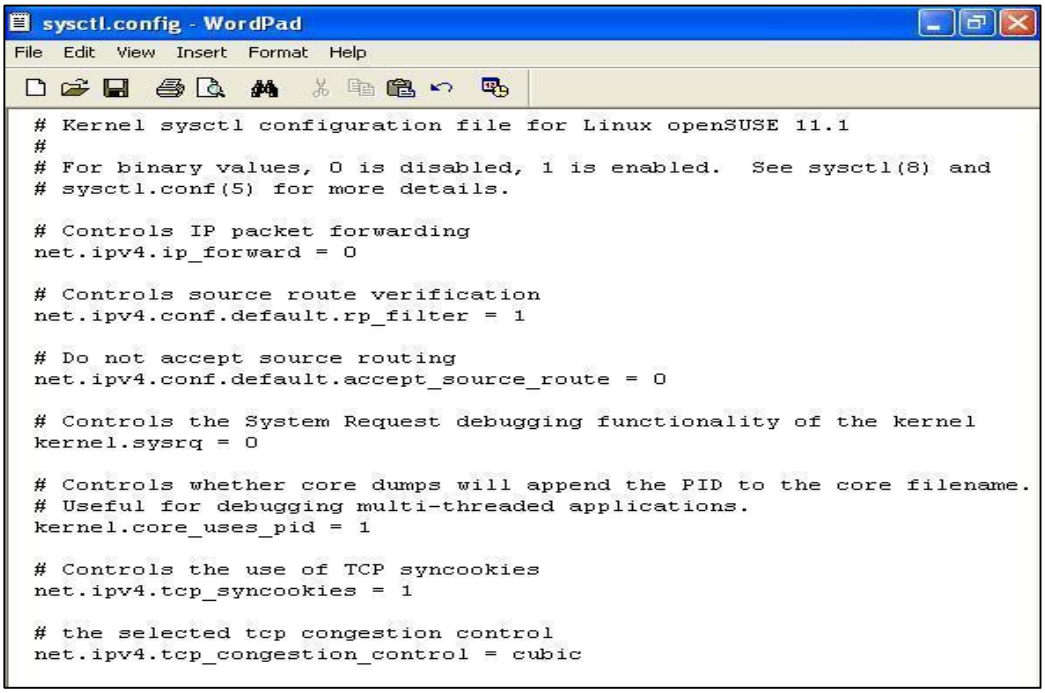

Figure 4.3: sysctl.config before modification

After modifying the file, press “save file” and close “gedit” window then write the following command to load the new configurations from sysctl.conf file without rebooting the system.

```
[root@localhost etc]# sysctl -p
[root@localhost etc]#
```

Directly, after executing the above command, the new configurations will be effective. This procedure was needed to switch between the implementations of TCP variants in the Linux kernel after each experiment completion stage.

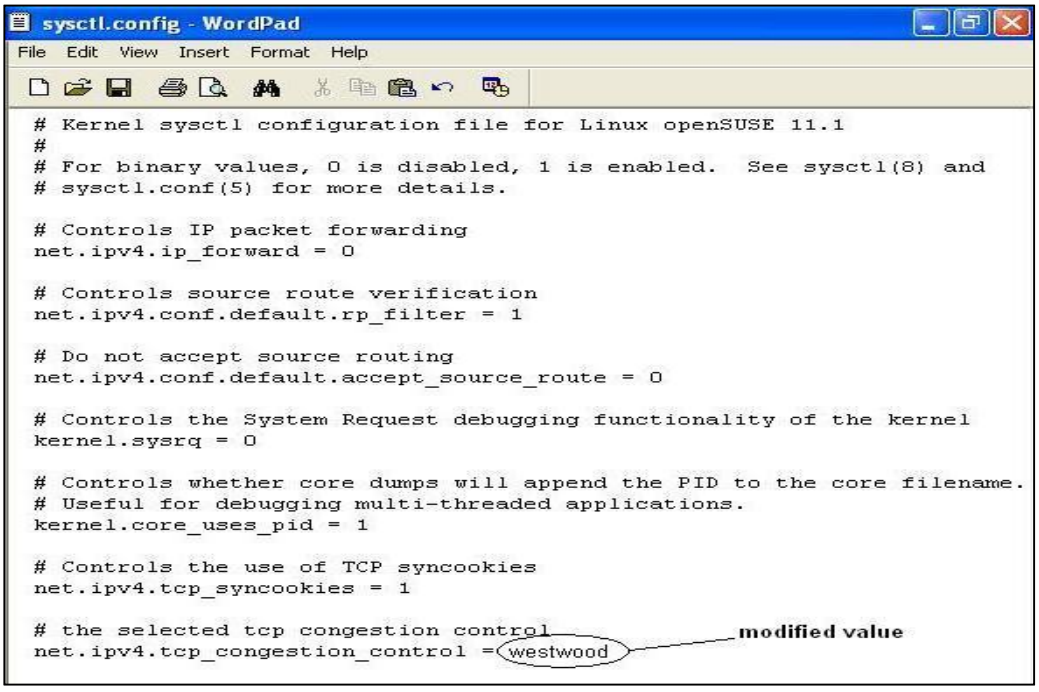

Figure 4.4: sysctl.config after modification

4.3 Collecting Data Using TCPdump

TCPdump is freeware software and it is a common packet sniffer that runs under the command line. It allows the user to intercept and display TCP/IP and other packets being transmitted or received over a network to which the computers are attached. It was originally written by Van Jacobson, Craig Leres and Steven McCanne who were, at the time, working in the Lawrence Berkeley Laboratory Network Research Group. TCPdump works on most Unix-like operating systems: Linux, Solaris, BSD, Mac and AIX among others. In those systems, TCPdump uses the libpcap library to capture packets. Furthermore, there is a port of TCPdump for Microsoft Windows called WinDump; this uses WinPcap, which is a port of libpcap to Windows [28].

In some Unix-like operating systems, a user must have superuser privileges to use TCPdump because the packet capturing mechanisms on those systems require elevated privileges. In other Unix-like operating systems, the packet capturing mechanism can be configured to allow non-privileged users to use it; if that is done, superuser privileges are not required. The user may optionally apply a BPF-based filter to limit the

number of packets seen by TCPdump; this renders the output more usable on networks with a high volume of traffic.

However, TCPdump version 4.0 has been used to monitor the network traffic and thus to collect the trace files from both sides (sender and receiver). Figure 4.5 shows one sample of trace files as it outputs from TCPdump.

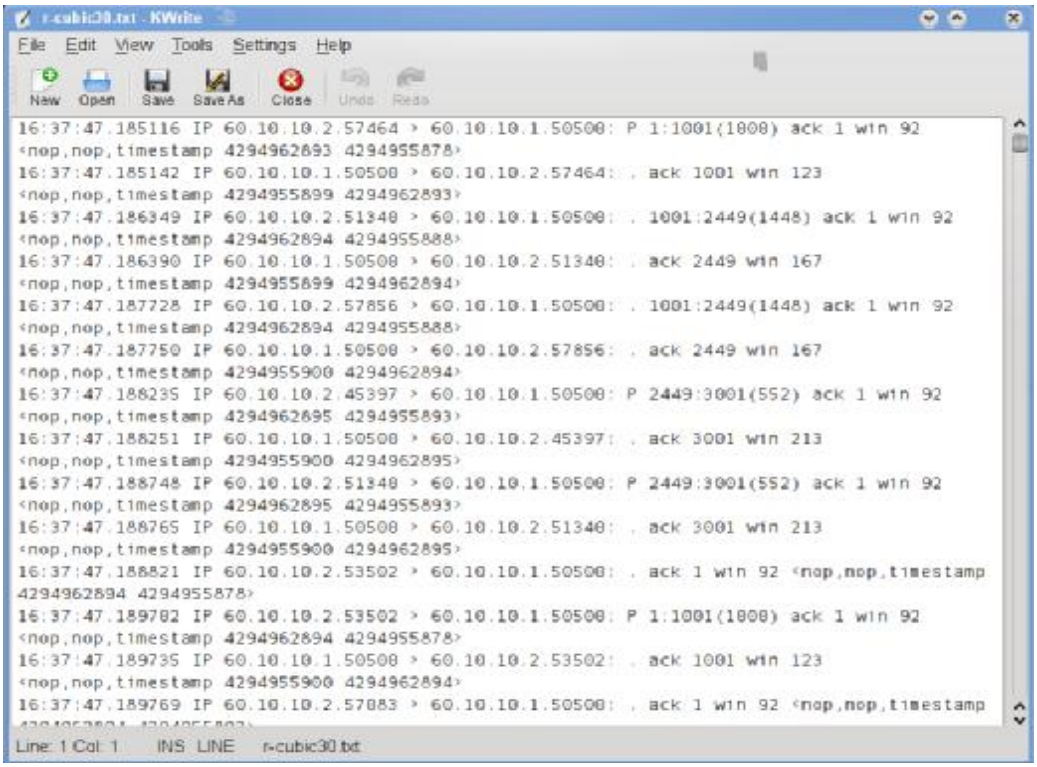

Figure 4.5: TCPdump output sample

4.4 Analyzing Data Using AWK

AWK is a programming language that is designed for processing text based data, either in files or data streams, and was created at Bell Labs in the 1970s. The name AWK is derived from the family names of its authors — Alfred Aho, Peter Weinberger, and Brian Kernighan. AWK, when written in all lowercase letters, refers to the UNIX or Plan 9 program that runs other programs written in the AWK programming language.

It is a language for processing text files. A file is treated as a sequence of records, and by default each line is a record. Each record is broken up into a sequence of fields, so it can thought that, the first word in a line is the first field, and the second word is the second field, and so on. An AWK program is a sequence of pattern-action statements. AWK reads the input a line at a time. A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed.

AWK is one of the early tools to appear in Unix Version 7 and gained popularity as a way to add computational features to a UNIX pipeline. A version of AWK language is a standard feature of nearly every modern Unix-

like operating system available today. AWK mentioned in the Single UNIX Specification as one of the mandatory utilities of a UNIX operating system. Besides the Bourne shell, AWK is the only other scripting language available in a standard UNIX environment. Implementations of AWK exist as installed software for almost all other operating systems [29].

