Improvement of TCP Congestion Control Mechanism over Computer Networks

A Thesis submitted in accordance with the requirement of the University of Menofyia for the degree of Master

In
Electronic Engineering
[Computer Engineering and Science]

By
Eng. Shaymaa Ibrahim Moustafa Haggag
Engineer in South Delta Electricity Company, Cairo, Egypt.

Supervised by

Prof. Dr. Nawal Ahmed EL-FESHAWY
Dept. of Computer Engineering and Science, Faculty of Electronic Engineering, Menouf Menofyia University.

Assoc. Prof. Dr. Ayman EL-SAYED
Dept. of Computer Engineering and Science, Faculty of Electronic Engineering, Menouf Menofyia University.

2012
Acknowledgments

First of all, thank Allah for guarding and guiding me all the times, and giving me power, strength and patience to finish this work. I would like to express my sincere gratitude to my adviser, Dr. Ayman EL-SAYED, for his guidance and support throughout my research. Who led me into this exciting area of networking research? I am privileged to have such a wonderful advisor, who is always enthusiastic, inspired, patient, helpful and encouraging. All I can say is that I can hardly ask for any more from an advisor. A special thanks goes to my advisor Professor Dr. Nawal EL-Feshawy, I have benefited from her detailed comments on every aspects of my work. I wish to thank my family for their constant love, support and encouragement, they support me during the various difficulties and hard times, and their support makes my life meaningful and enjoyable. I would like to thank my friend Hanaa Torkey, I have received a lot of assistance from her I hope to her a happy life. Last but not the least, I wish sincerely to express my gratitude to all the people who supported me.

Shimaa Hagag
Minoufiya University
2012
Our Achievements


ABSTRACT

Transport Control Protocol (TCP), the mostly used transport protocol, performs well over wired networks. As much as wireless network is deployed, TCP should be modified to work for both wired and wireless networks. Since TCP is designed for congestion control in wired networks, it cannot clearly detect non-congestion related packet loss from wireless networks. TCP Congestion control plays the key role to ensure stability of the Internet along with fair and efficient allocation of the bandwidth. So, congestion control is currently a large area of research and concern in the network community. Many congestion control mechanisms are developed and refined by researcher aiming to overcome congestion. During the last decade, several congestion control mechanisms have been proposed to improve TCP congestion control. Comparing these mechanisms, showing their differences and their improvements, and we identify, classify, and discuss some of these mechanisms of TCP congestion control such as Tahoe, Sack, Reno, NewReno, Vegas, and Westwood. TCP Westwood works for both wired and wireless network, and we propose a new algorithm called TCP WestwoodNew to increase the performance of TCP-Westwood. By enhanced the congestion avoidance of TCP Westwood by a new estimation to cwnd algorithm based on the network status. Also TCP WestwoodNew introduces a new estimation for Retransmission TimeOuts (RTO). RTO has been reported to be a problem on network paths involving links that are prone to sudden delays due to various reasons. Especially many wireless network
technologies contain such links. Spurious RTO often cause unnecessary retransmission of several segments, which is harmful for TCP performance, and unnecessary retransmissions can be avoided. We simulate the proposed algorithm TCP WestwoodNew using the well known network simulator ns-2, by comparing it to the original TCP-Westwood. Simulation results show that the proposed scheme achieves better throughput than TCP Westwood and decreases the delay.
Table of Contents

List of Tables ........................................................................................................... xx
List of Figures ............................................................................................................ xx
Abbreviations ........................................................................................................... xx

Chapter 1: Introduction .............................................................................................. xx
  1.1 Open System Interconnection Reference Mode .............................................. xx
  1.2 Transmission Control Protocol (TCP) .......................................................... xx
  1.3 Congestion Problem ....................................................................................... xx
  1.4 Congestion Control ......................................................................................... xx
  1.5 Survey of TCP Congestion Control ............................................................... xx
  1.6 Research Objectives ....................................................................................... xx
  1.7 Thesis Organization ......................................................................................... xx

Chapter 2: Transmission Control Protocol (TCP) .................................................. xx
  2.1 Connection – Oriented .................................................................................... xx
  2.2 Reliability ........................................................................................................ xx
  2.3 Byte Stream Delivery ...................................................................................... xx
  2.4 TCP Segment Format .................................................................................... xx
  2.5 TCP Flow Control ........................................................................................... xx
  2.6 TCP Time-Out Mechanism ............................................................................ xx
  2.7 TCP Congestion Control in The Internet ...................................................... xx
  2.8 Additive - Increase, Multiplicative –Decrease ............................................... xx
  2.9 Reaction to Timeout Events ........................................................................... xx

Chapter 3: TCP Congestion Control Algorithms ................................................... xx
  3.1 Slow Start ........................................................................................................ xx
  3.2 Congestion Avoidance ................................................................................... xx
  3.3 Fast Retransmit ................................................................................................ xx
**Table of Contents**

3.4 Fast Recovery .......................................................... xx
3.5 TCP's Retransmission Timeout Mechanism—Related Work .......... xx
3.6 Initial RTO Value – Examination of Related Work ................... xx

**Chapter 4: TCP Congestion Control Mechanisms .................. xx**

3.1 TCP Tahoe ................................................................ xx
3.2 TCP Reno ................................................................ xx
3.3 TCP NewReno ............................................................... xx
   3.3.1 Fast Retransmit&Fast Recovery Algorithms in NewReno ... xx
   3.3.2 Resetting the Retransmit Timer .................................. xx
   3.3.3 Avoiding Multiple Fast Retransmits ............................... xx
   3.3.4 Implementation Issues for the Data Receiver .................. xx
   3.3.5 Conclusion of NewReno Mechanism ............................. xx
3.4 TCP Vegas ................................................................ xx
3.5 TCP SACK ................................................................ xx
3.6 TCP Westwood .............................................................. xx
3.7 Conclusion of This Chapter .............................................. xx

**Chapter 5: TCP Westwood Overview ............................... xx**

5.1 Necessity of Bandwidth and its Management ......................... xx
5.2 Overview .................................................................. xx
5.3 Bandwidth Estimation .................................................. xx

**Chapter 6: Proposed Mechanism ................................... xx**

6.1 Principle Idea ............................................................... xx
6.2 Enhanced TCP Westwood Congestion Avoidance Algorithm ...... xx
6.3 Modified RTO Calculation Algorithm ................................ xx
6.4 WestwoodNew Mechanism Description ............................. xx
6.5 Implementation ............................................................ xx

**Chapter 7: Simulation Results and Discussions ................ xx**

7.1 Simulation Environments ............................................... xx
7.2 Evaluation Criteria ....................................................... xx
   7.2.1 Throughput .......................................................... xx
   7.2.2 Delay ................................................................ xx
   7.2.3 Packet losses ....................................................... xx
   7.2.4 Congestion Window ............................................... xx
7.3 Simulated Network Topologies & Simulation Results ............ xx
   7.3.1 Network Topology 1 (NT1) ....................................... xx
   7.3.2 Network Topology 2 (NT2) ....................................... xx

**Chapter 8: Conclusion and Future Work .......................... xx**
Table of Contents

8.1 Overview ........................................................................................................... xx
8.2 Concluding Remarks ......................................................................................... xx
8.3 Future Research Directions ............................................................................... xx

Appendix A: Network Simulator ............................................................................. xx
Appendix B: Network Topology Generator ............................................................ xx
References ............................................................................................................... xx
LIST OF TABLES

3.1 TCP Congestion Control Algorithms Pseudo code
3.2 TCP Congestion Control Algorithms
5.1 The WestwoodNew Congestion Control algorithms Pseudo code
# LIST OF FIGURES

1.1  The OSI reference model  
2.1  TCP connection establishment  
2.2  TCP segment header format  
2.3  Window flow control 'self-clocking'  
2.4  Additive-increase, multiplicative-decrease congestion control  
2.5  TCP Congestion Control  
3.1  TCP slow start and congestion avoidance behaviour in action  
3.2  TCP Fast retransmit algorithm  
3.3  TCP Fast Recovery Algorithm  
3.4  Current SACK option format  
3.5  TCP Inherence  
4.1  Bandwidth Management Activities  
4.2  The Bandwidth  
4.3  Bandwidth samples  
5.1  The architecture of the proposed mechanism  
5.2  flow chart of the proposed mechanism  
6.1  Network Topology1 (NT1)  
6.2  Throughput vs. Simulation Time for NT1  
6.3  Throughput versus time for Westwood and WestwoodNew for NT1  
6.4  Delay vs. Simulation Time for NT1  
6.5  Delay versus the time for Westwood and WestwoodNew for NT1  
6.6  Packet Losses vs. Time for NT1  
6.7  Packet Losses versus the time for Westwood and WestwoodNew for NT1
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8</td>
<td>Behaviour of Congestion Window (cwnd) for NT1</td>
</tr>
<tr>
<td>6.9</td>
<td>Behaviour of the Congestion Window for Westwood and WestwoodNew for NT1</td>
</tr>
<tr>
<td>6.10</td>
<td>AT&amp;T Network Topology2 (NT2)</td>
</tr>
<tr>
<td>6.11</td>
<td>Throughput vs. Simulation Time for NT2</td>
</tr>
<tr>
<td>6.12</td>
<td>Throughput versus time for Westwood and WestwoodNew for NT2</td>
</tr>
<tr>
<td>6.13</td>
<td>Delay vs. Simulation Time for NT2</td>
</tr>
<tr>
<td>6.14</td>
<td>Delay versus the time for Westwood and WestwoodNew for NT2</td>
</tr>
<tr>
<td>6.15</td>
<td>Packet Losses vs. Time for NT2</td>
</tr>
<tr>
<td>6.16</td>
<td>Packet Losses versus the time for Westwood and WestwoodNew for NT2</td>
</tr>
<tr>
<td>6.17</td>
<td>Behaviour of Congestion Window (cwnd) for NT2</td>
</tr>
<tr>
<td>6.18</td>
<td>Behaviour of the Congestion Window for Westwood and WestwoodNew for NT2</td>
</tr>
<tr>
<td>6.19</td>
<td>Network Topology of Scenario-2</td>
</tr>
<tr>
<td>6.20</td>
<td>Cwnd &amp; Ssthresh for TCP Tahoe case No Error</td>
</tr>
<tr>
<td>6.21</td>
<td>Cwnd &amp; Ssthresh for TCP Reno case No Error</td>
</tr>
<tr>
<td>6.22</td>
<td>Cwnd &amp; Ssthresh for TCP NewReno case No Error</td>
</tr>
<tr>
<td>6.23</td>
<td>Cwnd &amp; Ssthresh for TCP Sack case No Error</td>
</tr>
<tr>
<td>6.24</td>
<td>Cwnd &amp; Ssthresh for TCP Vegas case No Error</td>
</tr>
<tr>
<td>6.25</td>
<td>Cwnd &amp; Ssthresh for TCP Westwood case No Error</td>
</tr>
<tr>
<td>6.26</td>
<td>Cwnd &amp; Ssthresh for TCP WestwoodNew case No Error</td>
</tr>
<tr>
<td>6.27</td>
<td>Cwnd &amp; Ssthresh for TCP Tahoe case 1% Error</td>
</tr>
<tr>
<td>6.28</td>
<td>Cwnd &amp; Ssthresh for TCP Reno case 1% Error</td>
</tr>
<tr>
<td>6.29</td>
<td>Cwnd &amp; Ssthresh for TCP NewReno case 1% Error</td>
</tr>
<tr>
<td>6.30</td>
<td>Cwnd &amp; Ssthresh for TCP Sack case 1% Error</td>
</tr>
<tr>
<td>6.31</td>
<td>Cwnd &amp; Ssthresh for TCP Vegas case 1% Error</td>
</tr>
<tr>
<td>6.32</td>
<td>Cwnd &amp; Ssthresh for TCP Westwood case 1% Error</td>
</tr>
<tr>
<td>6.33</td>
<td>Cwnd &amp; Ssthresh for TCP WestwoodNew case 1% Error</td>
</tr>
<tr>
<td>6.34</td>
<td>Cwnd &amp; Ssthresh for TCP Tahoe case 2% Error</td>
</tr>
<tr>
<td>6.35</td>
<td>Cwnd &amp; Ssthresh for TCP Reno case 2% Error</td>
</tr>
<tr>
<td>6.36</td>
<td>Cwnd &amp; Ssthresh for TCP NewReno case 2% Error</td>
</tr>
<tr>
<td>6.37</td>
<td>Cwnd &amp; Ssthresh for TCP Sack case 2% Error</td>
</tr>
<tr>
<td>6.38</td>
<td>Cwnd &amp; Ssthresh for TCP Vegas case 2% Error</td>
</tr>
<tr>
<td>6.39</td>
<td>Cwnd &amp; Ssthresh for TCP Westwood case 2% Error</td>
</tr>
<tr>
<td>6.40</td>
<td>Cwnd &amp; Ssthresh for TCP WestwoodNew case 2% Error</td>
</tr>
</tbody>
</table>
Abbreviations