However, AWK v3.1.6 Linux tool has been used to analyze data (trace files) that have been collected by TCPdump as motioned in previous section. AWK code which written below has been used only to analyze the trace files as shown above in figure 4.5 and it produces a new trace file as shown in figure 4.6, the new file is ready to be drawn as graphs after applying another AWK codes on it to calculate the throughput, loss ratio and fairness.

```
BEGIN{ sth = 0; stm = 0; sts = 0; stf = 0; flag = 0 }
{
if (flag == 0){
    sth = $1    ;stm = 2$2     sts = 3$3     stf = 4$4     flag = 11
}
if (($6 == "192.168.0.2") && ($11 != "F") && ($11 != "P") &&
      ($11 != "FP") && ($11 != "ack") && ($11 != "S") &&
      ($12 != "PTR?″)){
    x1 = $1 - sth    ;x2 = $2 - stm ;x3 = $3 - sts;x4 = $4 - stf;
    newts = (x1 * 60 * 60 * 1000000) + (x2 * 60 * 1000000) +
            (x3 * 1000000) + (x4);
    print newts/100000 " " $6 " " $7 " " $11 " " $13
}
if (($6 == "192.168.0.2") && ($11 == "P")){
    x1 = $1 - sth ;x2 = $2 - stm ;x3 = $3 - sts    ;x4 = $4 - stf;
    newts = (x1 * 60 * 60 * 1000000) + (x2 * 60 * 1000000) +
            (x3 * 1000000) + (x4);
    print newts/100000 " " $6 " " $7 " " $12 " " $14
}
if (($6 == "192.168.0.2") && ($11 == "FP")){
    x1 = $1 - sth ;x2 = $2 - stm ;x3 = $3 - sts ;x4 = $4 - stf;
    newts = (x1 * 60 * 60 * 1000000) + (x2 * 60 * 1000000) +
            (x3 * 1000000) + (x4);
    print newts/100000 " " $6 " " $7 " " $12 " " $14
```

```
}
if (($6 == "192.168.0.2") && ($11 == "F")){
    x1 = $1 - sth ;x2 = $2 - stm ;x3 = $3 - sts    ;x4 = $4 - stf;
    newts = (x1 * 60 * 60 * 1000000) + (x2 * 60 * 1000000) +
            (x3 * 1000000) + (x4);
    print newts/100000 " " $6 " " $7 " " $12 " " $14
}
}
END{}
```

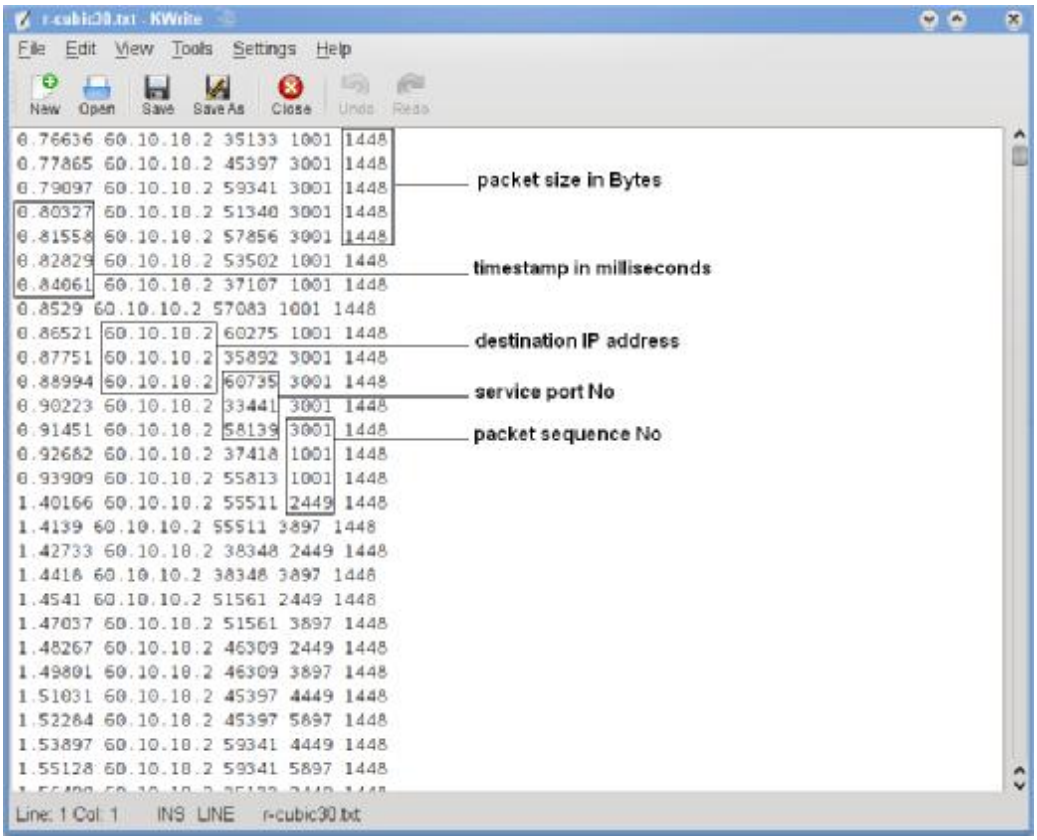

Figure 4.6: Trace file after AWK processing

In figure 4.6, the throughput and throughput ratio can be calculated by counting the “packet size” field while the time stamp being considered. Likewise, Fairness also can be calculated by counting the total throughput for each flow that can be differentiated by the field of “port No”; in the trace file

shown in figure 4.5, there are 30 TCP flows each one has a different port No than the others while the destination IP address is the same. All of these files have been collected from the receiver side.

On the other hand, the files that collected from the sender side have been used with the aforementioned trace files (from receiver side) to calculate the loss ratio (unnecessary retransmission).

4.5 Experiment Scenario

The whole of the experiment has been iterated four times to get accurate results. Each TCP variant (Reno, Scalable, HTCP, HSTCP, CUBIC) has its own experiment, each experiment contains seven stages for each TCP variant that means thirty five times for all TCP variants. For each TCP variant it started by one connection (single-based) then 5, 10, 15, 20, 25 and 30 connections (Parallel-based). After thecompletion of each TCP variant experiment, Linux kernel parameters have to be changed to switch to the next TCP variant. In this experiment, the network traffic has been monitored by using TCPdump on the Sender1 and Receiver1 in presence of background traffic from Sender2 to Receiver2.

4.6 Summary

This chapter shows how the algorithm was implemented using Microsoft C# and DotNet framework and how it is was running in Linux environment using MONO framework. Moreover, it shows to the readers how Linux kernel can be modified and tuned to fulfill the requirements of the experiment as well as it shows how the data has been collected by TCPdump and how it analyzed by using AWK programming language and shell commands.

CHAPTER 5

TEST-BED RESULTS AND DISCUSSION

This chapter gives a brief information and explanation on the test-bed experiment results. This results start with throughput ratio then loss ratio and finally TCP fairness graphs.

5.1 The Results

The results of this experiment as shown in figure 5.1 reveal that, the performance throughput ratio of the involved TCP variants are almost the same in all cases, which means that all of these TCP variants can achieve similar performance throughput and can provide the same link utilization. Moreover, all of these TCP variants achieved high performance in parallel modes than single mode. As shown in figure 5.1, the bandwidth of the bottleneck link has not been fully utilized in both cases of single connection and parallel of five TCP connections, while it is fully utilized in the rest cases (parallel of 10, 15, 20, 25, 30 TCP connections).

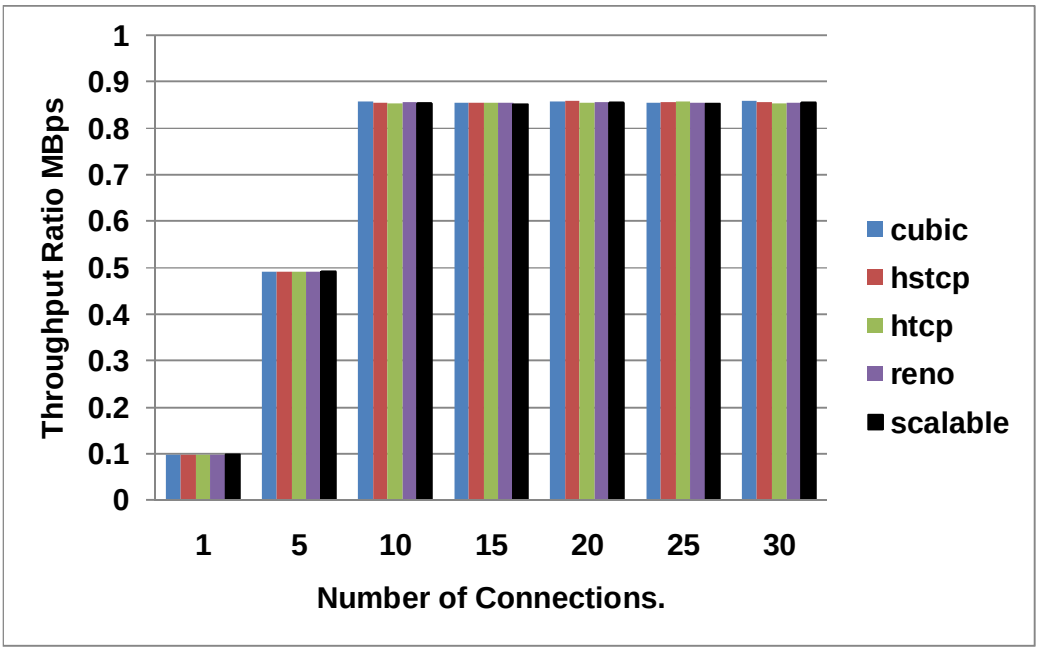

Figure 5.1: Throughput Ratio vs. Number of Connections

On the other hand, the main difference between TCP variants as shown figure 5.2 is loss ratio; the graph reveals that, the use of Scalable, HTCP and HSTCP cause a lot of unnecessary retransmission while CUBIC and Reno do not. So, means that CUBIC and Reno can achieve better performance than other TCP variants. Figure 5.3 shows the average of loss ratio among TCP variants, this graph gives a brief comparison to facilitate the operation of TCP variants evaluation, and it is so clear that, CUBIC and Reno was the best while TCP Scalable was the worst.

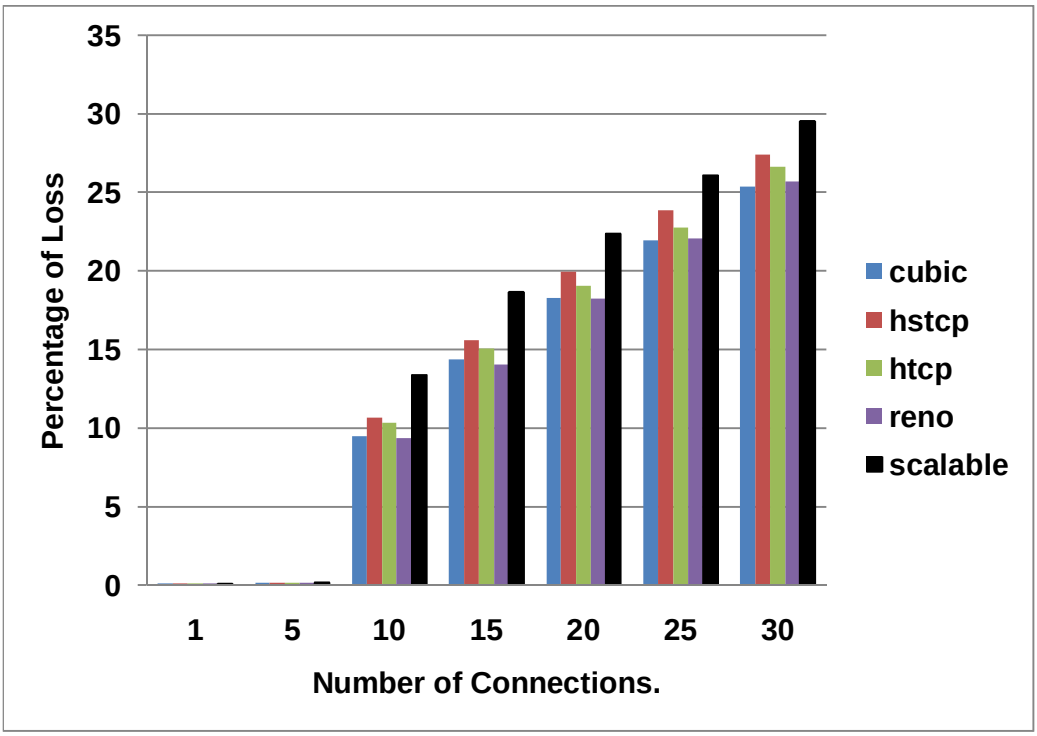

Figure 5.2: Loss Ratio vs. Number of Connections

As shown in figure 5.2 above, when the bottleneck is not fully utilized (no congestion) in both cases of single TCP connection and parallel of five TCP connections the loss ratio is very small. But, after the number of parallel TCP connections increased the loss ratio increases as well. In figure 5.1, the bottleneck link of the implemented topology has been fully utilized when the number of parallel TCP connections equals 10 TCP connections. Which means that, the increasing of the number of parallel TCP connections more than 10 connections for this link capacity will not be useful and it will cause TCP overhead which is the increasing of unnecessary retransmission while the throughput cannot exceed the bandwidth limit of the bottleneck link.

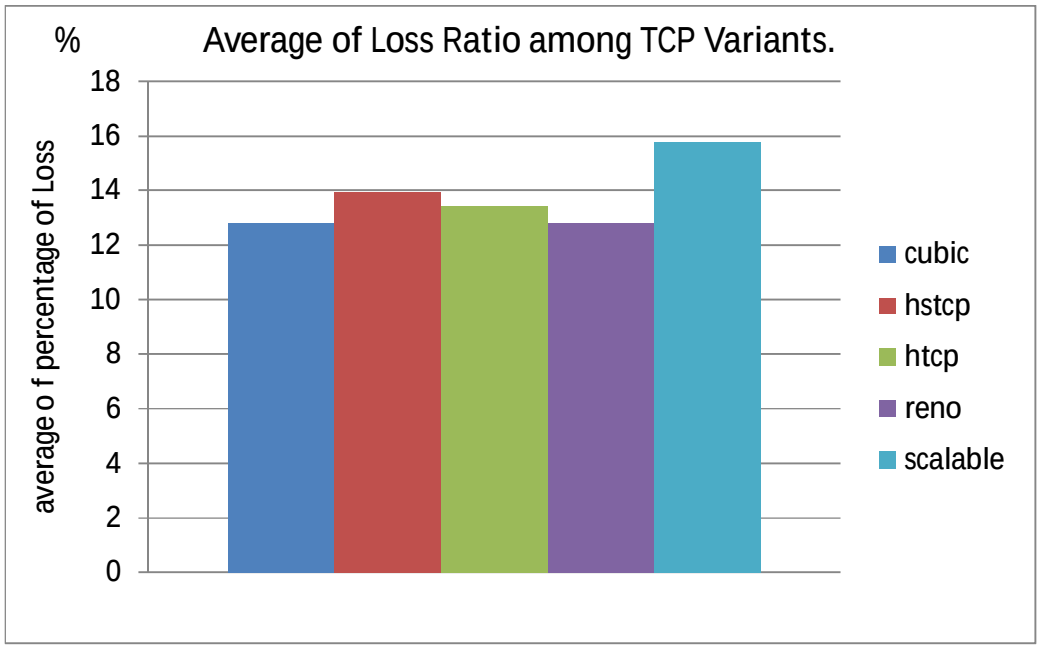

Figure 5.3: The average of loss ratio among TCP variants

Figure 5.3 shows that, the averages of loss ratio that recorded by these TCP variants were clearly different, for instance, Scalable TCP was the worst one and it has recorded around 16% of data loss from the entire throughput, contrarily CUBIC and Reno was the best ones and they have recorded around 13% of data loss from the entire throughput. Moreover, HTCP and HSTCP are partially reasonable and better than Scalable TCP.

Relatively, the range of fairness variation starts from 100% to 90%. As shown in figure 5.4, and it is very clear that, Reno scores the worst fairness index compared with the others while CUBIC and Scalable were the best, but Scalable was highly affected by increasing the number of connections

otherwise CUBIC was reasonably affected. When the bottleneck is not congested all of the TCP variants were almost the same and they achieves similar Fairness Index which is about 100% but after increasing the number of TCP connections to be more than 10 connections which makes the bottleneck highly congested the fairness index is slightly decreased.

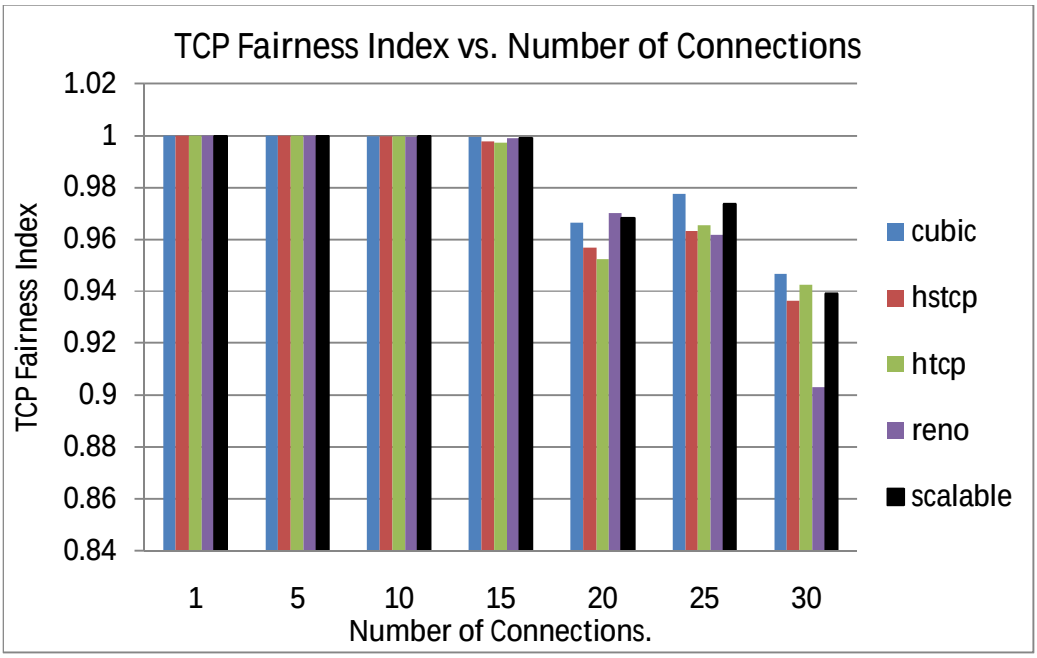

Figure 5.4: TCP Fairness Index vs. Number of Connections

Figure 5.5 shows in brief that, the order of TCP variants in term of TCP Fairness are CUBIC, Scalable, HTCP, HSTCP and Reno respectively, the first order the highest Fairness and the last order the worst Fairness.

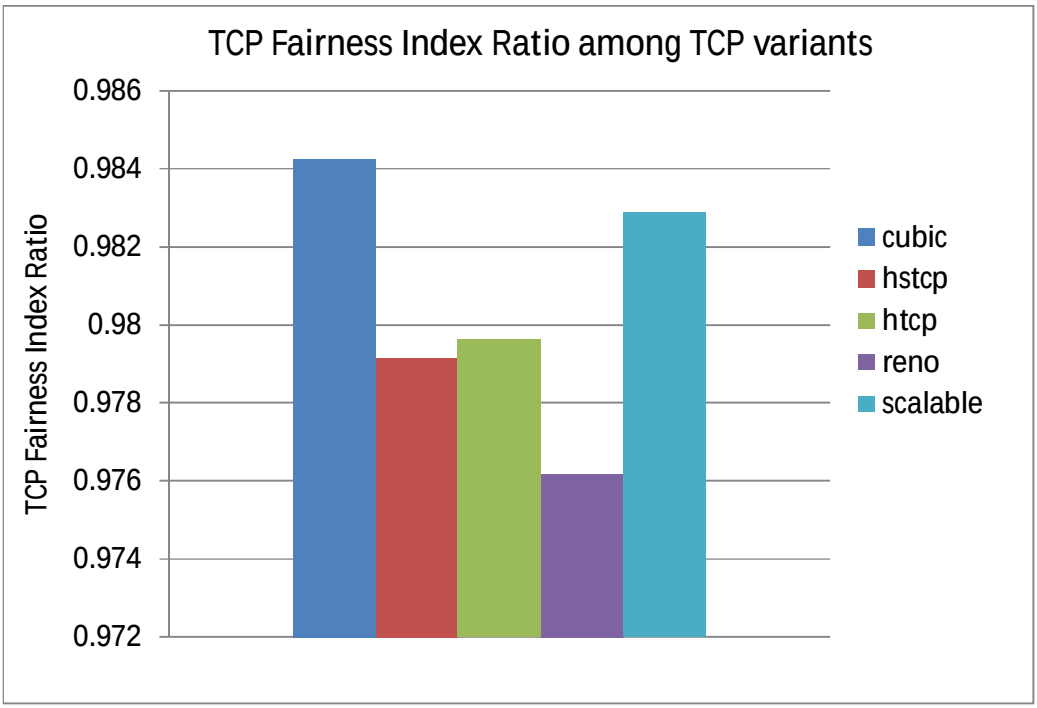

Figure 5.5: TCP Fairness Index Ratio among TCP variants

Clearly, for this topology with 10 Mbps bottleneck, the appropriate number of parallel TCP connection to fully utilize the available bandwidth is 10 concurrent TCP connections that will cause the least possible amount of unnecessary retransmissions with high fairness index. This number of parallel TCP connections considered as the threshold of TCP parallelism of this bottleneck bandwidth, and each link capacity has its parallelism threshold. This threshold should be carefully calculated before starting TCP parallelism based on some variables which not in the scope of this work.

5.2 Summary

From the discussion in previous section, it is very clear that:

- The increasing of parallelized TCP connections increases the throughput of these parallel TCP connections.
- The number of parallelized TCP connections should be carefully chosen based on some variables such as the available bandwidth otherwise it will cause TCP overhead (unnecessary retransmissions).
- The study reveals that, CUBIC is the best one of TCP variants, and it is better than the other TCP variants which are involved in this experiment in terms of its throughput, loss ratio and fairness.
- The proposed parallel TCP algorithm achieves high throughput while it maintains the fairness among the competed connections.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This thesis presents the design of the experiment hardware and software. Test bed experiment has been presented as well to show the impact of the proposed parallel TCP algorithm on TCP variants in terms of throughput, loss ratio and TCP fairness. The major conclusions and future work of this experiment will be presented in the following sections.