TCP  Transmission Control Protocol
IP   Internet Protocol
UDP  User Datagram Protocol
DNS  Domain Name System
ACK  Acknowledgment
CWND Congestion Window
ISO  International Standard Organization
OSI  Open System Interconnection
MTU  Maximum Transfer Unit
SS   Slow Start
ISPs Internet Service Providers
MSS  Maximum Segment Size
DUPACKs Duplicate Acknowledges
RTO  Retransmission Time Outs
RTT  Round Trip Time
CWND Congestion Window
SSTHRESH Slow Start Threshold
BW   Bandwidth
EWMA Exponential Weighted Moving Average
WWW  World Wide Web
FTP  File Transfer Protocol
VOD  Video-on-Demand
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMD</td>
<td>Additive Increase Multiplicative Decrease</td>
</tr>
<tr>
<td>CA</td>
<td>Congestion Avoidance</td>
</tr>
<tr>
<td>AWND</td>
<td>Receiver’s Advertised Window</td>
</tr>
<tr>
<td>BER</td>
<td>Bit Error Rate</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>FRet</td>
<td>Fast Retransmit</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>Sack</td>
<td>Selective Acknowledgment</td>
</tr>
<tr>
<td>RFC</td>
<td>Request For Comments</td>
</tr>
<tr>
<td>TCPW</td>
<td>TCP Westwood</td>
</tr>
<tr>
<td>BWE</td>
<td>Bandwidth Estimation</td>
</tr>
<tr>
<td>Kb/s</td>
<td>Kilo-bits per second</td>
</tr>
<tr>
<td>NS-2</td>
<td>Network Simulator Version 2</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>TCL</td>
<td>Tool Command Language</td>
</tr>
<tr>
<td>OTCL</td>
<td>Object Oriented Tool Command Language</td>
</tr>
<tr>
<td>GT-ITM</td>
<td>Georgia Tech Internetwork Topology Models</td>
</tr>
<tr>
<td>AT&amp;T</td>
<td>Institute</td>
</tr>
<tr>
<td>TIL</td>
<td>Time Interval length</td>
</tr>
<tr>
<td>NT1</td>
<td>Network Topology 1</td>
</tr>
<tr>
<td>NT2</td>
<td>Network Topology 2</td>
</tr>
</tbody>
</table>
Chapter One

Introduction

Nowadays, with the increasing popularity of the Internet, users demand massive bandwidth [1] to transmit increasingly more data and to provide advanced services with high quality. The Internet has dramatically changed our daily lives in many aspects. We exchange emails to communicate with our friends and colleagues, read the latest news, and search useful information on the Internet. Furthermore, we can make telephone calls; watch TV programs or movies, pay the bills through the Internet. The Internet brings people, who are geographically apart, closer and enhances our lives.

The Internet is a collection of thousands of networks connected by a common set of protocols which enable communication or/and allow the use of the services located on any of the other networks. The common set of protocols running on these networks is referred to as the TCP/IP protocol suite. TCP and UDP (among others) both implement the transport layer. UDP, the user datagram protocol is a simple method used to send datagram’s to another host. It is stateless and does not ensure any reliability or ordering. Applications using UDP [2] cannot assume each packet arrives without duplication or at all and cannot assume the order the packets will arrive in. For this reason, it is commonly used in time sensitive applications where retransmission of lost data is useless and in short request/response
Chapter 1

Introduction

applications such as DNS and broadcast/multicast applications where there may be many receivers unknown to the sender.

TCP, by contrast, is designed for reliable, ordered data transmission. TCP [3] is responsible for breaking up data into pieces (segments) which are suitable for IP, reassembling the stream in order at the receiver, retransmission of lost segments and flow control. TCP is used by many applications, In order to achieve this; the receiver advertises an amount of data which it is currently willing to accept (known as the “advertised” or “receive” window). The sender then sends no more data than the receive window allows, breaking it up into suitably sized segments. The sender places a sequence number in the header of each segment to tell the receiver at what position this packet fits in the data stream. The receiver can then reassemble the data stream on arrival using the sequence numbers and send back short acknowledgement (ACK) packets containing the sequence number of the last byte of continuous data received. This ACK packet allows the sender to verify receipt of data and the receiver to revise the receive window. If a segment arrives out of order (i.e. before a previous one in the sequence), the receiver responds by sending back a duplicate ACK, repeating the sequence number up to which it has full receipt. If the sender sees three such duplicate ACKs or a segment goes unacknowledged for a substantial time, a loss is detected and the lost packet is retransmitted.

In order that TCP can be bi-directional, every packet can simultaneously carry data being sent and an acknowledgement of data received. In the late 1980s the internet experienced a number of “congestion collapses”, where the amount of data being sent began to exceed the available bandwidth of the core internet backbone. This resulted in widespread packet loss, in response to which the senders resented, further exacerbating the issue. The only guide the sender had to control flow was the receive window and the speed of its local network interface, neither of which bore any relation to the available
bandwidth over the link as a whole. As a result, it sent data in bursts as the receive window changed. In response to this, Jacobson proposed a series of measures which the sender would implement, known collectively as “Congestion Avoidance” for TCP[4]. The main modification was to introduce a new limit to control the rate of data sending called Cwnd, the congestion window. The purpose of the congestion window is to restrict the amount of data sent so as to avoid congestion collapse. The amount of unacknowledged data is then constrained above both by the receiver window (which prevents overloading the receiver) and by the congestion window (which prevents overloading the network).

The algorithm used to calculate the congestion window [5] was very important. It was recognised that packet loss was usually a signal that the network was congested and therefore when a loss was detected, rather than simply retransmitting, the sending rate (i.e. the congestion window) should be lowered to relieve the congestion. It was also recognised that the rate should be increased in a controlled manner. A principle of ‘packet conservation’ was employed in that a new packet would only be sent onto the network in response to the return of an existing packet.

1.1 Open System Interconnection Reference Model

The International Standard Organization (ISO) proposed the Open System Interconnection (OSI) reference model [6], shown in Figure 1.1, for the standardization of computer network protocols. The OSI reference model is composed of seven layers, each specifying particular network functions, and provides a conceptual framework for communication between computers.

Actual communication is achieved by using communication protocols; a formal set of rules and conventions that govern how computer exchange information over network medium. One OSI layer communicates with another layer to make use of the services provided by that layer. The services
provided by adjacent layers help a given OSI layer to communicate with its peer layer in other computer systems. The OSI protocols have not become as prevalent as one may expect, given the degree to which OSI has been predicted as the basis for networking. The protocol suite that has attained a stronger foothold is TCP/IP.

The services provided by adjacent layers help a given OSI layer to communicate with its peer layer in other computer systems. The OSI protocols have not become as prevalent as one may expect, given the degree to which OSI has been predicted as the basis for networking. The protocol suite that has attained a stronger foothold is TCP/IP.

![Figure 1.1: The OSI reference model.](image)

### 1.2 Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP) [6] is a reliable, connection-oriented, end-to-end, error free in-order protocol. A TCP connection is a virtual circuit between two computers, conceptually very much like a telephone connection but with reliable data transmission between them. A sending host divides the
data stream into segments. Each segment is labelled with an explicit sequence number to guarantee ordering and reliability. When a host receives in sequence the segments, it sends a cumulative acknowledgment (ACK) in return, notifying the sender that all of the data preceding that segment’s sequence number has been received. If an out-of-sequence segment is received, the receiver sends an acknowledgement indicating the sequence number of the segment that was expected. If outstanding data is not acknowledged for a period of time, the sender will timeout and retransmit the unacknowledged segments. For more details, see the next chapter.

1.3 Congestion Problem

Congestion [7] occurs when the demand is greater than the available resources, such as bandwidths of links, buffer space and processing capacity at the intermediate nodes such as routers. Congestion control is concerned with allocating the resources in a network such that network can operate at an acceptable performance level when the demand exceeds the capacity of the network resources. Careful design is required to provide good service under heavy load. Otherwise, there can be a congestion collapse that is highly resource wasteful and causes undesirable state of operation. Congestion collapse was first observed during the early growth phase of the Internet in the mid 1980s. It was mainly due to TCP connections unnecessarily retransmitting packets that were either in transit or had already been received at the receiver. The original TCP implementations used window-based flow control to control the use of buffer space at the receiver and Go-Back-N retransmission after a packet drop for reliable delivery, but did not include dynamic adjustment of the flow-control window in response to congestion. Different types of congestion collapse are categorized in: a) classical congestion collapse, which occurs when the network is flooded with unnecessary retransmitted packets and was fixed with modern TCP retransmit
timer and congestion control algorithm, b) fragmentation-based congestion collapse, was fixed with Maximum Transfer Unit (MTU) discovery, and c) congestion collapse from undelivered packets, which occurs when networks overloaded with packets that are discarded before they reach the receiver.

The popularity of the Internet has caused a proliferation in the number of TCP implementations. Some of these may fail due to logic errors, or misinterpretations of the specification. Others may deliberately be implemented with the congestion control algorithms that use the available resources more aggressive than other TCP implementations. The consequence of such applications may lead to a state where effectively no congestion control and the Internet are chronically congested. There is also a significant number of TCP non-compatible and non-responsive bandwidth hungry traffic flows in the Internet, which can also pose significant threats to the stability of the Internet. The development of TCP must avoid making radical changes that may stress the deployed network into congestion collapse, and also must avoid a congestion control arms race among competing protocols. TCP has experienced number of changes in its primitive design, during its development process, over the last three decades. The exponential growth in the Internet usage increased the congestion problems. Consequently, many versions of TCP exist today. Presently, all major types of TCP employs congestion control algorithms, which include slow-start (SS), congestion avoidance and fast retransmit and fast recovery.