6.1 Conclusion

From the results and discussion in the previous chapter, it has been concluded that:

- Single-based TCP cannot overcome parallel TCP especially in heterogeneous networks.
- As shown in the results all TCP variants can achieve a good performance throughput but when the bottleneck is fully utilized which means that, there is a congestion the clear difference will not be in the throughput but it will be in Loss Ratio (unnecessary retransmission) and TCP Fairness.

- CUBIC TCP achieves higher performance than the other TCP variants in terms of throughput, loss-ratio and fairness, that is why it is chosen as default Linux TCP congestion control in the latest versions such as Fedora 10, 11 and openSUSE 11, 11.1 ,etc.
- The proposed parallel TCP algorithm achieves high throughput and it can effectively utilize the high-speed network links while it maintains the fairness among the competed connections.

6.2 Future Work

In this experiment, some of TCP features like SACK and FACK have been disabled to show the impact of changing the congestion control algorithms on the TCP performance, but there is a strong intention to repeat this experiment with different settings. For instance, SACK and FACK will be enabled to emphasize their impact and to show either they are worth to be used with the proposed parallel TCP algorithm or not. On the other hand, there are some modification have to be done later in Linux Kernel to tune and optimize TCP CUBIC congestion control algorithm.

REFERENCES

[1] Hacker, Thomas J. (2004). Improving end-to-end reliable transport using parallel transmission control protocol sessions. Ph.D. thesis, University of Michigan, United States.

[2] Comer, Douglas E. (2006). Internetworking with TCP/IP: Principles, Protocols, and Architecture. 5th edition. Prentice Hall.

[3] M. Bateman, S. Bhatti, G. Bigwood, D. Rehunathan, C. Allison, T. Henderson, and D. Miras (2008). A Comparison of TCP Behaviour at High Speeds Using NS-2 and Linux. CNS 08 Proceedings of the 11th communications and networking simulation symposium, USA, ACM, 2008, pp: 30-37.

[4] Jacobson, Van (1995). Congestion Avoidance and Control. ACM SIGCOMM Computer Communication Review, 1995, 25(1), pp: 157–187.

[5] J. Lekashman, Type of Service Wide Area Networking. Conference on High Performance Networking and Computing, 1989, pp: 732 - 736.

[6] D. Iannucci, J. Lekashman (1992). MFTP: Virtual TCP Window Scaling Using Multiple Connections. RND-92-002, NASA Ames Research Centre, 1992, pp: 1-16.

[7] M. Allman, H. Kruse, S. Ostermann (1996). An application-level solution to TCP's satellite inefficiencies. WOSBIS, November 1996, pp: 1-8.

[8] B. Allcock, J. Bester, J. Bresnahan, A.L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, S. Tuecke (2002). Data management and transfer in high-performance computational grid environments. J. Parallel Computing, 28(5), pp: 749 - 771.

[9] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster (2005), The globus striped GridFTP framework and server. Proceedings of Super Computing, 2005, pp: 54-65.

[10] H. Sivakumar, S. Bailey, R. Grossman (2000). PSockets: the case for application-level network striping for data intensive applications using high speed wide area networks. Conference on High Performance Networking and Computing, 2000, Article No. 37.

[11] R. Grossman, Y. Gu, D. Hamelberg, D. Hanley, X. Hong, J. Levera, M. Mazzucco, D. Lillethun, J. Mambrett, J. Weinberger (2002). Experimental studies using photonic data services at IGrid. J. Future Computer. Systems, 2003, 19(6), pp: 945-955.

[12] H. Balakrishnan, H. Rahul, S. Seshan. An integrated congestion management architecture for internet hosts. SIGCOMM, 1999, 29(4), pp: 175-187.

[13] L. Eggert, J. Heidemann, J. Touch (2000). Effects of ensemble–TCP, ACM SIGCOMM, 2000, 30(1), pp: 15-29.

[14] Hacker, T.J., Noble, B.D. and Athey, B.D (2004). Improving throughput and maintaining fairness using parallel TCP. INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, 2004, Vol 4, pp: 2480 – 2489.

[15] T. Hacker, B. Noble, B. Athey (2002). The effects of systemic packet loss on aggregate TCP flows, ACM/IEEE conference on Supercomputing, 2002, pp: 1-15.

[16] J. Crowcroft, P. Oechslin (1998). Differentiated end-to-end Internet services using a weighted proportionally fair sharing TCP. ACM SIGCOMM, 1998, 28(3), pp: 53-69.

[17] Hung-Yun Hsieh; Sivakumar, R (2002). pTCP: an end-to-end transport layer protocol for striped connections. IEEE/ICNP, 2002, pp: 24-33.

[18] T. Kelly (2003). Scalable TCP: improving performance in highspeed wide area networks. ACM SIGCOMM, 2003, 32(2), pp: 83-91.

[19] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, December 2003.

[20] R. Shorten, D. Leith (2004). H-TCP: TCP for high-speed and long-distance networks, PFLDnet 2004, Argonne, USA.

[21] L. Xu, K. Harfoush, I. Rhee (2004). Binary increase congestion control for fast long-distance networks. INFOCOM, 2004, 4, pp: 2514-2524.

[22] Sangtae Ha, I. Rhee, L. Xu (2005). CUBIC: a new TCP-friendly high-speed TCP variant, ACM SIGOPS, 2005, 42(5), pp: 64-74.

[23] D. Wei, C. Jin, S. Low, S. Hegde (2006). FAST TCP: motivation, architecture, algorithms, performance. IEEE/ACM TON, 2006, 14(6), pp: 1246-1259.

[24] Qiang Fu, Jadwiga Indulska, Sylvie Perreau and Liren Zhang (2007). Exploring TCP Parallelization for performance improvement in heterogeneous networks. Computer Communications, 2007, 30(17), pp: 3321-3334.

[25] Floyd, S. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), 2003.

[26] D. Chiu and R. Jain. Analysis of the increase/decrease algorithms for congestion avoidance in computer networks. Computer Networks and ISDN, 1989, 17(1), pp: 1-24.

[27] Jain, R., Chiu, D.M., and Hawe, W (1984). A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Systems. DEC Research Report TR-301, 1984.

[28] Retrieved 13/5/2009 from http://en.wikipedia.org/wiki/Tcpdump

[29] Retrieved 13/5/2009 from http://en.wikipedia.org/wiki/AWK

BIBLIOGRAPHY

Mohamed A. Alrshah received his B.Sc degree in Computer Science from Naser University - Libya, 2000. He worked as a technician of computer laboratories at Naser University - Libya. From 2004 to June 2006, he has joined Generic Electric Company of Libya (GECOL) as system developer, and from June 2006 to Dec 2007; he has joined Asmarya University of Islamic Science – Libya, besides, his private work in field of networking such as VIA-Satellite communication networks which provide Internet services and Communication networks, in addition to Wire and Wireless Networks from 2001 until now. Below some of the companies that he provide network solutions for:

BESIX Group: is the largest Belgian construction group, a conglomerate of companies active in the construction, engineering, environmental, real estate and concession sectors. The Group was founded in 1909 and since then has known impressive and regular growth. . One VIA-satellite Communication systems has been installed in Zuara site in Libya. http://www.besix.com/

CEMAG Anlagenbau Dessau GmbH: develops solutions for the cement industry, the pit and quarry industry, and the disposal industry. Their process engineering and design advances are based on many decades of experience, which they pass on to their customers for their benefit. One VIA-satellite Communication systems has been installed in Zliten site in Libya. http://www.cemag.de/

SIDEM: executes large desalination projects on a turnkey basis, and in association with sister companies of the Veolia group is the sole provider of hybrid diesel solutions involving both thermal distillation and reverse osmosis. Two VIA-satellite Communication systems have been installed in Tripoli and Zuara sites in Libya. http://www.sidem-desalination.com/en/

Ahlia Cement Company (ACC): is the largest cement company in Libya, it was founded in 1965. Six VIA-satellite Communication systems have been installed in different cities, and wireless local area network has been installed in Zliten Cement Plant to cover its area. http://www.acc.com.ly/

Arab Union for Cement and Building Materials (AUCBM): is an inter-Arab International organization, affiliated to the General Secretariat of the Arab League and the Council of Arab Economic Unity. It was founded in 1977. Two VIA-satellite Communication systems have been installed in Alborj Cement Plant, Zliten, Libya. http://www.aucbm.org/english/main.htm.

SSB Enterprises Private Limited: is Indian company provides services of cement testing. Two VIA-satellite Communication systems have been installed in Alborj Cement Plant, Zliten, Libya.

Furthermore, providing Wireless Internet Service Providers solutions in two cities (Zliten and Alkhoms). These solutions involve hardware and Software installations and WISP management.

APPENDIX

Appendix A: TC Source Code

- Server Class:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net;
using System.Threading;
using System.Net.Sockets;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace MyServer
{
    public struct Pair
    {
        public Pair(TcpClient _client, int _clientID)
        {
            this.Client = _client;
            this.ClientID = _clientID;
        }
        public TcpClient Client;
        public int ClientID;
    }

    public partial class ServerForm : Form
    {
        private static volatile SynchronizationContext context;
        private static volatile List<Thread> AllThreads =
                         new List<Thread>();
        private static volatile List<int> Connected =
                         new List<int>();
        private static volatile List<int> Disconnected =
                         new List<int>();
        private static volatile List<TraceRecord> ServerPerformance=
                         new List<TraceRecord>();
        private static volatile Thread listenerThread;
        private static volatile Thread progressingThread;
        private static volatile TcpListener listener;
        private static volatile bool StopFlag = false;
        private static volatile IPAddress serverIP;
        private static volatile int serverPort;
        private static volatile int StartTime;
        private static volatile int interval_time;
        private static volatile int Indexer;
        private static volatile int bufferSize;
            private static volatile object MyLock = new object();

        public ServerForm()
        {
            InitializeComponent();
            context = SynchronizationContext.Current;
            if (context == null) context =
```

```
                    new SynchronizationContext();
}

private void startButton_Click(object sender, EventArgs e)
{
    StopFlag = false;
    Indexer = 0;
    bufferSize = Convert.ToInt32(bufferSizeBox.Value);
    serverIP = IPAddress.Parse(textBox1.Text);
    serverPort = int.Parse(textBox2.Text);
    interval_time = Int16.Parse(intervalBox.Text);
    listenerThread = new Thread(new
                ThreadStart(StartListener));
    listenerThread.IsBackground = true;
    listenerThread.Start();
    progressingThread = new Thread(new
                ThreadStart(Progressing));
    progressingThread.IsBackground = true;
    progressingThread.Start();
}

private void stopButton_Click(object sender, EventArgs e)
{
    StopFlag = true;
    Indexer = 0;
    if (listener != null) listener.Stop();
    if (AllThreads.Count > 0)
        foreach (Thread thr in AllThreads)
            if (thr != null) thr.Abort();
    AllThreads.Clear();
    ResultBox.AppendText("All Client connections are
                                    Disconnected.\r\n");
    status.Text = "Active Connections : (" +
                AllThreads.Count.ToString() + ")...";
    statusBar1.Text = "The server status is: Stopped....";
}

private void ServerForm_Load(object sender, EventArgs e)
{
    textBox1.Text = "60.10.10.1";
    textBox2.Text = "50500";
}

private void StartListener()
{
    try
    {
        listener = new TcpListener(serverIP, serverPort);
        listener.Start();
        context.Send(new SendOrPostCallback((s) =>
                statusBar1.Text = "The server status is:
                Waiting For New Connection From any
                Client...."), null);
        TcpClient newClient;
        Thread clientThread;
        int my_bufferSize = bufferSize;
        bool IsFirstTime = true;
        while (true)
        {
            newClient = listener.AcceptTcpClient();
            Indexer++;
```