1.4 Congestion Control

Congestion control [8] concerns in controlling the network traffic in a telecommunications network, to prevent the congestive collapse by trying to avoid the unfair allocation of any of the processing or capabilities of the networks and making the proper resource reducing steps by reducing the rate of packets sent.
Goals that are taken for the evaluation process of a congestion control algorithm are:

I. To accomplish a high bandwidth utilization.

II. To congregate to fairness quickly and efficiently.

III. To reduce the amplitude of oscillations.

IV. To sustain a high responsiveness.

V. To coexist fairly and be compatible with long established widely used protocols.

1.5 Survey of TCP Congestion Control

There has been some serious discussion given to the potential of a large-scale Internet collapse due to network overload or congestion. So far the Internet has survived, but there have been a number of incidents throughout the years where serious problems have disabled large parts of the network. Some of these incidents were a result of algorithms used or not used in the Transmission Control Protocol (TCP). Others were a result of problems in areas such as security, or perhaps more accurately, the lack thereof. The popularity of the Internet has heightened the need for more bandwidth throughout all tiers of the network. Home users need more bandwidth than the traditional 64Kb/s channel a telephone provider typically allows. Video, music, games, file sharing and browsing the web requires more and more bandwidth to avoid the “World Wide Wait” as it has come to be known by those with slower and often heavily congested connections. Internet Service Providers (ISPs) who provide the access to the average home customer had to keep up as more and more users get connected to the information superhighway. Core backbone providers had to ramp up their infrastructure to support the increasing demand from their customers below. Today it would be unusual to find someone in the world that has not heard of the Internet, let alone experiencing it in one form or another. The Internet has become the
fastest growing technology of all time. So far, the Internet is still chugging along, but a good question to ask is “Will it continue to do so?” Although this paper does not attempt to answer that question, it can help us to understand why it will or why it might not. Good and bad network performance is largely dependent on the effective implementation of network protocols. TCP, easily the most widely used protocol in the transport layer on the Internet (e.g. HTTP, TELNET, and SMTP), plays an integral role in determining overall network performance. Amazingly, TCP has changed very little since its initial design in the early 1980’s. A few “tweaks” and “knobs” have been added, but for the most part, the protocol has withstood the test of time. However, there are still a number of performance problems on the Internet and fine tuning TCP software continues to be an area of work for a number of people. Over the past few years, researchers have spent a great deal of effort exploring alternative and additional mechanisms for TCP and related technologies in lieu of potential network overload problems. Some techniques have been implemented; others left behind and still others remain on the drawing board. We’ll begin our examination of TCP by trying to understand the underlying design concepts that have made it so successful. This paper does not cover the basics of the TCP protocol itself, but rather the underlying designs and techniques as they apply to problems of network overload and congestion.

1.6 Research Objectives

The objective of this thesis is to improve the performance of the TCP congestion control so as to avoid the network congestion, to achieve this objective the current end–to-end congestion control mechanisms TCP Tahoe, Reno, Newreno, Sack, Vegas and Westwood are first studied and compared with each other, the purpose of this study is to show the performance of the existing mechanisms and to proof the best of them, then an adaptive dynamic algorithm called Westwood new is developed, the proposed algorithm is
based on an enhancement of the current TCP Westwood algorithm, the additional enhancements are developed based on weakness on the behavior of west wood algorithm.

1.7 Thesis Organization

The remainder of this thesis is organized as follows: An overview of Transmission Control Protocol (TCP) is shown in the chapter 2. TCP congestion control Algorithms is explained in chapter 3. TCP congestion control mechanisms are explained in chapter 4. A theoretical background and a brief description of TCP Westwood congestion control mechanism is described in chapter 5. The brief description of our proposed mechanisms that explains the idea of enhancing TCP Westwood congestion control and the implementation of the enhanced algorithm is discussed in chapter 6. The simulation results and discussions are shown in chapter 7, in which the performance of the original mechanism and the proposed mechanism are discussed and evaluated based on simulation. Finally, the conclusion and discussion is presented in chapter 8.
Chapter Two

Transmission Control Protocol (TCP)

TCP [9] The Transmission Control Protocol is the most used transport protocol in the Internet, is an end-to-end, point-to-point transport protocol used in the Internet. The point-to-point protocol means that there is always a single sender and a single receiver for a TCP session. Being an end-to-end protocol, on the other hand, means that TCP session should cover all parameters and transportations involved from the source host to the destination host. TCP provides applications with reliable byte-oriented delivery of data on the top of the Internet Protocol (IP). TCP sends user data in segments not exceeding the Maximum Segment Size (MSS) of the connection. Each byte of the data is assigned a unique sequence number. The receiver sends an acknowledgment (ACK) upon reception of a segment. TCP acknowledgments are cumulative; the sender has no information whether some of the data beyond the acknowledged byte has been received. TCP has an important property of self-clocking; in the equilibrium condition each arriving ACK triggers a transmission of a new segment. Data are not always delivered to TCP in a continuous way; the network can lose, duplicate or re-order packets. Arrived bytes that do not begin at the number of the next unacknowledged byte are called out-of-order data. As a response to out-of-order segments, TCP sends duplicate acknowledgments (DUPACK) that curry the same acknowledgment number as the previous ACK. In
combination with a retransmission timeout (RTO) on the sender side, ACKs provide reliable data delivery [10]. The retransmission timer is set up based on the smoothed round trip time (RTT) and its variation. RTO is backed off exponentially at each unsuccessful retransmit of the segment [11]. When RTO expires, data transmission is controlled by the slow start algorithm described below. To prevent a fast sender from overflowing a slow receiver, TCP implements the flow control based on a sliding window [12]. When the total size of outstanding segments, segments in flight (Flight Size), exceeds the window advertised by the receiver, further transmission of new segments is blocked until ACK that opens the window arrives.

Early in its evolution, TCP was enhanced by congestion control mechanisms to protect the network against the incoming traffic that exceeds its capacity. A TCP connection starts with a slow-start phase by sending out the initial window number of segments. The current congestion control standard allows the initial window of one or two segments [13]. During the slow start, the transmission rate is increased exponentially. The purpose of the slow start algorithm is to get the “ACK clock” running and to determine the available capacity in the network. A congestion window (cwnd) is a current estimation of the available capacity in the network. At any point of time, the sender is allowed to have no more segments outstanding than the minimum of the advertised and congestion windows. Upon reception of an acknowledgment, the congestion window is increased by one, thus the sender is allowed to transmit the number of acknowledged segments plus one. This roughly doubles the congestion window per RTT. The slow start ends when a segment loss is detected or when the congestion window reaches the slow-start threshold (ssthresh). When the slow start threshold is exceeded, the sender is in the congestion avoidance phase and increases the congestion window roughly by one segment per RTT. When a segment loss is detected, it is taken as a sign of congestion and the load on the network is decreased.
Chapter 2

TCP

The slow start threshold is set to the half of the current congestion window. After a retransmission timeout, the congestion window is set to one segment and the sender proceeds with the slow start.

TCP recovery was enhanced by the fast retransmit and fast recovery algorithms to avoid waiting for a retransmit timeout every time a segment is lost [14]. Recall that DUPACKs are sent as a response to out-of-order segments. Because the network may re-order or duplicate packets, reception of a single DUPACK is not sufficient to conclude a segment loss. A threshold of three DUPACKs was chosen as a compromise between the danger of spurious loss detection and a timely loss recovery. Upon the reception of three DUPACKs, the fast retransmit algorithm is triggered. The DUPACKed segment is considered lost and is retransmitted. At the same time congestion control measures are taken; the congestion window is halved. The fast recovery algorithm controls the transmission of new data until a non-duplicate ACK is received. The fast recovery algorithm treats each additional arriving DUPACK as an indication that a segment has left the network. This allows inflating the congestion window temporarily by one MSS per each DUPACK. When the congestion window is inflated enough, each arriving DUPACK triggers a transmission of a new segment, thus the ACK clock is preserved. When a non-duplicate ACK arrives, the fast recovery is completed and the congestion window is deflated.

We, in turn, discuss the meaning for each of these descriptive terms:

- Connection - Oriented
- Reliability
- Byte Stream Delivery

2.1 Connection - Oriented

Before any data transfer could be started, a connection must be established through a process called three-way handshake. During this process, the TCP
sender and receiver come to an agreement in the establishment of a connection and set the relevant parameters such as Maximum Segment Size (MSS) [15]. For example, if a client computer is contacting a server to send it some information, a TCP connection is established by exchanging control messages as follows in Figure 2.1:

- The client sends a packet with the SYN bit set and a sequence number $x$.
- The server then sends a packet with an ACK number of $x+1$, the SYN bit set and a sequence number $y$.
- The client sends a packet with an ACK number $y+1$ and the connection is established.

![Figure 2.1: TCP connection establishment.](image)

Such signalling period before the exchange of data could sometimes put an unacceptable delay in the applications that are sensitive to delay such as real-time voice.

### 2.2 Reliability

A number of mechanisms, namely checksums, duplicate data detection, sequencing, retransmissions and timers, help TCP to provide reliable data
delivery. All TCP segments carry a checksum, which is used by the receiver to detect corrupted data. TCP keeps track of bytes received in order to detect and drop duplicate transmissions. In packet switched network [16], packets can arrive out of sequence. TCP delivers the byte stream data to an application in order by properly sequencing segments it receives. Corrupted or lost data must be retransmitted in order to guarantee delivery of data. The use of positive acknowledgements by the receiver to the sender confirms successful reception of data. The lack of positive acknowledgements, coupled with a timeout period, calls for a retransmission. TCP maintains a collection of static and dynamic timers on data sent. The TCP sender waits for the receiver to reply with an acknowledgement within a bounded length of time. If the timer expires before receiving any acknowledgement, the sender can retransmit the segment.

2.3 Byte Stream Delivery
TCP interfaces between the application layer above and the network layer below. A stream of 8-bit bytes is exchanged across the TCP connection between the two applications. An application sends data to TCP in 8-bit byte streams, which is then broken by TCP sender into segments in order to transmit data in manageable pieces to the receiver. The size of the application layer payload is variable but may not be larger than MSS, which is usually announced by the TCP receiver during connection establishment using the MSS option in the TCP header. However, it is limited by the outbound link’s Maximum Transfer Unit (MTU) [17]. Alternatively, the sender may use the path MTU discovery to derive an appropriate MSS.

2.4 TCP Segment Format
The TCP segment [18] consists of a TCP header followed by a payload. The payload includes information data passed from the application layer above for
transmission. The TCP header includes address information for the segment and all information required for implementation of algorithms used in TCP. An option field is included in the TCP header that can include specific information for a particular TCP connection. The default TCP header size is 20 bytes. However, this may go up to 60 bytes with inclusion of an option field. To this effect, a header length filed is also included in the TCP header as shown in Figure 2.2.

<table>
<thead>
<tr>
<th>Source port number (16-bits)</th>
<th>Destination port number (16-bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>4-bit reserved (8 bits)</td>
<td>U A P R S F R C S S Y I G K H T N N</td>
</tr>
<tr>
<td>Receiver window size (16-bits)</td>
<td></td>
</tr>
<tr>
<td>Checksum (16-bits)</td>
<td>Urgent Data (16-bits)</td>
</tr>
<tr>
<td>Options</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2: TCP segment header format

TCP segments are sent as IP datagram. The IP header carries several fields, including the source and destination host addresses. A TCP header follows the IP header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP.

2.5 TCP Flow Control

TCP flow control [19] is provided through the well-known sliding window mechanism [20]. ACKs sent by the TCP receiver carry the advertised window, which limits the number of bytes the TCP sender may have outstanding at any time. The advertised window corresponds to the size of TCP receiver’s receive socket buffer. The key feature of the sliding window protocol is that it permits pipelined communication to better utilize the channel capacity. The sender can send a maximum W frames without
acknowledgement, where \( W \) is the window size of the sliding window. The sliding window maps to the frames in sender’s buffer that are to be sent, or have been sent and now are waiting for acknowledgement. For maximum throughput, the amount of data in transit at any given time should be the channel bandwidth-delay product, which refers to the product of a data link’s capacity (in bits per second) and its end-to-end delay (in seconds).

![Diagram of window flow control 'self-clocking']

Figure 2.3: Window flow control 'self-clocking'

End-to-end protocols that implement sliding window flow control, like TCP, share an important self-clocking property. A schematic representation of a sender and receiver on high bandwidth networks connected by a slow link, the bottleneck link, is shown in Figure 2.3 [4]. The vertical and horizontal dimensions are the bandwidth (BW) and time respectively. Each of the shaded boxes represents a packet. The area of each box is the packet size because ‘BW-delay product = bits’. The number of bits in a packet does not change as it goes through the network so a packet on the slow link has to spread out more in time.