```
                newClient.ReceiveBufferSize = my_bufferSize;
                clientThread = new Thread(new
                    ParameterizedThreadStart(AcceptMessages));
                clientThread.IsBackground = true;
                clientThread.Start(new
                                Pair(newClient, Indexer));
                AllThreads.Add(clientThread);
                if (IsFirstTime)
                {
                                lock(MyLock)
                                {
                                      StartTime =
                    LongTime.GetLongTime().ToMillisecondes();
                                      IsFirstTime = false;
                                }
                }
                Connected.Add(Indexer);
            }
        }
        catch (Exception ex)
        {
            if (!StopFlag) MessageBox.Show("Please check your
                           network setting.\r\n" + ex.Message);
        }
    }

    private void Progressing()
    {
        while (!StopFlag)
        {
            while (Connected.Count > 0)
            {
                context.Send(new SendOrPostCallback((s) =>
                    ResultBox.AppendText("Client " +
                    Connected[0].ToString() + "  has been
                    Connected.\r\n")), null);
                Connected.RemoveAt(0);
            }
            while (Disconnected.Count > 0)
            {
                context.Send(new SendOrPostCallback((s) =>
                    ResultBox.AppendText("Client " +
                    Disconnected[0].ToString() + "  has been
                    disconnected.\r\n")), null);
                Disconnected.RemoveAt(0);
            }
            context.Send(new SendOrPostCallback((s) =>
                    status.Text = "Active Connections : (" +
                    Indexer.ToString() + ")..."), null);
            Thread.Sleep(100);
        }
    }

    private void AcceptMessages(object inputPair)
    {
        Pair pair = (Pair)inputPair;
        TcpClient client = pair.Client;
        int indx = pair.ClientID;
        int bytesRead;
        string receivedMessage;
        int rbSize = client.ReceiveBufferSize;
```

```
        byte[] data = new byte[rbSize];
        int my_interval_time = interval_time;
        int my_startTime = StartTime;
        try
        {
            NetworkStream ns = client.GetStream();
            int currentTime;
            while (!StopFlag)
            {
                bytesRead = ns.Read(data, 0, rbSize);
                receivedMessage = Encoding.ASCII.GetString(
                                        data, 0, bytesRead);
                currentTime =
                    LongTime.GetLongTime().ToMillisecondes() -
                    my_startTime;
                if (receivedMessage.Length != 0)

                    ServerPerformance.Add(TraceRecord.Create(ind
                    x, currentTime));
                else
                    throw new ManualStopExcption();
                Thread.Sleep(my_interval_time);
            }
        }
        catch{}
        finally
        {
            client.Close();
            Indexer--;
            try
            {
                AllThreads.Remove(Thread.CurrentThread);
            }
            catch { }
            Disconnected.Add(indx);
        }
    }

    private void Reset_Click(object sender, EventArgs e)
    {
        stopButton_Click(sender, e);
        ServerPerformance.Clear();
        ResultBox.Clear();
    }

    private void WriteToTextFile(object o)
    {
        List<TraceRecord> inputList = (List<TraceRecord>)o;
        MemoryStream ms = new MemoryStream();
        StreamWriter writer = new
                            StreamWriter(ms, Encoding.UTF8);
        TraceRecord smallRecord;
        int missedRecords = 0;

        for (int i = 1; i < inputList.Count; i++)
        {
            if (inputList[i] == null)
            {
                inputList.RemoveAt(i);
                i--;
                missedRecords++;
```

```
        }
        else
        {
            if (inputList[i - 1] == null)
            {
                inputList.RemoveAt(i - 1);
                i--;
                missedRecords++;
            }
            else
            {
                if (inputList[i].Duration_MSs <
                            inputList[i - 1].Duration_MSs)
                {
                    if (i == 1)
                    {
                        smallRecord = inputList[i];
                        inputList.RemoveAt(i);
                        inputList.Insert(0, smallRecord);
                    }
                    else
                    {
                        for (int j = i - 1; j == 0; j--)
                        {
                            if (inputList[i].Duration_MSs >
                                  inputList[j].Duration_MSs)
                            {
                                smallRecord = inputList[i];
                                inputList.RemoveAt(i);
                                inputList.Insert(j + 1,
                                              smallRecord);
                                break;
                            }
                        }
                    }
                }
            }
        }
    }

    MessageBox.Show("The damaged records count is " +
                            missedRecords.ToString());
    foreach (var traceRecord in inputList)
        writer.WriteLine(traceRecord.ToString());
    byte[] output = ms.ToArray();
    FileStream fileStream = new
                FileStream(Application.StartupPath +
                "\\date" +

                DateTime.Now.Day.ToString() + "-" +

                DateTime.Now.Hour.ToString() + "-" +

                DateTime.Now.Minute.ToString() + "-" +

                DateTime.Now.Second.ToString() + "-" +

                DateTime.Now.Millisecond.ToString() +
                ".txt", FileMode.OpenOrCreate,
                FileAccess.Write);
    try
```

```
            {
                fileStream.Write(output, 0, output.Length);
            }
            catch (Exception ex)
            {
                MessageBox.Show("The Trace File Can not be saved on
                                the hard disk !!!\r\n" + ex.Message);
            }
            finally
            {
                fileStream.Close();
                writer.Close();
                ms.Close();
            }
        }

        private void Save_Click(object sender, EventArgs e)
        {
            Thread SaveThread = new Thread(new
                  ParameterizedThreadStart(WriteToTextFile));
            SaveThread.IsBackground = true;
            SaveThread.Start(ServerPerformance);
        }

        private void ServerForm_FormClosing(object sender,
                                            FormClosingEventArgs e)
        {
            Reset_Click(sender, e);
        }
    }
}
```

TcpListener Class:

```
using System;
using System.Net;

namespace System.Net.Sockets
{
    // Summary:
    //     Listens for connections from TCP network clients.
    public class TcpListener
    {
        // Summary:
        //     Initializes a new instance of the
            //System.Net.Sockets.TcpListener class that
        //     listens on the specified port.
        //
        // Parameters:
        //   port:
        // The port on which to listen for incoming connection
            //attempts.
        //
        // Exceptions:
        //   System.ArgumentOutOfRangeException:
        //     port is not between System.Net.IPEndPoint.MinPort and
            //System.Net.IPEndPoint.MaxPort.
        [Obsolete("This method has been deprecated. Please use
```

```
            TcpListener(IPAddress localaddr, int port)
            instead.
            http://go.microsoft.com/fwlink/?linkid=14202")]
public TcpListener(int port);
//
// Summary:
//     Initializes a new instance of the
    //System.Net.Sockets.TcpListener class with
//     the specified local endpoint.
//
// Parameters:
//   localEP:
//     An System.Net.IPEndPoint that represents the local
    //endpoint to which to bind
//     the listener System.Net.Sockets.Socket.
//
// Exceptions:
//   System.ArgumentNullException:
//     localEP is null.
public TcpListener(IPEndPoint localEP);
//
// Summary:
//     Initializes a new instance of the
    //System.Net.Sockets.TcpListener class that
//     listens for incoming connection attempts on the
    //specified local IP address
//     and port number.
//
// Parameters:
//   localaddr:
//     An System.Net.IPAddress that represents the local IP
    //address.
//
//   port:
//     The port on which to listen for incoming connection
    //attempts.
//
// Exceptions:
//   System.ArgumentNullException:
//     localaddr is null.
//
//   System.ArgumentOutOfRangeException:
//     port is not between System.Net.IPEndPoint.MinPort and
    //System.Net.IPEndPoint.MaxPort.
public TcpListener(IPAddress localaddr, int port);

// Summary:
//     Gets a value that indicates whether
    //System.Net.Sockets.TcpListener is actively
//     listening for client connections.
//
// Returns:
//     true if System.Net.Sockets.TcpListener is actively
    //listening; otherwise,
//     false.
protected bool Active { get; }
//
// Summary:
//     Gets or sets a System.Boolean value that specifies
    //whether the System.Net.Sockets.TcpListener
//     allows only one underlying socket to listen to a
```

```
    //specific port.
//
// Returns:
//     true if the System.Net.Sockets.TcpListener allows
    //only one System.Net.Sockets.TcpListener
//     to listen to a specific port; otherwise, false. . The
    //default is true for
//     Windows Server 2003 and Windows XP Service Pack 2 and
    //later, and false for
//     all other versions.
//
// Exceptions:
//   System.InvalidOperationException:
//     The System.Net.Sockets.TcpListener has been started.
    //Call the System.Net.Sockets.TcpListener.Stop()
//     method and then set the
//System.Net.Sockets.Socket.ExclusiveAddressUse property.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when attempting to access the
    //underlying socket.
//
//   System.ObjectDisposedException:
//     The underlying System.Net.Sockets.Socket has been
    //closed.
public bool ExclusiveAddressUse { get; set; }
//
// Summary:
//     Gets the underlying System.Net.EndPoint of the
    //current System.Net.Sockets.TcpListener.
//
// Returns:
//     The System.Net.EndPoint to which the
    //System.Net.Sockets.Socket is bound.
public EndPoint LocalEndpoint { get; }
//
// Summary:
//     Gets the underlying network
    //System.Net.Sockets.Socket.
//
// Returns:
//     The underlying System.Net.Sockets.Socket.
public Socket Server { get; }

// Summary:
//     Accepts a pending connection request.
//
// Returns:
//A System.Net.Sockets.Socket used to send and receive data.
//
// Exceptions:
//   System.InvalidOperationException:
//     The listener has not been started with a call to
    //System.Net.Sockets.TcpListener.Start().
public Socket AcceptSocket();
//
// Summary:
//     Accepts a pending connection request
//
// Returns:
//     A System.Net.Sockets.TcpClient used to send and
```