Figure 2.3 shows the ideal case in which a single sender fully utilizes the non-shared bottleneck link, the slowest link in the path, with a fixed bandwidth and always sends fixed size segments. In this case, the ACK inter-
arrival time \((As)\) at the sender is constant and equal to the packet transmission delay over the bottleneck link, \(P_B\). This constant stream of returning ACKs is referred to as ACK clock. The arrival of an ACK moves the sliding window to the right by one segment and clocks out a new segment, thereby keeping the number of outstanding packets, i.e. the window constant.

### 2.6 TCP Time-Out Mechanism

In order to avoid long delays when there is no response from the receiver in a TCP connection, a time-out mechanism [21] is employed. Therefore, after each TCP segment transmission by a sender, a timer is set and it starts counting down. If the TCP sender does not receive a threshold number of ACKs before the timer expires, it assumes that either the packet or the ACK is lost, and retransmits the same packet again until an ACK is received. The TCP retransmit timeout (RTO) value must be carefully chosen. If RTO value is too small, the time expires quickly and premature time-outs will be generated during the usual TCP operation and thus unnecessary retransmission will occur. On the other hand, if RTO value is too large, the TCP will slowly respond to the segment loss, which means longer end-to-end delay and can also degrade performance. Therefore, the RTO value must be optimized to the extent possible.

When a packet is sent over a TCP connection, the sender times how long it takes for it to be acknowledged, producing a sequence of round-trip samples. Older TCP implementations only time one segment per RTT [22], whereas newer implementations use the timestamp option to time every segment. Timing every segment allows much closer tracking of changes in RTT. We refer to the RTT sampling rate as the number of RTT samples the TCP sender captures per RTT divided by the TCP sender’s load. In case the TCP sender times every segment and the TCP receiver acknowledges every segment, the
RTT sampling rate is 2. If the TCP sender times every segment and the TCP receiver acknowledges every other segment (delayed-ACK), the RTT sampling rate is 0.5. The closer the sampling rate to 1 the more accurately the TCP sender measures the RTT. TCP uses a mechanism to estimate the round-trip time (RTT) in the network, based on which the timer can be set accordingly. This will be done continually so that a variable estimation will happen. TCP collects information on the most recent RTTs and then makes an average value, called a sample RTT [23]. The \textit{EstimatedRTT} is then computed in an iterative manner by using the following equation 2.1:

\begin{equation}
\text{EstimatedRTT} = (1-O) \text{Estimated RTT} + O \text{ Sample RTT} \tag{2.1}
\end{equation}

Where O is a constant between 0 and 1 that control how rapidly the estimated RTT adapts to changes and the typical value for O is 0.125 [4], which decide trade-off between efficiency and fairness. This method is called Exponential Weighted Moving Average (EWMA) owing to the inclusion of the factor O. The method provides that the influence of given sample decreases exponentially fast and puts more weight on the recent sample instead. In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT. In [24] the author defines the RTT variations, \textit{DevRTT}, as an estimate of how much Sample RTT typically deviates from \textit{EstimatedRTT} as the following equation 2.2.

\begin{equation}
\text{DevRTT} = (1 - R) \text{DevRTT} + R |\text{SampleRTT} - \text{Estimated RTT}| \tag{2.2}
\end{equation}

Note that \textit{DevRTT} is a EWMA of the difference between \textit{SampleRTT} and \textit{EstimatedRTT}. If the \textit{SampleRTT} values have little fluctuation, then DevRTT will be small; on the other hand, if there is a lot of fluctuation, DevRTT will be large. The recommended value of R is 0.25 [4]. After computing EstimatedRTT, the TCP RTO interval is set to that value plus a safety margin in order to avoid
any unnecessary retransmissions and large data transfer delay as the following equation 2.3:

\[ RTO = \text{Estimated RTT} + 4 \text{DevRTT} \]  

(2.3)

### 2.7 TCP Congestion Control in the Internet

With the fast development of the network, more and more networks access the Internet. The Internet has been expanded in terms of its scale, coverage and users quantities. More and more users use the Internet as their data transmission platform to implement various applications.

Apart from traditional applications of World Wide Web (WWW), e-mail and File Transfer Protocol (FTP), network users try to expand some new applications, such as tele-education, video telephone, video conference and Video-on-Demand (VoD), on the Internet. A best-effort network like the Internet does not have the notion of admission control or resource reservation to control the imposed network load, i.e., the total number of packets that can reside within the network. A best-effort network under high network load is called congested.

If the network becomes congested [25], no one can use the network resources at all and also the fact that when the network is congested, any additional transmitted packets would be lost because of lack of network resources such as the buffer spaces at the routers. So, network endpoints sharing a best-effort network need to respond to congestion by implementing congestion control in order to avoid further packet drop. Otherwise, it may cause the following negative effects:

- Increase the delay and jitter of packet transmission
- Packet retransmission caused by high delay
- Decrease the network throughput and lower the utilization of network resources
- Intensified congestion can occupy too many network resources and the irrational.
Assignment of resources even can lead to congestion collapse: the network load stays extremely high but throughput is reduced to close to zero. The main objective of TCP’s congestion control is to limit the sending rate to avoid overwhelming the network when it faces congestion on the path to the destination. Let us first examine how a TCP sender limits the rate at which it sends traffic into its connection. Each side of a TCP connection consists of a receive buffer, a send buffer and several variables. The TCP congestion control has each side of a connection keep track of an additional variable, the congestion window (CWND). The CWND size imposes a constraint on the rate a TCP sender can send traffic into the network. TCP controls the rate of transmission of the packets as well as the congestion occurrence in the network. Therefore, the throughput of the TCP becomes a function of the size of the congestion window (cwnd) and the RTT. If the throughput is measured in bytes per second, then with MSS bytes in each segment, the TCP throughput will be expressed as in Equation 2.4.

\[
\text{TCP Throughput} = \frac{(cwnd \times \text{MSS})}{\text{RTT}}
\]  

(2.4)

Let us next consider how a TCP sender perceives that there is congestion on the path between itself and the destination. A loss event at a TCP sender is defined as the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver. When there is an excessive congestion, one (or more) router buffers along the path overflows, causing a datagram to be dropped. The dropped datagram, in turn, results in a loss event at the sender, either by a timeout or the receipt of three duplicate ACKs, which is taken by the sender to be an indication of congestion on the sender-to-receiver path. Notice that TCP congestion control algorithm does not require any support of routers for their functioning. TCP congestion algorithm has three major components: additive increase and multiplicative decrease, slow-start and reaction to timeout events.
2.8 Additive - Increase, Multiplicative - Decrease

A TCP sender additively increases its rate when it perceives that the end-to-end path is congestion free, and multiplicatively decreases its rate when it detects (via a loss event) that the path is congested. For this reason, TCP congestion control is often referred to as an additive increase, multiplicative-decrease (AIMD) [26]. The rationale for an increase in rate when it perceives no congestion is that if there is no detected congestion, then there is likely to be available bandwidth that could be additionally used by TCP connection. In such circumstances, TCP increases its CWND slowly, cautiously probing for additional available bandwidth in the end-to-end path: it does increment its CWND a little each time it receives an ACK, with the goal of increasing CWND by 1 MSS every RTT [27] as in Equation 2.5.

\[
\text{CWND} = \text{CWND} + \text{MSS} \left(\frac{\text{MSS}}{\text{CWND}}\right)
\]  

(2.5)

![Figure 2.4 Additive-increase, multiplicative-decrease congestion control.](image)

The linear increase phase of TCP’s congestion control protocol is known as congestion avoidance (CA). The value of CWND repeatedly goes through
cycles during which it increases linearly and then suddenly drops to half its current value (multiplicative-decrease) when a loss event occurs, giving rise to a saw-toothed pattern in long-lived TCP connections, as shown in Figure 2.4.

2.9 Reaction to Timeout Events

If an ACK for a given segment is not received in a certain amount of time, known as RTO value, a timeout event occurs and the segment is resent [6]. After a timeout event, a TCP sender enters a slow-start phase; it sets the CWND to 1 MSS and then grows the congestion window exponentially until CWND reaches SSTHRESH. When CWND reaches SSTHRESH, TCP enters the CA phase, during which CWND ramps up linearly. Assuming initial value of CWND equals 1MSS, initial value of SSTHRESH is large, i.e. 64 Kbytes, and TCP sender begins in slow start state, a visual description of slow start and congestion avoidance [28] followed by a timeout is shown in Figure 2.5.

![Figure 2.5 TCP Congestion Control.](image)

The TCP is based on the notion of the sliding window in order to guarantee reliable and in order packet delivery. All major types of TCP employs
congestion control algorithms. However, the implementation of the fast recovery and fast retransmit mechanism is quite different.
Chapter Three

TCP Congestion Control Algorithms

The basis of TCP congestion control lies in additive increase multiplicative decrease (AIMD). Where TCP sender half the congestion window for every window containing a packet loss, and increasing the congestion window by roughly one segment per round trip time (RTT) otherwise. Let AIMD (A, B) congestion control refer to pure AIMD congestion control that uses an increase parameter (A) and a decrease parameter (B). That is, after a loss event the congestion window is decreased from \( W \) to \((1-B) \times W\) packets, and otherwise the congestion window is increased from \( W \) to \((W+A)\) packets each round-trip time. Currently, TCP uses AIMD (1, 1/2) congestion control [30].

TCP deploys the window based additive increase multiplicative decrease AIMD (1, 1/2) algorithm, which relies on the timely feedback of acknowledgment (ACK) to control the progress of information flow. It is well known that as channel bandwidth, link delay, and bit error rate (BER) increase, TCP shows inefficiency and instability due to its window based congestion control algorithms.

In detail, besides the receiver’s advertised window, \( awnd \), TCP’s congestion control introduced two new variables for the connection: the congestion window, \( cwnd \), and the slow start threshold, \( ssthresh \). The window size of the sender, \( w \), was defined to equal: \( \text{Window Size} = \text{Min}(cwnd, awnd) \), instead
The congestion window \( awnd \) is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver. The former is based on the sender’s assessment of perceived network congestion, and the latter is related to the amount of available buffer space at the receiver for this connection. The congestion window can be thought of as being a counterpart to advertised window. Whereas receiver advertised window is used to prevent the sender from overrunning the resources of the receiver, the purpose of congestion window is to prevent the sender from sending more data than the network can accommodate in the current load conditions.

As a congestion control mechanism, TCP sender dynamically increasing or decreasing its window size according to the degree of network congestion. The idea of congestion control is thus to modify the congestion window adaptively to reflect the current load of the network. In practice, this is done through detection of lost packets which can basically be detected either via a time-out mechanism or via duplicate acknowledge (DUPACK) [32]. Modern implementations of TCP contain four algorithms as basic Internet standard:

- Slow Start
- Congestion Avoidance
- Fast Retransmit
- Fast Recovery

Understanding these basic TCP congestion control algorithms is very helpful in modifying, enhancing or developing several congestion control mechanisms.

### 3.1 Slow Start Algorithm

When TCP finished the three-way handshake [33] it bursts out as many packets allowed by the agreed window size, \( wnd \). This was not a large problem in the small networking, but as the networks grew, and amount of
connected hosts increased, these large bursts turned out to be a cause of problems. Congestion started to occur in network bottlenecks, data is added up faster than it could be forwarded or received. Therefore an algorithm to prevent immediate bursts was introduced. With the incorporation of slow start (SS) [34] two new variables were introduced: the slow start threshold (ssthresh) and the congestion window (cwnd). When starting a transmission cwnd is set to 1 MSS and ssthresh is set to an arbitrary size depending on the OS used. The amount of data the sender is allowed to send is determined by min [cwnd, wnd] and since cwnd = 1 at start-up only one packet is allowed. cwnd will then increase by 1 MSS for every ACK received (every RTT). This exponential growth will continue until loss detection or cwnd = sshtresh, when this is happens the congestion avoidance algorithm will take over. Slow Start algorithm is shown as follows:

<table>
<thead>
<tr>
<th>Slowstart algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize: cwnd = 1</td>
</tr>
</tbody>
</table>
| for (each segment ACKed) cwnd++;
| until (congestion event or cwnd > ssthresh) |

### 3.2 Congestion Avoidance Algorithm

To avoid congestion on the network the exponential increase of cwnd must be halted. This is usually not a problem in small localized LANs where the usual limitation is the window size. However, in large WANs there are many more hosts that are supposed to share the network capacity and if all hosts would run at full capacity then congestion is hard to avoid.

Congestion Avoidance (CA) [35] handles this by lowering the cwnd increase to only 1 packet per RTT, giving cwnd a lower and linear growth. If the Retransmit Time Out (RTO) occurs, CA will consider this as a loss of packet. CA will then set ssthresh to half the current cwnd and after this reset cwnd to one and initiate a SS. Congestion avoidance algorithm is shown as follow:
Figure 3.1: TCP slow start and congestion avoidance behaviour in action.

Figure 3.1 shows the congestion window (cwnd) versus round trip time, for the TCP slow start and congestion avoidance algorithms behaviour in action.

### 3.3 Fast Retransmit Algorithm

Fast retransmit (FRet) [36] is a short simple algorithm, treating three received DUPACKs as a sign of loss. It is unlikely that the missing packet has gone so far astray from the others that three later packets would arrive before the lost one finds its way to the receiver. FRet was created to remove the need to wait for an RTO by quickly retransmitting the lost packet after three DUPACKs, preventing unnecessary long downtime in the transmission. After the packet
has been retransmitted $FRet$ sets $ssthresh=1/2 \times cwnd$ and enters slow start.

Fast Retransmit algorithm is shown as follows:

<table>
<thead>
<tr>
<th>Fast retransmit algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>If receiving 3DUPACK or RTO</td>
</tr>
<tr>
<td>Retransmit the packet</td>
</tr>
<tr>
<td>$ssthresh = cwnd/2$</td>
</tr>
<tr>
<td>$cwnd = 1$</td>
</tr>
<tr>
<td>perform slowstart</td>
</tr>
</tbody>
</table>

Figure 3.2 shows the Fast retransmit algorithm.
3.4 Fast Recovery Algorithm

Fast recovery algorithm [37] immediately is after Fast Retransmit, after fast retransmit sends what appears to be the missing segment, congestion avoidance, but not slow start is performed. This is the fast recovery algorithm. It is an improvement that allows high throughput under moderate congestion, especially for large windows. The reason for not performing slow start in this case is that the receipt of the duplicate ACKs tells TCP more than just a packet has been lost. Since the receiver can only generate the duplicate ACK when another segment is received, that segment has left the network and is in the receiver's buffer. That is, there is still data flowing between the two ends, and TCP does not want to reduce the flow abruptly by going into slow start. The fast retransmit and fast recovery algorithms are usually implemented together as follows:

<table>
<thead>
<tr>
<th>Fast retransmit algorithm (Reno)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If receiving 3DUPACK or RTO</td>
</tr>
<tr>
<td>Retransmit the packet</td>
</tr>
<tr>
<td>After retransmission do not enter slowstart</td>
</tr>
<tr>
<td>Enter fast recovery</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fast recovery algorithm (Reno)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set ssthresh = cwnd/2 ;</td>
</tr>
<tr>
<td>cwnd = ssthresh + 3 ;</td>
</tr>
<tr>
<td>/* the three extra packets to compensate for the three packets leaving the network causing the DUPACKs */</td>
</tr>
<tr>
<td>Each duplicate ACK received</td>
</tr>
<tr>
<td>cwnd+ + ;</td>
</tr>
<tr>
<td>/* to compensate for the one leaving the network */</td>
</tr>
<tr>
<td>transmit new packet if allowed</td>
</tr>
<tr>
<td>If new ACK</td>
</tr>
<tr>
<td>cwnd= ssthresh ;</td>
</tr>
<tr>
<td>return to congestion avoidance</td>
</tr>
</tbody>
</table>

Fast Recovery works according to the following steps:

1) When the third duplicate ACK in a row is received, set ssthresh to one-half the current congestion window, cwnd, but not less than two segments. Retransmit the missing segment. Set cwnd to ssthresh plus 3
times the segment size. This inflates the congestion window by the number of segments that have left the network and which the other end has cached.

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

3) When the next ACK arrives that acknowledges new data, set \( cwnd \) to \( ssthresh \) (the value set in step 1). This ACK should be the acknowledgment of the retransmission from step 1, one round-trip time after the retransmission. Additionally, this ACK should acknowledge all the intermediate segments sent between the lost packet and the receipt of the first duplicate ACK. This step is congestion avoidance, since TCP is down to one-half the rate it was at when the packet was lost, as shown in Figure 3.3

![Figure 3.3 TCP Fast Recovery Algorithm.](image)
3.5 TCP’S Retransmission Timeout Mechanism–Related Work

The area of RTO estimation is an area of TCP that has not received the same level of scrutiny as other TCP flow control mechanisms. Estimating an appropriate value for the RTO is very important. Too small a value may result in needless sender timeout despite the ACK being in transit from receiver to sender. Too large an RTO value could result in significantly reduced overall goodput for a flow. Recent results from a particular large-scale Internet Traffic study by Balakrishnan [38] have shown that approximately 50% of all packet losses require a timeout to recover. Another recent study [39] found that over 85% of all timeouts are due to non-trigger of the fast retransmit mechanism. While there are some proposals that seek to reduce TCP’s reliance on timeout expiry, these will take time to be discussed, agreed-upon and possibly eventually deployed on a wide-enough scale. In the mean time, there is a clear need for renewed research into the TCP RTO mechanisms.

TCP currently estimates the RTO once per RTT and does not use update its RTO calculation for retransmitted packets – Karn’s algorithm [40]. The current scheme for TCP’s RTO estimation was first presented in [41] and is captured in [42]:

\[
\begin{align*}
\text{srttnew} &= (1-a) \text{srttold} + a \cdot \text{rttcurrent} \\
\text{mdevnew} &= (1- b) \text{mdevold} + b \cdot (\text{rttcurrent} - \text{srttnew}) \\
\text{RTO} &= \text{srttnew} + 4 \text{mdevnew}
\end{align*}
\]

The scheme combines a smoothed weighted average estimate of the RTT (srtt) and a mean deviation of the RTT (mdev) to obtain the timeout estimate (RTO). RTT samples are obtained by TCP sender implementations by correlating transmitted packets with returned acknowledgements. Karen’s algorithm ensures RTT sanity by requiring the sender to collect RTT samples only for original packets and not for retransmitted packets.
In [43], Allman and Paxson undertake a detailed investigation into varied parameter values for a and b. They use a passive analysis method to study Internet traffic from 1995. They conclude that the currently used values of a=0.125 and b=0.25 are reasonably good and the accuracy of the RTO mechanism is “virtually unaffected by the settings of the parameters in the exponentially weighted moving average (EWMA) estimators commonly used”. The authors also conclude that the minimum RTO value used plays an important role in the performance of the flow while the timer granularity has less impact and the RTT sampling rate hardly affects RTO estimation. Their study makes no comment on the role of the initial RTO value selected.

Ludwig and Slower [44] through analytical modeling and experimentation with long-lived flows in a controlled network, provide results that cause them to disagree with the conclusion of Paxson and Allman. Specifically, the authors claim that as the RTT sampling rate and sender load increase, the choice of parameters for the EWMA estimators is essential to improved RTO estimation. Further, the authors argue that timer granularity needs to be on the order of common RTT found on the Internet today – this is clearly not the case with the 500ms timer as used in many operating systems today. The authors propose the Eifel retransmission timer as an improvement over the current TCP RTO mechanism.

More recently, Allman, Griner and Richard [45] investigate the performance of the retransmission timer over links with varying propagation delay. The authors perform a simulation study with single long-lived flows and no competing traffic. Their conclusion is that the current RTO mechanism sufficiently tracks the RTT of a flow even in the presence of drastic RTT fluctuations. Further, they show that a fine-grained 1- millisecond RTO timer results in reduced flow throughput when compared against a 500- millisecond timer. The authors do not reproduce the above results with short flows or more heavily loaded network links.
In 1998 simulation-based studies, Aron and Druschel [46] showed examples of a TCP flow with an RTT of 60-milliseconds and an estimated RTO value of 1.6 seconds. They conclude that the coarse-grained TCP timers are the key reason for such over-prediction and suggest that for the case where the RTT is significantly less than 500- milliseconds, the RTO be fixed to the minimum value of 2 ticks (500 milliseconds to 1 second). In addition they suggest that fine-grained timers (1 millisecond) be used to estimate the RTT while the coarse-grained timers be used to schedule the RTO.

### 3.6 Initial RTO Value – Examination of Related Work

RFC 1122 [8] states that the initial RTO value for TCP should be set to 3 seconds. However, the author of this work has not been able to find any references that investigate or justify selection of that particular value. Further, as subsequent paragraphs will indicate, most TCP implementations did not adhere to this specification for the initial RTO value.

Selection of an appropriate initial RTO value is important. Too large an initial RTO will be problematic for flows with small RTT as it may result in performance degradation in the case of early packet drop. Too small an initial RTO will be problematic for flows with large RTT as it may cause needless retransmissions due to bad timeouts. RFC 2525 [47] documents some examples where low initial RTO values caused needless performance degradation via early retransmissions.

In Comer and Lin’s [48] 1994 study of 5 operating systems, none of the surveyed TCP implementations conformed to the 3-second initial RTO. The initial RTO values ranged from 200-milliseconds to 1.5 seconds.

In a subsequent study published in 1997, Dawson [49] actively probed the behaviour of 6 commercial TCP implementations and found 5 of the implementations had an initial RTO of 1 second while one of the implementations had an initial RTO of 300- milliseconds. Jens Vockler
performed detailed tests [50] to determine the initial RTO in December of 1998. He experiments with various user settable parameters for a machine running Solaris to understand the relationship between the parameters and the initial timeout value that is seen.
Chapter Four

TCP Congestion Control Mechanisms

Congestion control [29] defines the methods for implicitly interpreting signals from the network in order for a sender to adjust its rate of transmission. In this chapter, we are precisely concerned with the end-to-end congestion control mechanisms widely used in TCP implementations today. A detailed description of the standard algorithms including slow start, congestion avoidance, fast retransmit and fast recovery are first given. Then the current end-to-end congestion control mechanisms, their benefits and weaknesses are described. These mechanisms include TCP Tahoe, Reno, New Reno, SACK and Vegas.

A TCP receiver uses cumulative acknowledgements to specify the sequence number of the next packet the receiver expects. The generation of acknowledgements allows the sender to get continuous feedback from the receiver. Every time a sender sends a segment, the sender starts a timer and waits for the acknowledgement. If the timer expires before the acknowledgement is received, TCP assumes that the segment is lost and retransmits it. This expiration of the timer is referred to as a timeout. If the acknowledgement is received, however, TCP records the time at which the segment was received and calculates the Round Trip Time (RTT). A weighted moving average of the RTT is maintained and used to calculate the timeout value for each segment.
TCP congestion control Mechanisms

TCP uses a sliding window mechanism to achieve flow control that allows multiple packets to be present in flight so that the available bandwidth can be used more efficiently. This keeps the sender from overwhelming the receiver’s buffers. However, the most important variation of TCP’s sliding window mechanism over other sliding window mechanisms is the variation of the window size in TCP with respect to time. If the receiver is unable to send acknowledgements at the rate at which the sender is sending data, the sender reduces its sending window. The sender and receiver agree upon the number of packets that a sender can send without being acknowledged, and upon number of packets the receiver is able to receive, before its buffers become overwhelmed. This is accomplished by the Advertised Window (AWND) parameter, which is the receiver side estimate of the number of packets it is able to receive without overflowing its buffer queues.