```
    //receive data.
//
// Exceptions:
//   System.InvalidOperationException:
//     The listener has not been started with a call to
    //System.Net.Sockets.TcpListener.Start().
//
//   System.Net.Sockets.SocketException:
//     Use the System.Net.Sockets.SocketException.ErrorCode
    //property to obtain the
//     specific error code. When you have obtained this
    //code, you can refer to the
//     Windows Sockets version 2 API error code
    //documentation in MSDN for a detailed
//     description of the error.
public TcpClient AcceptTcpClient();
//
// Summary:
//     Begins an asynchronous operation to accept an
    //incoming connection attempt.
//
// Parameters:
//   callback:
//     An System.AsyncCallback delegate that references the
    //method to invoke when
//     the operation is complete.
//
//   state:
//     A user-defined object containing information about
    //the accept operation.
//     This object is passed to the callback delegate when
    //the operation is complete.
//
// Returns:
//     An System.IAsyncResult that references the
    //asynchronous creation of the
    //System.Net.Sockets.Socket.
//
// Exceptions:
//   System.Net.Sockets.SocketException:
//     An error occurred while attempting to access the
    //socket. See the Remarks
//     section for more information.
//
//   System.ObjectDisposedException:
//     The System.Net.Sockets.Socket has been closed.
public IAsyncResult BeginAcceptSocket(AsyncCallback
                                      callback, object state);
//
// Summary:
//     Begins an asynchronous operation to accept an
    //incoming connection attempt.
//
// Parameters:
//   callback:
//     An System.AsyncCallback delegate that references the
    //method to invoke when
//     the operation is complete.
//
//   state:
//     A user-defined object containing information about
```

```
    //the accept operation.
//     This object is passed to the callback delegate when
    //the operation is complete.
//
// Returns:
//     An System.IAsyncResult that references the
    //asynchronous creation of the
    //System.Net.Sockets.TcpClient.
//
// Exceptions:
//   System.Net.Sockets.SocketException:
//     An error occurred while attempting to access the
    //socket. See the Remarks
//     section for more information.
//
//   System.ObjectDisposedException:
//     The System.Net.Sockets.Socket has been closed.
public IAsyncResult BeginAcceptTcpClient(AsyncCallback
                                    callback, object state);
//
// Summary:
//     Asynchronously accepts an incoming connection attempt
    //and creates a new System.Net.Sockets.Socket
//     to handle remote host communication.
//
// Parameters:
//   asyncResult:
//     An System.IAsyncResult returned by a call to the
    //System.Net.Sockets.TcpListener.BeginAcceptSocket
    //(System.AsyncCallback,System.Object)
//     method.
//
// Returns:
//     A System.Net.Sockets.Socket.
//
// Exceptions:
//   System.ObjectDisposedException:
// The underlying System.Net.Sockets.Socket has been closed.
//
//   System.ArgumentNullException:
//     The asyncResult parameter is null.
//
//   System.ArgumentException:
//     The asyncResult parameter was not created by a call
//to the System.Net.Sockets.TcpListener.BeginAcceptSocket
//(System.AsyncCallback,System.Object)
//     method.
//
//   System.InvalidOperationException:
//     The System.Net.Sockets.TcpListener.EndAcceptSocket
    //(System.IAsyncResult) method
//     was previously called.
//
//   System.Net.Sockets.SocketException:
//     An error occurred while attempting to access the
    //System.Net.Sockets.Socket.
//     See the Remarks section for more information.
public Socket EndAcceptSocket(IAsyncResult asyncResult);
//
// Summary:
//     Asynchronously accepts an incoming connection attempt
```

```
    //and creates a new System.Net.Sockets.TcpClient
//     to handle remote host communication.
//
// Parameters:
//   asyncResult:
//     An System.IAsyncResult returned by a call to the
//System.Net.Sockets.TcpListener.BeginAcceptTcpClient
//(System.AsyncCallback,System.Object)
//     method.
//
// Returns:
//     A System.Net.Sockets.TcpClient.
public TcpClient EndAcceptTcpClient(IAsyncResult
                                                asyncResult);
//
// Summary:
//     Determines if there are pending connection requests.
//
// Returns:
//     true if connections are pending; otherwise, false.
//
// Exceptions:
//   System.InvalidOperationException:
//     The listener has not been started with a call to
    //System.Net.Sockets.TcpListener.Start().
public bool Pending();
//
// Summary:
//     Starts listening for incoming connection requests.
//
// Exceptions:
//   System.Net.Sockets.SocketException:
//     Use the System.Net.Sockets.SocketException.ErrorCode
    //property to obtain the
//     specific error code. When you have obtained this
    //code, you can refer to the
//     Windows Sockets version 2 API error code
    //documentation in MSDN for a detailed
//     description of the error.
public void Start();
//
// Summary:
//     Starts listening for incoming connection requests
    //with a maximum number of
//     pending connection.
//
// Parameters:
//   backlog:
//     The maximum length of the pending connections queue.
//
// Exceptions:
//   System.Net.Sockets.SocketException:
//     An error occurred while accessing the socket. See the
    //Remarks section for
//     more information.
//
//   System.ArgumentOutOfRangeException:
//     The backlog parameter is less than zero or exceeds
    //the maximum number of
//     permitted connections.
//
```

```
        //   System.InvalidOperationException:
        //     The underlying System.Net.Sockets.Socket is null.
        public void Start(int backlog);
        //
        // Summary:
        //     Closes the listener.
        //
        // Exceptions:
        //   System.Net.Sockets.SocketException:
        //     Use the System.Net.Sockets.SocketException.ErrorCode
            //property to obtain the
        //     specific error code. When you have obtained this
            //code, you can refer to the
        //     Windows Sockets version 2 API error code
            //documentation in MSDN for a detailed
        //     description of the error.
        public void Stop();
    }
}
```

- Helper Classes:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MyServer
{

    public class LongTime
    {
        public int Hours;
        public int Minutes;
        public int Seconds;
        public int Milliseconds;

        public static LongTime GetLongTime()
        {
                DateTime now = DateTime.Now;
                LongTime lt = new LongTime();
                lt.Milliseconds = now.Millisecond;
                lt.Seconds      = now.Second;
                lt.Minutes      = now.Minute;
                lt.Hours        = now.Hour;
                return lt;
        }



        public int ToMillisecondes()
        {
            int ms = Milliseconds + (Seconds * 1000)
                     + (Minutes * 60 * 1000)
                     + (Hours * 60 * 60 * 1000);
            return ms;
        }
```

```
        public override string ToString()
        {
            string longTime = String.Format("{0:00}", Hours) + ":" +
                  String.Format("{0:00}", Minutes)
                  + ":" + String.Format("{0:00}", Seconds) + ":" +
                  String.Format("{0:000}", Milliseconds);
            return longTime;
        }
    }

    public class ManualStopExcption : Exception { }

    public class TraceRecord
    {
        public int Conn_ID;

        public int Msg_Count;

        public int Duration_MSs;

        public static TraceRecord Create(int index, int
                                                duration_MSs)
        {
            TraceRecord tr = new TraceRecord();
            tr.Conn_ID = index;
            tr.Duration_MSs = duration_MSs;
            return tr;
        }
        public override string ToString()
        {
            return Conn_ID.ToString() + "\t" +
                              (Duration_MSs).ToString();
        }
    }

    public static ThrouputRecord Create(int client_ID,
            int msg_Count, int duration_MSs)
    {
        ThrouputRecord tr = new ThrouputRecord();
        tr.Client_ID = client_ID;
        tr.Msg_Count = msg_Count;
        tr.Duration_MSs = duration_MSs;
        if (tr.Duration_MSs != 0)
            tr.Throuput = tr.Msg_Count / tr.Duration_MSs;
        else
            tr.Throuput = 0;
        return tr;
    }

    public override string ToString()
    {
        return Client_ID.ToString() + ", " +
               Msg_Count.ToString() + ", " +
               Duration_MSs.ToString() + ", " +
               Throuput.ToString();
    }
}
```

Appendix B: TG Source Code

- Client Class:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.Net.Sockets;
using System.Net;

namespace MyClient
{
    enum TrafficType
    {
        Synchronous,
        Asynchronous
    }

    public partial class MyClientForm : Form
    {
        private static volatile List<Thread> AllThreads =
                                        new List<Thread>();

        private static volatile SynchronizationContext context;
        private static volatile Thread ProgressingThread;
        private static volatile TrafficType traffic =
                                        TrafficType.Synchronous;
        private static volatile bool StopFlag = false;
        private static volatile IPAddress serverIP;
        private static volatile int serverPort = 0;
        private static volatile int conn_no = 0;
        private static volatile int start_time = 0;
        private static volatile int sim_time = 0;
        private static volatile int interval_time = 0;
        private static volatile int bufferSize = 0;
        private static volatile int minRandom = 0;
        private static volatile int maxRandom = 0;

        public MyClientForm()
        {
            InitializeComponent();
            context = SynchronizationContext.Current;
            if (context == null) context =
                            new SynchronizationContext();
        }

        private void btnGo_Click(object sender, EventArgs e)
        {
            ////////////////////////////////////////////////
            serverIP = IPAddress.Parse(ipAddressBox.Text);
            serverPort = int.Parse(portnoBox.Text);
            bufferSize = Convert.ToInt32(bufferSizeBox.Text) *
                            Convert.ToInt32(messageSizeBox.Text);
            StopFlag = false;
```

```
        progressBar1.Maximum = int.Parse(sTimeBox.Text);
        conn_no = decimal.ToInt32(conn_no_box.Value);
        start_time = LongTime.GetLongTime().ToMillisecondes();
        sim_time = int.Parse(sTimeBox.Text) * 1000;
        interval_time = int.Parse(intervalBox.Text);
        minRandom = int.Parse(minBox.Text);
        maxRandom = int.Parse(maxBox.Text);
        if ((AllThreads != null) && (AllThreads.Count > 0))
              AllThreads.Clear();
        ////////////////////////////////////////////////
        if (SynchronousRadioBtn.Checked)
            traffic = TrafficType.Synchronous;
        else
            if (AsynchronousRadioBtn.Checked)
                traffic = TrafficType.Asynchronous;
        ////////////////////////////////////////////////
              Thread startThreads = new Thread(new
                                  ThreadStart(ThreadsCreator));
              startThreads.IsBackground = true;
              startThreads.Start();
    }

        private void ThreadsCreator()
        {
        for (int i = 0; i < conn_no; i++)
        {
            Thread thr = new Thread(new
                                  ThreadStart(StartClientWork));
            thr.IsBackground = true;
            thr.Start();
            AllThreads.Add(thr);
        }
        ProgressingThread = new Thread(new
                                  ThreadStart(Progressing));
        ProgressingThread.IsBackground = true;
        ProgressingThread.Start();
        }

    private void Progressing()
    {
        try
        {
            int my_elapsed_time = 0;
            int my_sim_time = sim_time;
            int my_start_time = start_time;
            while (my_elapsed_time < my_sim_time)
            {
                my_elapsed_time =
                    LongTime.GetLongTime().ToMillisecondes() -
                    my_start_time;
                context.Send(new SendOrPostCallback((s) =>
                          progressBar1.Value = my_elapsed_time /
                          1000), null);
                context.Send(new SendOrPostCallback((s) =>
                          timeLabel.Text = (my_elapsed_time /
                          1000).ToString() + " second(s).",
                          null);
                if (StopFlag) throw new ManualStopExcption();
                Thread.Sleep(100);
            }
        }
```