TCP also includes several variables for performing congestion control. The CWND variable defines the number of consecutive packets that a sender is able to send before receiving an acknowledgement and the variable is changed based on network conditions. At any given point in time the sender is allowed to send as many consecutive packets as provided by the minimum of CWND and AWND, thereby considering the condition of the receiver and the network simultaneously. At the connection start up time, in order that effective communication take place between the sender and the receiver, TCP uses error, flow and congestion control algorithms.

The TCP protocol basics are specified in RFC 793 [6]. In order to avoid the network congestion that became a serious problem as the number of network hosts increased dramatically, the basic algorithms for performing congestion control were given by Jacobson [4]. Later, the congestion control algorithms have been included in the standards track TCP specification by the IETF [51].
The TCP sender uses a *congestion window* \((cwnd)\) in regulating its transmission rate based on the feedback it gets from the network. The congestion window is the TCP sender’s estimate of how much data can be outstanding in the network without packets being lost. After initializing \(cwnd\) to one or two segments, the TCP sender is allowed to increase the congestion window either according to a *slow start* algorithm, that is, by one segment for each incoming acknowledgement (ACK), or according to *congestion avoidance*, at a rate of one segment in a round-trip time. The *slow start threshold* \((ssthresh)\) is used to determine whether to use slow start or congestion avoidance algorithm. The TCP sender starts with the slow start algorithm and moves to congestion avoidance when \(cwnd\) reaches the \(ssthresh\). The TCP sender detects packet losses from incoming duplicate acknowledgements, which are generated by the receiver when it receives out-of-order segments. After three successive duplicate ACKs, the sender retransmits a segment and sets \(ssthresh\) to half of the amount of currently outstanding data. \(cwnd\) is set to the value of \(ssthresh\) plus three segments, accounting for the segments that have already left the network according to the arrived duplicate ACKs. In effect the sender halves its transmission rate from what it was before the loss event. This is done because the packet loss is taken as an indication of congestion, and the sender needs to reduce its transmission rate to alleviate the network congestion.

The retransmission due to incoming duplicate ACKs is called *fast retransmit*. After fast retransmit the TCP sender follows the *fast recovery* algorithm until all segments in the last window have been acknowledged. During fast recovery the TCP sender maintains the number of outstanding segments by sending a new segment for each incoming acknowledgement, if the congestion window allows. The TCP congestion control specification temporarily increases the congestion window for each incoming duplicate ACK to allow forward transmission of a segment, and deflates it back to the
value at the beginning of the fast recovery when the fast recovery is over. Two variants of the fast recovery algorithm have been suggested by the IETF. The standard variant exits the fast recovery when the first acknowledgement advancing the window arrives at the sender. However, if there is more than one segment dropped in the same window, the standard fast retransmit does not perform efficiently. In 1988, Van Jacobson published the paper [4] that has become the standard for TCP congestion control algorithms that are shown in chapter 3.

After that many enhancements have been made to TCP congestion control by many researches. This was leading to a set of End-to-End mechanisms. Over the years, many mechanisms have been widely acknowledged for maintaining stability of the Internet. This section presents a brief description to the different TCP variants of end-to-end congestion control [52] including: Tahoe, Reno, NewReno, Sack, Vegas and Westwood.

4.1 TCP Tahoe

Early TCP implementations followed a go-back-n technique using cumulative positive acknowledgement, and required a retransmit timer expiration to resend data lost during the flight. These TCPs did very little to handle congestion. TCP Tahoe [53] added a number of new algorithms and refinements to earlier implementations. The new algorithms include slow-start, congestion avoidance, and fast-retransmit. One of the major refinements was the modification of the roundtrip time estimator used to set retransmission timeout values. Initially, it was assumed that lost packets represented congestion. Therefore, it was assumed by Jacobson that when a packet loss occurred, the sender should lower its share of the bandwidth. Tahoe does not deal well with multiple packet drops within a single window of data. The two phases in increasing the congestion window, the slow- start and the congestion avoidance phases can be summed up with the following
equations, where \( ssthresh \) is the threshold value at which TCP changes its phase from slow-start to congestion avoidance.

**Slow-start phase:**

\[
\begin{align*}
\text{cwnd} &= \text{cwnd} + 1 \\
&\text{if } \text{cwnd} < \text{ssthresh}
\end{align*}
\]

**Congestion avoidance phase:**

\[
\begin{align*}
\text{cwnd} &= \text{cwnd} + 1/\text{cwnd} \\
&\text{if } \text{cwnd} \geq \text{ssthresh}
\end{align*}
\]

When a segment loss is detected, the cwnd and ssthresh are updated as follows.

\[
\begin{align*}
\text{ssthresh} &= \text{cwnd}/2 \\
\text{cwnd} &= 1
\end{align*}
\]

During the time when TCP Tahoe came up, the network environment and the applications that were being used did not demand high bandwidth links. Hence, this variant of TCP did not have to face the challenge of scaling to the high bandwidth delay product network. TCP Tahoe has major drawbacks as a mean of providing data services over a multimedia network, since random loss resulting from fluctuations in real-time traffic can lead to significant throughput deterioration in the high bandwidth delay product network. The results of these studies conclude that the performance is degraded when the product of the loss probability and the square of the bandwidth-delay product are large. Also, for the high bandwidth delay product network, TCP is extremely unfair towards connections with higher propagation delays.

### 4.2 TCP Reno

The TCP Reno [54] implementation modified the sender to incorporate a mechanism called fast recovery. Unlike Tahoe, Reno does not empty the pipe unnecessarily on the receipt of a few numbers of dupacks. Instead, with the mechanism of fast recovery the congestion window is set to half its previous value. The idea is that the only way for a loss to be detected via a timeout and
not via the receipt of a dupack is when the flow of packets and ACKs has completely stopped, which would be an indication of heavy congestion. But if the sender is still able to receive an ACK, then it should not fall back into slow-start, as it does in the case of TCP Tahoe. This case does not imply heavy congestion, since the flow still exists, but the sender should send with relatively less vigour, utilizing a lower amount of resources. The mechanism of fast recovery comes into picture at this stage. After receiving a certain number of dupacks, the sender will retransmit the lost packet; but, unlike Tahoe, it will not fall back into slow start. It will rather take advantage of the fact that the currently existing flow should keep on sending, albeit using fewer resources. By using fast recovery the sender uses a congestion window that is half the size of the congestion window present just before the loss. This factor forces Reno to send fewer packets out until it knows that it is feasible to send more. Therefore, it has indeed reduced its utilization of the network. Although Reno TCP is better than Tahoe in cases of single packet loss, Reno TCP is not much better than Tahoe when multiple packets are lost within a window of data. Fast recovery ensures that the pipe does not become empty. Therefore, slow-start is executed only when a packet is timed out. This is implemented by setting sssthresh to half the current congestion window size and then setting the congestion window to 1 segment, causing the TCP connection to slow-start until the sssthresh is reached; then it goes into the congestion avoidance phase like in the case of Tahoe. The Reno TCP represented in equation form looks like this.

Slow-start phase
\[ cwnd = cwnd + 1 \]

When a segment is detected, the fast retransmission algorithm halves the congestion window.
\[ sssthresh = \frac{cwnd}{2} \]
\[ cwnd = sssthresh \]
TCP Reno then enters fast recovery phase. In this phase, the window size is increased by one segment when a duplicate acknowledgement is received; and the congestion window is restored to ssthresh when a non-duplicate acknowledgement corresponding to the retransmitted segments is received.

The basic problem in TCP Reno is that fast retransmit assumes that only one segment was lost. This can result in loss of ACK clocking and timeouts if more than one segment is lost. Reno faces several problems when multiple packet losses occur in a window of data. This usually occurs when fast retransmit and fast recovery is invoked. It is invoked several times in succession leading to multiplicative decreases of cwnd and ssthresh impacting the throughput of the connection. Another problem with Reno TCP is ACK starvation. This occurs due to the ambiguity of duplicate ACKs. The sender reduces the congestion window when it enters fast retransmit; it receives dupacks that inflate the congestion window so that it sends new packets until it fills its sending window. It then receives a non-dupack and exits fast recovery. However, due to multiple losses in the past, the ACK will be followed by 3 dupacks signalling that another segment was lost; this way, fast retransmit is entered again after another reducing of ssthresh and cwnd. This happens several times in succession and during this time the left edge of the sending window advances only after each successive fast retransmit; and the amount of data in flight eventually becomes more than the congestion window. When there are no more ACKs to be received, the sender stalls and recovers from this deadlock only through 15 timeout, which causes slow-start. There are two solutions available for the above problems: Newreno and TCP SACK.

4.3 TCP NewReno

For the typical implementation of the TCP Fast Recovery algorithm described in Reno the TCP data sender only retransmits a packet after a retransmit
timeout has occurred, or after three duplicate acknowledgements have arrived
triggering the Fast Retransmit algorithm. A single retransmit timeout might
result in the retransmission of several data packets, but each invocation of the
Reno Fast Retransmit algorithm leads to the retransmission of only a single
data packet.

Problems can arise, therefore, when multiple packets have been dropped from
a single window of data and the Fast Retransmit and Fast Recovery
algorithms are invoked. In this case, if the SACK option is available, the TCP
sender has the information to make intelligent decisions about which packets
to retransmit and which packets not to retransmit during Fast Recovery.

In the absence of SACK, there is little information available to the TCP
sender in making retransmission decisions during Fast Recovery.

From the three duplicate acknowledgements, the sender infers a packet loss,
and retransmits the indicated packet. After this, the data sender could receive
additional duplicate acknowledgements, as the data receiver acknowledges
additional data packets that were already in flight when the sender entered
Fast Retransmit.

In the case of multiple packets dropped from a single window of data, the
first new information available to the sender comes when the sender receives
an acknowledgement for the retransmitted packet (that is the packet
retransmitted when Fast Retransmit was first entered). If there had been a
single packet drop, then the acknowledgement for this packet will acknowledge all of the packets transmitted before.

Fast Retransmit was entered (in the absence of reordering). However, when
there were multiple packet drops, then the acknowledgement for the
retransmitted packet will acknowledge some but not all of the packets
transmitted before the Fast Retransmit. We call this packet a partial
acknowledgment.
Along with several other suggestions, suggested that during Fast Recovery [55] the TCP data sender respond to a partial acknowledgment by inferring that the indicated packet has been lost, and retransmitting that packet. Fast Recovery algorithm NewReno is a modification to the Fast Recovery algorithm in Reno TCP that incorporates a response to partial acknowledgements received during Fast Recovery.

### 4.3.1 The Fast Retransmit and Fast Recovery Algorithms in NewReno

The standard implementation of the Fast Retransmit and Fast Recovery algorithms is given in [13]. The NewReno modification of these algorithms is given below. A Fast Recovery procedure begins when three duplicate ACKs are received and ends when either a retransmission timeout occurs or an ACK arrives that acknowledges all of the data up to and including the data that was outstanding when the Fast Recovery procedure began.

1. When the third duplicate ACK is received and the sender is not already in the Fast Recovery procedure, set ssthresh to no more than the value given in equation 4.1 below.

\[
\text{ssthresh} = \max (\text{FlightSize} / 2, 2\times\text{MSS})
\]  

where Flight Size is defined as the amount of data that has been sent but not yet acknowledged.

Record the highest sequence number transmitted in the variable "recover".

2. Retransmit the lost segment and set cwnd to ssthresh plus 3*MSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.

3. For each additional duplicate ACK received, increment cwnd by MSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.