```
    catch { }
}

private void StartClientWork()
{
    TcpClient client = new TcpClient();
    try
    {
        client.Connect(serverIP, serverPort);
        string message =
                "01234567890123456789012345678 9” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789" +
                "012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789" +
                "012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789" +
                "012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789" +
                "012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789" +
                "012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789" +
                "012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789" +
                "012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789” +
                “012345678901234567890123456789";


        client.SendBufferSize = bufferSize;
        NetworkStream ns = client.GetStream();
        int my_start_time = start_time;
        int my_sim_time = sim_time;
        int my_interval_time = interval_time;
        int my_elapsed_time = 0;
        TrafficType my_traffic = traffic;
        byte[] data = Encoding.ASCII.GetBytes(message);
        Random random = new Random();
```

```
            while (my_elapsed_time < my_sim_time)
            {
                if (StopFlag) throw new ManualStopExcption();
                ns.Write(data, 0, data.Length);
                my_elapsed_time =
                    LongTime.GetLongTime().ToMillisecondes() -
                    my_start_time;

                switch (my_traffic)
                {
                    case TrafficType.Synchronous:
                        {
                            Thread.Sleep(my_interval_time);
                        }
                        break;
                    case TrafficType.Asynchronous:
                        {
                            Thread.Sleep(random.Next(minRandom,
                                                     maxRandom));
                        }
                        break;
                    default:
                        break;
                }
            }
            client.Close();
            AllThreads.Remove(Thread.CurrentThread);
        }
        catch (Exception ex)
        {
            if (ex is ManualStopExcption)
            {
                client.Close();
                AllThreads.Remove(Thread.CurrentThread);
            }
            else
                MessageBox.Show(" The Server may not ready.");
        }
    }

    private void btnCancel_Click(object sender, EventArgs e)
    {
        StopFlag = true;
    }

    private void MyClientForm_Load(object sender, EventArgs e)
    {
        ipAddressBox.Text = "60.10.10.1";
        portnoBox.Text = "50500";
        HeavyRadioBtn_Click(sender, e);
    }

    private void MyClientForm_FormClosed(object sender,
                                         FormClosedEventArgs e)
    {
        if (AllThreads != null) foreach (Thread thr in
                                         AllThreads) thr.Abort();
        AllThreads.Clear();
    }

    private void btnReset_Click(object sender, EventArgs e)
```

```
        {
            StopFlag = false;
            progressBar1.Value = 0;
            timeLabel.Text = "0 second(s).";
            if (AllThreads != null)
                  if (AllThreads.Count < 0)
                        foreach (var item in AllThreads)
                              item.Abort();
            AllThreads.Clear();
            if (ProgressingThread != null)
                  ProgressingThread.Abort();
        }

        private void HeavyRadioBtn_Click(object sender, EventArgs e)
        {
            if (SynchronousRadioBtn.Checked)
            {
                intervalBox.Enabled = true;
                minBox.Enabled = false;
                maxBox.Enabled = false;
            }
        }

        private void UniformRadioBtnClick(object sender,EventArgs e)
        {
            if (AsynchronousRadioBtn.Checked)
            {
                intervalBox.Enabled = false;
                minBox.Enabled = true;
                maxBox.Enabled = true;
            }
        }
    }
}
```

- TcpClient Class:

```
using System;
using System.Net;

namespace System.Net.Sockets
{
    // Summary:
    //     Provides client connections for TCP network services.
    public class TcpClient : IDisposable
    {
        // Summary:
        //     Initializes a new instance of the
        //     System.Net.Sockets.TcpClient class.
        public TcpClient();
        //
        // Summary:
        //     Initializes a new instance of the
        //     System.Net.Sockets.TcpClient class with
        //     the specified family.
        //
        // Parameters:
        //   family:
        //     The System.Net.IPAddress.AddressFamily
        //     of the IP protocol.
```

```
//
// Exceptions:
//   System.ArgumentException:
//     The family parameter is not equal to
//     AddressFamily.InterNetwork -or- The
//     family parameter is not equal to
//     AddressFamily.InterNetworkV6
public TcpClient(AddressFamily family);
//
// Summary:
//     Initializes a new instance of the
//     System.Net.Sockets.TcpClient class and
//     binds it to the specified local endpoint.
//
// Parameters:
//   localEP:
//     The System.Net.IPEndPoint to which you bind the TCP
//     System.Net.Sockets.Socket.
//
// Exceptions:
//   System.ArgumentNullException:
//     The localEP parameter is null.
public TcpClient(IPEndPoint localEP);
//
// Summary:
//     Initializes a new instance of the
//     System.Net.Sockets.TcpClient class and
//     connects to the specified port on the specified host.
//
// Parameters:
//   hostname:
//     The DNS name of the remote host to which you intend
//     to connect.
//
//   port:
//     The port number of the remote host to which you
//     intend to connect.
//
// Exceptions:
//   System.ArgumentNullException:
//     The hostname parameter is null.
//
//   System.ArgumentOutOfRangeException:
//     The port parameter is not between
//  System.Net.IPEndPoint.MinPort and
//  System.Net.IPEndPoint.MaxPort.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when accessing the socket. See the
//        Remarks section for
//     more information.
public TcpClient(string hostname, int port);

// Summary:
//     Gets or set a value that indicates whether a
//        connection has been made.
//
// Returns:
//     true if the connection has been made; otherwise,
//        false.
protected bool Active { get; set; }
```

```
//
// Summary:
//     Gets the amount of data that has been received from
//        the network and is available
//     to be read.
//
// Returns:
//     The number of bytes of data received from the network
//        and available to be read.
//
// Exceptions:
//   System.Net.Sockets.SocketException:
//     An error occurred when attempting to access the
//        socket. See the Remarks section
//     for more information.
//
//   System.ObjectDisposedException:
//     The System.Net.Sockets.Socket has been closed.
public int Available { get; }
//
// Summary:
//     Gets or sets the underlying
//     System.Net.Sockets.Socket.
//
// Returns:
//     The underlying network System.Net.Sockets.Socket.
public Socket Client { get; set; }
//
// Summary:
//     Gets a value indicating whether the underlying
//  System.Net.Sockets.Socket
//     for a System.Net.Sockets.TcpClient is connected to a
//        remote host.
//
// Returns:
//     true if the System.Net.Sockets.TcpClient.Client
    //socket was connected to a
//     remote resource as of the most recent operation;
//otherwise, false.
public bool Connected { get; }
//
// Summary:
//     Gets or sets a System.Boolean value that specifies
    //whether the System.Net.Sockets.TcpClient
//     allows only one client to use a port.
//
// Returns:
//     true if the System.Net.Sockets.TcpClient allows only
    //one client to use a
//     specific port; otherwise, false. The default is true
    //for Windows Server 2003
//     and Windows XP Service Pack 2 and later, and false
    //for all other versions.
//
// Exceptions:
//   System.Net.Sockets.SocketException:
//     An error occurred when attempting to access the
    //underlying socket.
//
//   System.ObjectDisposedException:
//     The underlying System.Net.Sockets.Socket has been
```

```
    //closed.
public bool ExclusiveAddressUse { get; set; }
//
// Summary:
//  Gets or sets information about the sockets linger time.
//
// Returns:
//     A System.Net.Sockets.LingerOption. By default,
    //lingering is disabled.
public LingerOption LingerState { get; set; }
//
// Summary:
//     Gets or sets a value that disables a delay when send
    //or receive buffers are
//     not full.
//
// Returns:
//     true if the delay is disabled, otherwise false. The
    //default value is false.
public bool NoDelay { get; set; }
//
// Summary:
//     Gets or sets the size of the receive buffer.
//
// Returns:
//     The size of the receive buffer, in bytes. The default
    //value is 8192 bytes.
//
// Exceptions:
//   System.Net.Sockets.SocketException:
//     An error occurred when setting the buffer size.  -or-
    //In .NET Compact Framework
//     applications, you cannot set this property. For a
    //workaround, see the Platform
//     Note in Remarks.
public int ReceiveBufferSize { get; set; }
//
// Summary:
//     Gets or sets the amount of time a
    //System.Net.Sockets.TcpClient will wait
//     to receive data once a read operation is initiated.
//
// Returns:
//     The time-out value of the connection in milliseconds.
    //The default value is 0.
public int ReceiveTimeout { get; set; }
//
// Summary:
//     Gets or sets the size of the send buffer.
//
// Returns:
//     The size of the send buffer, in bytes. The default
    //value is 8192 bytes.
public int SendBufferSize { get; set; }
//
// Summary:
//     Gets or sets the amount of time a
    //System.Net.Sockets.TcpClient will wait
//     for a send operation to complete successfully.
//
// Returns:
```

```
// The send time-out value,in milliseconds.The default is 0.
public int SendTimeout { get; set; }

// Summary:
//     Begins an asynchronous request for a remote host
    //connection. The remote host
//     is specified by an System.Net.IPAddress and a port
    //number (System.Int32).
//
// Parameters:
//   address:
//     The System.Net.IPAddress of the remote host.
//
//   port:
//     The port number of the remote host.
//
//   requestCallback:
//     An System.AsyncCallback delegate that references the
    //method to invoke when
//     the operation is complete.
//
//   state:
//     A user-defined object that contains information about
    //the connect operation.
//     This object is passed to the requestCallback delegate
    //when the operation
//     is complete.
//
// Returns:
//     An System.IAsyncResult object that references the
    //asynchronous connection.
//
// Exceptions:
//   System.ArgumentNullException:
//     The address parameter is null.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when attempting to access the
    //socket. See the Remarks section
//     for more information.
//
//   System.ObjectDisposedException:
//     The System.Net.Sockets.Socket has been closed.
//
//   System.Security.SecurityException:
//     A caller higher in the call stack does not have
    //permission for the requested
//     operation.
//
//   System.ArgumentOutOfRangeException:
//     The port number is not valid.
public IAsyncResult BeginConnect(IPAddress address, int
                            port, AsyncCallback
                            requestCallback, object state);
//
// Summary:
//     Begins an asynchronous request for a remote host
    //connection. The remote host
//     is specified by an System.Net.IPAddress array and a
    //port number (System.Int32).
//
```

```
// Parameters:
//   addresses:
//     At least one System.Net.IPAddress that designates the
    //remote hosts.
//
//   port:
//     The port number of the remote hosts.
//
//   requestCallback:
//     An System.AsyncCallback delegate that references the
    //method to invoke when
//     the operation is complete.
//
//   state:
//     A user-defined object that contains information about
    //the connect operation.
//     This object is passed to the requestCallback delegate
    //when the operation
//     is complete.
//
// Returns:
//     An System.IAsyncResult object that references the
    //asynchronous connection.
//
// Exceptions:
//   System.ArgumentNullException:
//     The addresses parameter is null.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when attempting to access the
    //socket. See the Remarks section
//     for more information.
//
//   System.ObjectDisposedException:
//     The System.Net.Sockets.Socket has been closed.
//
//   System.Security.SecurityException:
//     A caller higher in the call stack does not have
    //permission for the requested
//     operation.
//
//   System.ArgumentOutOfRangeException:
//     The port number is not valid.
public IAsyncResult BeginConnect(IPAddress[] addresses, int
                                        port, AsyncCallback
                                        requestCallback,
                                        object state);
//
// Summary:
//     Begins an asynchronous request for a remote host
    //connection. The remote host
//     is specified by a host name (System.String) and a
    //port number (System.Int32).
//
// Parameters:
//   host:
//     The name of the remote host.
//
//   port:
//     The port number of the remote host.
//
```