4. Transmit a segment, if allowed by the new value of cwnd and the receiver’s advertised window.
5. When an ACK arrives that acknowledges new data, this ACK could be the acknowledgment elicited by the retransmission from step 2, or elicited by a later retransmission.

If this ACK acknowledges all of the data up to and including "recover", then the ACK acknowledges all the intermediate segments sent between the original transmission of the lost segment and the receipt of the third duplicate ACK.

Set cwnd to either (1) min (ssthresh, FlightSize + MSS); or (2) ssthresh, where ssthresh is the value set in step 1; this is termed "deflating" the window. (We note that "FlightSize" in step 1 referred to the amount of data outstanding in step 1, when Fast Recovery was entered, while "FlightSize" in step 5 refers to the amount of data outstanding in step 5, when Fast Recovery is exited.) If the second option is selected, the implementation should take measures to avoid a possible burst of data, in case the amount of data outstanding in the network was much less than the new congestion window allows.

**Exit the Fast Recovery procedure.**

If this ACK does *not* acknowledge all of the data up to and including "recover", and then this is a partial ACK. In this case, retransmit the first unacknowledged segment. Deflate the congestion window by the amount of new data acknowledged, then add back one MSS and send a new segment if permitted by the new value of cwnd. This "partial window deflation" attempts to ensure that, when Fast Recovery eventually ends, approximately ssthresh amount of data will be outstanding in the network.

Do not exit the Fast Recovery procedure (i.e., if any duplicate ACKs subsequently arrive, execute Steps 3 and 4 above).

For the first partial ACK that arrives during Fast Recovery, also reset the retransmit timer.
Note that in Step 5, the congestion window is deflated when a partial acknowledgement is received. The congestion window was likely to have been inflated considerably when the partial acknowledgement was received. In addition, depending on the original pattern of packet losses, the partial acknowledgement might acknowledge nearly a window of data. In this case, if the congestion window was not deflated, the data sender might be able to send nearly a window of data back-to-back.

There are several possible variants to the simple response to partial acknowledgements described above.

There is a related question of how many packets to retransmit after each partial acknowledgement. The algorithm described above retransmits a single packet after each partial acknowledgement. This is the most conservative alternative, in that it is the least likely to result in an unnecessarily-retransmitted packet. A variant that would recover faster from a window with many packet drops would be to effectively Slow-Start, requiring less than $N$ roundtrip times to recover from $N$ losses. With this slightly-more-aggressive response to partial acknowledgements, it would be advantageous to reset the retransmit timer after each retransmission.

Avoiding multiple Fast Retransmits is particularly important if more aggressive responses to partial acknowledgements are implemented, because in this case the sender is more likely to retransmit packets already received by the receiver?

As a final note, we would observe that in the absence of the SACK option, the data sender is working from limited information. One could spend a great deal of time considering exactly which variant of Fast Recovery is optimal for which scenario in this case. When the issue of recovery from multiple dropped packets from a single window of data is of particular importance, the best alternative would be to use the SACK option.
4.3.2 Resetting the Retransmit Timer.

There is a question of when to reset the retransmit timer after a partial acknowledgement, the above algorithm resets the retransmit timer only after the first partial ACK. In this case, if a large number of packets were dropped from a window of data, the TCP data sender’s retransmit timer will ultimately expire, and the TCP data sender will invoke Slow-Start. We call this the impatient variant of NewReno.

In contrast, the NewReno simulations in [56] illustrate the algorithm described above, with the modification that the retransmit timer is reset after each partial acknowledgement. We call this the Slow-but-Steady variant of NewReno. In this case, for a window with a large number of packet drops, the TCP data sender retransmits at most one packet per roundtrip time.

For TCP implementations where the Retransmission Timeout Value (RTO) is generally not much larger than the round-trip time (RTT), the impatient variant can result in a retransmit timeout even in a scenario with a small number of packet drops. For TCP implementations where the Retransmission Timeout Value (RTO) is usually considerably larger than the round-trip time (RTT), the slow but-Steady variant can remain in Fast Recovery for a long time when multiple packets have been dropped from a window of data.

Neither of these variants is optimal; one possibility for a more optimal algorithm might be one that recovered more quickly from multiple packet drops, and combined this with the Slow-but-Steady variant in terms of resetting the retransmit timers. We note, however, that there is a limitation to the potential performance in this case in the absence of the SACK option.

4.3.3 Avoiding Multiple Fast Retransmits

Avoiding multiple Fast Retransmits caused by the retransmission of packets already received by the receiver, in the absence of the SACK option, a duplicate acknowledgement carries no information to identify the data packet or packets at the TCP data receiver that triggered that duplicate
The TCP data sender is unable to distinguish between a duplicate acknowledgement that results from a lost or delayed data packet, and a duplicate acknowledgement that results from the sender’s retransmission of a data packet that had already been received at the TCP data receiver. Because of this, multiple segment losses from a single window of data can sometimes result in unnecessary multiple Fast Retransmits (and multiple reductions of the congestion window).

With the Fast Retransmit and Fast Recovery algorithms in Reno or NewReno TCP, the performance problems caused by multiple Fast Retransmits are relatively minor (compared to the potential problems with Tahoe TCP, which does not implement Fast Recovery). Nevertheless, unnecessary Fast Retransmits can occur with Reno or NewReno TCP, particularly if a Retransmit Timeout occurs during Fast Recovery.

With NewReno, the data sender remains in Fast Recovery until either a Retransmit Timeout, or until all of the data outstanding when Fast Retransmit was entered has been acknowledged. Thus with NewReno, the problem of multiple Fast Retransmits from a single window of data can only occur after a Retransmit Timeout.

The NewReno algorithm eliminates the problem of multiple Fast Retransmits. NewReno uses a new variable "send_high", whose initial value is the initial send sequence number. After each retransmit timeout, the highest sequence numbers transmitted so far is recorded in the variable "send_high". If, after a retransmit timeout, the TCP data sender retransmits three consecutive packets that have already been received by the data receiver, then the TCP data sender will receive three duplicate acknowledgements that do not acknowledge "send_high". In this case, the duplicate acknowledgements are not an indication of a new instance of congestion. They are simply an indication that the sender has unnecessarily retransmitted at least three packets.
Chapter 4  TCP congestion control Mechanisms

We note that if the TCP data sender receives three duplicate acknowledgements that do not acknowledge "send_high", the sender does not know whether these duplicate acknowledgements resulted from a new packet drop or not. For a TCP that implements the bugfix described in this section for avoiding multiple fast retransmits, the sender does not infer a packet drop from duplicate acknowledgements in these circumstances. As always, the retransmit timer is the backup mechanism for inferring packet loss in this case.

When the third duplicate ACK is received and the sender is not already in the Fast Recovery procedure, check to see if those duplicate ACKs cover more than "send_high". If they do, then set ssthresh to no more than the value given in equation 1, record the highest sequence number transmitted in the variable "recover", and go to Step 2. If the duplicate ACKs don’t cover "send_high", then do nothing. That is, do not enter the Fast Retransmit and Fast Recovery procedure, do not change ssthresh, do not go to Step 2 to retransmit the "lost" segment, and do not execute Step 3 upon subsequent duplicate ACKs.

4.3.4 Implementation Issues for the Data Receiver

"Out-of-order data segments should be acknowledged immediately, in order to trigger the fast retransmit algorithm that some data receivers do not send an immediate acknowledgement when they send a partial acknowledgment, but instead wait first for their delayed acknowledgement timer to expire. This severely limits the potential benefit from NewReno by delaying the receipt of the partial acknowledgement at the data sender. Our recommendation is that the data receiver sends an immediate acknowledgement for an out-of-order segment, even when that out-of-order segment fills a hole in the buffer.

4.3.5 Conclusion of NewReno Mechanism

The TCP Newreno [57] modifies the fast retransmit and fast recovery mechanisms of Reno TCP. These modifications are implemented to fix the
drawbacks of TCP Reno. Here, the wait for the retransmit timer is eliminated when multiple packets are lost from a window. Newreno is the same as Reno but applies more intelligence during fast recovery. It utilizes the idea of partial ACKs. When there are multiple packet losses, the ACK for the retransmitted packet will acknowledge some but not all the packets sent before the fast retransmit. In Newreno, a partial ACK is taken as an indication of another lost packet and as such the sender transmits the first unacknowledged packet. Unlike Reno, partial ACKs do not take Newreno out of fast recovery. This way Newreno retransmits 1 packet per RTT until all lost packets are retransmitted, and avoids requiring multiple fast retransmits from a single window of data. This Newreno modification of Reno TCP defines a fast recovery procedure that begins when three duplicate ACKs are received and ends when either a retransmission timeout occurs or an ACK arrives that acknowledges all of the data up to and including the data that was outstanding when the fast recovery procedure began. The Newreno algorithm can be explained in the following steps:

1) On the receipt of the third dupack, if the sender is not already in fast recovery procedure, then set ssthresh to no more than the value below: ssthresh = \text{max}(\text{flightsize}/2, 2\times\text{MSS}). Also, remember the highest sequence number transmitted in a variable.

2) Retransmit the lost packet and set cwnd to ssthresh + 3\times\text{MSS}. This artificially inflates the congestion window by the number of segments that have left the network and that the receiver has buffered.

3) For each additional dupack received, increment the congestion window by MSS.

4) Transmit a segment, if allowed by the new value of cwnd and the receivers advertised window.
5) When an ACK arrives that acknowledges new data, this ACK could be the acknowledgement elicited by the retransmission from step 2, or one elicited by a later retransmission.

### 4.4 TCP Vegas

In 1995, Brakmo, O'Malley and Peterson [58] came with a new TCP implementation called Vegas that achieves between 40% and 70% better throughput and 1/5 to 1/2 the losses when compared with TCP Reno. TCP Vegas [59] also had all the changes and modifications on the sender side. In Reno, the RTT is computed using a coarse-grained timer, which does not give an accurate estimate of RTT. Tests conducted conclude that for losses that resulted in a timeout, it took Reno an average of 1100ms from the time it sent a segment that was lost, until it timed out and resent the segment; whereas less than 300ms would have been the correct timeout interval had a more accurate clock been used.

TCP Vegas fixes this problem using a finer coarse-grained timer. Vegas also changed the retransmission mechanism. The system clock is read and saved each time a segment is sent; when an ACK arrives, the clock is read again and the RTT calculation is computed using this time and the timestamp recorded for the relevant segment. With the use of this accurate RTT, retransmission is decided as follows: When a dupack is received, Vegas checks to see if the new RTT is greater than RTO. If it is, Vegas re-transmits the segment without having to wait for the 3rd dupack. Whereas, when a non-dupack is received, if it is the first or second one after a retransmission, Vegas checks again to see if $\text{RTT} > \text{RTO}$; if so, then the segment is retransmitted. This process catches any other segment that may have been lost previous to the retransmission without requiring a waiting period to receive a dupack. Vegas treats the receipt of certain ACKs as a trigger to check if a timeout should happen, but still contain Reno’s timeout code in
Chapter 4  

TCP congestion control Mechanisms

case this mechanism fails to recognize a lost segment. Vegas' congestion avoidance actions are based on changes in the estimated amount of extra data in the network. Vegas define the RTT of a connection as its BaseRTT when the connection is not congested. In practice, it is the minimum of all measured roundtrip times and mostly it is the RTT of the first segment sent by the connection before the router queues increase. Vegas use this value to calculate the expected throughput. Secondly, it calculates the current actual sending rate. This is done by recording the sending time for a segment, recording how many bytes are transmitted between the time of segment sending and receiving its acknowledgement, computing the RTT for the segment when its acknowledgement arrives, and dividing the number of bytes transmitted by the sample RTT. This calculation is done once per round trip time.