```
//   requestCallback:
//     An System.AsyncCallback delegate that references the
    //method to invoke when
//     the operation is complete.
//
//   state:
//     A user-defined object that contains information about
    //the connect operation.
//     This object is passed to the requestCallback delegate
    //when the operation
//     is complete.
//
// Returns:
//     An System.IAsyncResult object that references the
    //asynchronous connection.
//
// Exceptions:
//   System.ArgumentNullException:
//     The host parameter is null.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when attempting to access the
    //socket. See the Remarks section
//     for more information.
//
//   System.ObjectDisposedException:
//     The System.Net.Sockets.Socket has been closed.
//
//   System.Security.SecurityException:
//     A caller higher in the call stack does not have
    permission for the requested
//     operation.
//
//   System.ArgumentOutOfRangeException:
//     The port number is not valid.
public IAsyncResult BeginConnect(string host, int port,
                                 AsyncCallback requestCallback,
                                 object state);
//
// Summary:
//     Disposes this System.Net.Sockets.TcpClient instance
    //without closing the underlying
//     connection.
public void Close();
//
// Summary:
//     Connects the client to a remote TCP host using the
    //specified remote network
//     endpoint.
//
// Parameters:
//   remoteEP:
// The System.Net.IPEndPoint to which you intend to connect.
//
// Exceptions:
//   System.ArgumentNullException:
//     The remoteEp parameter is null.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when accessing the socket. See the
    //Remarks section for
```

```
//     more information.
//
//   System.ObjectDisposedException:
//     The System.Net.Sockets.TcpClient is closed.
public void Connect(IPEndPoint remoteEP);
//
// Summary:
//     Connects the client to a remote TCP host using the
    //specified IP address and
//     port number.
//
// Parameters:
//   address:
//     The System.Net.IPAddress of the host to which you
    //intend to connect.
//
//   port:
//     The port number to which you intend to connect.
//
// Exceptions:
//   System.ArgumentNullException:
//     The address parameter is null.
//
//   System.ArgumentOutOfRangeException:
//     The port is not between System.Net.IPEndPoint.MinPort
    //and System.Net.IPEndPoint.MaxPort.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when accessing the socket. See the
    //Remarks section for
//     more information.
//
//   System.ObjectDisposedException:
//     System.Net.Sockets.TcpClient is closed.
public void Connect(IPAddress address, int port);
//
// Summary:
//     Connects the client to a remote TCP host using the
    //specified IP addresses
//     and port number.
//
// Parameters:
//   ipAddresses:
//     The System.Net.IPAddress array of the host to which
    //you intend to connect.
//
//   port:
//     The port number to which you intend to connect.
//
// Exceptions:
//   System.ArgumentNullException:
//     The ipAddresses parameter is null.
//
//   System.ArgumentOutOfRangeException:
//     The port number is not valid.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when attempting to access the
    //socket. See the Remarks section
//     for more information.
//
```

```
//   System.ObjectDisposedException:
//     The System.Net.Sockets.Socket has been closed.
//
//   System.Security.SecurityException:
//     A caller higher in the call stack does not have
    //permission for the requested
//     operation.
//
//   System.NotSupportedException:
//     This method is valid for sockets that use the
    //System.Net.Sockets.AddressFamily.InterNetwork
//     flag or the
    //System.Net.Sockets.AddressFamily.InterNetworkV6 flag.
public void Connect(IPAddress[] ipAddresses, int port);
//
// Summary:
//     Connects the client to the specified port on the
    //specified host.
//
// Parameters:
//   hostname:
//     The DNS name of the remote host to which you intend
    //to connect.
//
//   port:
//     The port number of the remote host to which you
    intend to connect.
//
// Exceptions:
//   System.ArgumentNullException:
//     The hostname parameter is null.
//
//   System.ArgumentOutOfRangeException:
//     The port parameter is not between
    //System.Net.IPEndPoint.MinPort and
    //System.Net.IPEndPoint.MaxPort.
//
//   System.Net.Sockets.SocketException:
//     An error occurred when accessing the socket. See the
    //Remarks section for
//     more information.
//
//   System.ObjectDisposedException:
//     System.Net.Sockets.TcpClient is closed.
public void Connect(string hostname, int port);
//
// Summary:
//     Releases the unmanaged resources used by the
    //System.Net.Sockets.TcpClient
//     and optionally releases the managed resources.
//
// Parameters:
//   disposing:
//     Set to true to release both managed and unmanaged
    //resources; false to release
//     only unmanaged resources.
protected virtual void Dispose(bool disposing);
//
// Summary:
//   Asynchronously accepts an incoming connection attempt.
//
```

```
        // Parameters:
        //   asyncResult:
        //     An System.IAsyncResult object returned by a call to
            //Overload:System.Net.Sockets.TcpClient.BeginConnect.
        //
        // Exceptions:
        //   System.ArgumentNullException:
        //     The asyncResult parameter is null.
        //
        //   System.ArgumentException:
        //     The asyncResult parameter was not returned by a call
        //to a Overload:System.Net.Sockets.TcpClient.BeginConnect
        //     method.
        //
        //   System.InvalidOperationException:
        //     TheSystem.Net.Sockets.TcpClient.EndConnect
            //(System.IAsyncResult) method was
        //     previously called for the asynchronous connection.
        //
        //   System.Net.Sockets.SocketException:
        //     An error occurred when attempting to access the
            //System.Net.Sockets.Socket.
        //     See the Remarks section for more information.
        //
        //   System.ObjectDisposedException:
        // The underlying System.Net.Sockets.Socket has been closed.
        public void EndConnect(IAsyncResult asyncResult);
        //
        // Summary:
        //     Returns the System.Net.Sockets.NetworkStream used to
            //send and receive data.
        //
        // Returns:
        //     The underlying System.Net.Sockets.NetworkStream.
        //
        // Exceptions:
        //   System.InvalidOperationException:
        //     The System.Net.Sockets.TcpClient is not connected to
            //a remote host.
        //
        //   System.ObjectDisposedException:
        //     The System.Net.Sockets.TcpClient has been closed.
        public NetworkStream GetStream();
    }
}
```

Appendix C: Samples of Files

- AWK File Sample

```
BEGIN{ sth = 0; stm = 0; sts = 0; stf = 0; flag = 0{
}
if (flag == 0}(
    sth = 1$1
```

```
    stm = 2$2
    sts = 3$3
    stf = 4$4
    flag = 11
{
if (($6 == "60.10.10.2") && ($11 != "F") && ($11 != "P") && ($11 !=
"FP") && ($11 != "ack") && ($11 != "S") && ($12 != "PTR}(("?
    x1 = $1 - sth:
    x2 = $2 - stm:
    x3 = $3 - sts:
    x4 = $4 - stf:
    newts = (x1 * 60 * 60 * 1000000) + (x2 * 60 * 1000000) + (x3 *
1000000) + (x4:(
    print newts/100000 " " $6 " " $7 " " $11 " " $13
{
if (($6 == "60.10.10.2") && ($11 == "P}(("
    x1 = $1 - sth:
    x2 = $2 - stm:
    x3 = $3 - sts:
    x4 = $4 - stf:
    newts = (x1 * 60 * 60 * 1000000) + (x2 * 60 * 1000000) + (x3 *
1000000) + (x4:(
    print newts/100000 " " $6 " " $7 " " $12 " " $14
{
if (($6 == "60.10.10.2") && ($11 == "FP}(("
    x1 = $1 - sth:
    x2 = $2 - stm:
    x3 = $3 - sts:
    x4 = $4 - stf:
    newts = (x1 * 60 * 60 * 1000000) + (x2 * 60 * 1000000) + (x3 *
1000000) + (x4:(
    print newts/100000 " " $6 " " $7 " " $12 " " $14
{
if (($6 == "60.10.10.2") && ($11 == "F}(("
    x1 = $1 - sth:
    x2 = $2 - stm:
    x3 = $3 - sts:
    x4 = $4 - stf:
    newts = (x1 * 60 * 60 * 1000000) + (x2 * 60 * 1000000) + (x3 *
1000000) + (x4:(
    print newts/1000000 " " $6 " " $7 " " $12 " " $14
{
{
END{}
```

- Patch File Sample (Fairness):

```
echo "cubic fairness"
awk -f awk/f.awk r-cubic/r-cubic1.txt
awk -f awk/f.awk r-cubic/r-cubic5.txt
awk -f awk/f.awk r-cubic/r-cubic10.txt
awk -f awk/f.awk r-cubic/r-cubic15.txt
awk -f awk/f.awk r-cubic/r-cubic20.txt
awk -f awk/f.awk r-cubic/r-cubic25.txt
awk -f awk/f.awk r-cubic/r-cubic30.txt

echo "highspeed fairness"
```

```
awk -f awk/f.awk r-highspeed/r-hs1.txt
awk -f awk/f.awk r-highspeed/r-hs5.txt
awk -f awk/f.awk r-highspeed/r-hs10.txt
awk -f awk/f.awk r-highspeed/r-hs15.txt
awk -f awk/f.awk r-highspeed/r-hs20.txt
awk -f awk/f.awk r-highspeed/r-hs25.txt
awk -f awk/f.awk r-highspeed/r-hs30.txt

echo "htcp fairness"
awk -f awk/f.awk r-htcp/r-htcp1.txt
awk -f awk/f.awk r-htcp/r-htcp5.txt
awk -f awk/f.awk r-htcp/r-htcp10.txt
awk -f awk/f.awk r-htcp/r-htcp15.txt
awk -f awk/f.awk r-htcp/r-htcp20.txt
awk -f awk/f.awk r-htcp/r-htcp25.txt
awk -f awk/f.awk r-htcp/r-htcp30.txt

echo "reno fairness"
awk -f awk/f.awk r-reno/r-reno1.txt
awk -f awk/f.awk r-reno/r-reno5.txt
awk -f awk/f.awk r-reno/r-reno10.txt
awk -f awk/f.awk r-reno/r-reno15.txt
awk -f awk/f.awk r-reno/r-reno20.txt
awk -f awk/f.awk r-reno/r-reno25.txt
awk -f awk/f.awk r-reno/r-reno30.txt

echo "scalable fairness"
awk -f awk/f.awk r-scalable/r-scalable1.txt
awk -f awk/f.awk r-scalable/r-scalable5.txt
awk -f awk/f.awk r-scalable/r-scalable10.txt
awk -f awk/f.awk r-scalable/r-scalable15.txt
awk -f awk/f.awk r-scalable/r-scalable20.txt
awk -f awk/f.awk r-scalable/r-scalable25.txt
awk -f awk/f.awk r-scalable/r-scalable30.txt
```