Thirdly, Vegas compares actual to expected throughput and adjusts the window accordingly. Difference between the actual and expected throughput is recorded. Vegas defines two thresholds, α and β, which roughly correspond to having too little and too much extra data in the network, respectively. Following is the mechanism of the congestion control in equation form. Diff is the difference between actual and expected throughput.

\[
\begin{align*}
\text{Diff} < 0 & : \text{change BaseRTT to the latest sampled RTT} \\
\text{Diff} < \alpha & : \text{increase the congestion window linearly} \\
\text{Diff} > \beta & : \text{decrease the congestion window linearly} \\
\alpha < \text{Diff} < \beta & : \text{do nothing}
\end{align*}
\]

To be able to detect and avoid congestion during slow-start, Vegas allows exponential growth only every other RTT. In between, the congestion window stays fixed so a valid comparison of the expected and actual rates can be made. When the actual rate falls below the expected rate by the
equivalent of one router buffer, Vegas changes from slow start mode to linear increase/decrease mode.

A couple of problems with TCP Vegas that could have a serious impact on its performance are the issues of rerouting and stability. Rerouting a path may change the propagation delay of the connection; Vegas uses the connection to adjust the window size and it can affect the throughput considerably. Another issue of TCP Vegas is its stability.

Since each TCP connection attempts to keep a few packets in the network when their estimation of the propagation delay is off, this could lead the connection to inadvertently keep many more packets in the network causing a persistent congestion.

Research on TCP Vegas to date consists primarily of analyses of the protocol, improving its congestion avoidance and detection techniques. Most of the studies involving TCP Vegas consist of its performance evaluation with respect TCP Reno.

4.5 TCP SACK

TCP throughput can be affected considerably by multiple packets lost from a window of data. TCP’s cumulative acknowledgement scheme causes the sender to either wait for a round trip time to find out about a lost packet, or to unnecessarily retransmit segments that have been correctly received. With this type of scheme, multiple dropped segments generally cause TCP to lose its ACK-based clock, which reduces the overall throughput. Selective Acknowledgement (SACK) [60] is a strategy that rectifies this behaviour. With selective acknowledgement, the data receiver can inform the sender about all segments that have arrived successfully, so that the sender need retransmit only those segments that have actually been lost. This mechanism uses two TCP options: the first is an enabling option, ‘SACK-permitted’ which can be sent in a SYN segment to indicate that the SACK option can be
used once the connection is established; the second is the SACK option itself, which may be sent once permission has been given by SACK20 permitted. In other words, a selective acknowledgement (SACK) mechanism combined with a selective repeat retransmission policy can help to overcome these limitations. The receiving TCP sends back SACK packets to the sender TCP indicating to the sender data that has been received. The sender can then retransmit only the missing segments.

The congestion control algorithms present in the standard TCP implementations must be preserved. In particular, to preserve robustness in the presence of packets reordered by the network, recovery is not triggered by single ACK reporting out-of-order packets at the receiver. Further, during the recovery, the data sender limits the number of segments sent in response to each ACK. Existing implementations limit the data sender to sending one segment during Reno-style fast recovery, or two segments during slow-start. Other aspects of congestion control, such as reducing the congestion window in response to congestion, must similarly be preserved. The use of time-outs as a fallback mechanism for detecting dropped packets is unchanged by the SACK option. Because the data receiver is allowed to discard SACKed data, when a retransmit timeout occurs the data sender must ignore prior SACK information, when determining which data to retransmit. Studies regarding TCP SACK include issues concerning aggressiveness of the protocol in the presence of congestion in comparison to other TCP implementations. TCP SACK has also been used to enhance performance of TCP in satellite environments.

The SACK option format as defined in RFC 2018 [61] is as shown in Figure 4.1.
3.3.5.1 Limitations of SACK

In [62], Floyd addressed various issues related to the behavior and performance of TCP SACK. One important limitation mentioned by the author is that TCP SACK requires 64 bits (8 bytes) to represent the upper and lower bound sequence numbers of every block it selectively acknowledges. Since TCP protocol limits the maximum length of the options field to 40 bytes and SACK is usually implemented along with TCP Timestamp options, an acknowledgment packet can carry a maximum of only 3 blocks' information.

In the extension proposed to SACK in RFC 2883 [63], a new mechanism called DSACK was proposed wherein; the first block of SACK is used to carry information about the latest duplicate packet received. This further reduces the amount of actual SACK information that can be carried in the ACK packets. Similarly, if other TCP options are enabled, this maximum number would decrease further. This is a very likely scenario as more and more options to TCP are being put forward and accepted, particularly in the wireless networks environment. This rather small maximum number of SACK block information can lead to efficiency problems in the performance of TCP.
4.6 TCP Westwood

TCP Westwood [64] is yet another improvement in the TCP Reno family line. The Fast Recovery algorithm from TCP New Reno has been modified. To help gain faster recovery bandwidth estimation (BWE) algorithm also has been added. This BWE function is what makes TCP Westwood standout. Influenced by TCP Vegas, BWE uses the RTT and the amount of data that has been sent during this interval to calculate an estimate of the currently successful transfer rate. The bandwidth estimate is then used when a loss is detected, setting cwnd and ssthresh at values near the estimation. The main purpose behind this is to improve the throughput in wireless links, where loss is more often caused by link failure than by congestion. There is also the general benefit that starting CA at higher values will lower the recovery time on most networks, thus lowering the transfer times, chapter 4 describes TCP Westwood in details.

4.7 Discussion

The Transmission Control Protocol TCP was standardized in 1981 with the publication of RFC 793[6]. After only a short period it was evident that it had some flaws in its behaviour and a new version named Tahoe was released.

Figure 4.2 indicates the TCP Inherence start by TCP Tahoe which added a Slow Start (SS) function, which started the transmission of data slowly but exponentially. An algorithm named Congestion Avoidance (CA) was also added, designed to slow the growth of the senders output lowering the possibility of causing congestion. The final algorithm added in the TCP Tahoe version is called Fast Retransmit (FRet).
Fast Retransmit resend the first unacknowledged packet in the send buffer after receiving three DUPACKs after each other instead of waiting for a RTO. TCP SACK is a feature of Selective Acknowledgement, telling the sender what packets have been successfully received at the receiver and not just that a packet has been lost. TCP SACK works exceptionally well, compared to ordinary TCP clones, on a network with problems with multiple packet losses. This help keeping the retransmission queue small and saves time waiting not needing to wait for the next ACK to see if something else is missing. TCP SACK can be used with many later versions of TCP. TCP Reno is an upgrade of TCP Tahoe. Adding an algorithm Fast Recovery ($FRec$), designed to help TCP recover faster to maximum output after suffering a packet loss. Fast Recovery keeps the flow going instead of performing a SS. TCP New Reno was an improvement of the cooperation between FRet and FRec. To improve the behaviour, when facing with rapid multiple packet losses on connections that cannot use the TCP SACK feature. Still keeping the flow going when receiving DUPACKs, TCP New Reno is less careful on how to update the cwnd and ssthresh variables, usually ending
up giving them higher values, than its predecessor. TCP Vegas is more of a spinoff TCP clone than part of the evolution of TCP Tahoe.

<table>
<thead>
<tr>
<th>Commentary</th>
<th>TCP variants</th>
<th>TCP Sender Congestion-control Action</th>
<th>Event</th>
<th>State or Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resulting in a doubling of Cwnd every RTT</td>
<td>Tahoe Reno NewReno Westwood Vegas</td>
<td>Cwnd = Cwnd + 1 if (cwnd &gt; ssthresh) set state to &quot;Congestion Avoidance&quot;</td>
<td>ACK receipt for previously unacknowledged data</td>
<td>Slow Start (SS)</td>
</tr>
<tr>
<td>Additive increase, resulting in increasing of Cwnd by 1 MSS every RTT</td>
<td>Tahoe Reno NewReno Westwood Vegas</td>
<td>cwnd = cwnd + 1/cwnd</td>
<td>ACK receipt for previously unACK data</td>
<td>Congestion Avoidance (CA)</td>
</tr>
<tr>
<td>Fast recovery, implementing multiplicative decrease. cwnd will not drop below 1 MSS.</td>
<td>Tahoe Reno (partial ack) NewReno (continue with partial ACK until full ack) Vegas Westwood</td>
<td>ssthresh = Cwnd/2, Cwnd = Ssthresh, set state to &quot;Congestion Avoidance&quot;</td>
<td>Loss event detected by 3 DUP ACK</td>
<td>SS or CA</td>
</tr>
<tr>
<td>Enter slow start.</td>
<td>Tahoe Reno NewReno Vegas Westwood</td>
<td>sssthresh = cwnd/2, Cwnd = 1 set state to &quot;Slow Start&quot;</td>
<td>Timeout</td>
<td>SS or CA</td>
</tr>
<tr>
<td>cwnd and ssthresh not changed</td>
<td>Tahoe Reno NewReno Westwood Vegas</td>
<td>Increment duplicate ACK count for segment being acknowledged</td>
<td>Duplicate ACK</td>
<td>SS or CA</td>
</tr>
</tbody>
</table>

Table 4.1: TCP Congestion Control Algorithms.

Using a time based estimate of the capacity and limiting the output to avoid congestion, TCP Vegas is a smooth and intelligent TCP clone. However, it does not work well with the TCP Reno family, due to the more aggressive nature of those TCP versions. On its own or together with other TCP Vegas
instances it is impressively fair in its sharing and smooth in its throughput. TCP Westwood uses an advanced bandwidth estimation (BWE) to try and figure out the capacity of the network and uses this knowledge to lower the loss in throughput caused by packet loss. This BWE takes the sender output as a measure of the bandwidth of the network and sets the ssthresh accordingly when suffering a loss. Table 4.1 summarizes the different TCP variants. The table shows how slow start, congestion avoidance and fast recovery differ, as well as the ACK format required. Westwood TCP is friendly towards New Reno TCP and improves fairness in bandwidth allocation whereas Vegas TCP is fair but it is not able to grab its bandwidth share when coexisting with Reno or in the presence of reverse traffic because of its RTT-based congestion detection mechanism.
Chapter Five

TCP Westwood Overview

TCP Westwood [65] is a simple modification of the TCP source protocol stack, which allows the source to estimate the available bandwidth, and to use the bandwidth estimation to recover faster, thus achieving higher throughput. TCP Westwood exploits two basic concepts: the end-to-end estimation of the available bandwidth, and the use of such estimate to set the slow start threshold and the congestion window. It is worth underscoring that TCP Westwood (TCPW) does not require any intervention from network layer or proxy agents.

TCP Westwood (TCPW) source continuously estimates the packet rate of the connection by properly averaging the rate of returning ACKs. The estimate is used to compute the “permissible” congestion window and slow start threshold to be used after a congestion episode is detected, that is, after three duplicate acknowledgments or after a timeout. The rationale of this strategy is simple: in contrast with TCP Reno, which simply halves the congestion window after three duplicate ACKs, TCP Westwood (TCPW) attempts to make a more “informed” decision. It selects a slow start threshold and a congestion window that they are consistent with the effective connection rate at the time congestion is experienced. We call such mechanism faster recovery. The “key innovation” of TCP Westwood (TCPW) is to use the bandwidth estimate “directly” to drive the window, instead of using it to
compute the backlog. The rationale is that if a connection is currently achieving a given rate, then it can safely use the window corresponding to that rate without causing congestion in the network. TCP Westwood (TCPW) offers a number of features that are not available in TCP Reno and SACK. For example, the knowledge of the available bandwidth can be used to adjust the rate of a variable rate source (assuming such source is controlled by TCP). Like TCP Reno, TCP Westwood (TCPW) cannot distinguish between buffer overflow loss and random loss. However, in presence of random loss, TCP Reno overreacts and reduces the window by half. TCP Westwood (TCPW), on the other hand, after packet loss and retransmission timeout, resumes with the previous window as long as the bottleneck is not yet saturated (i.e., no buffer overflow). To prevent the unnecessary window reduction of TCP Reno in case of random packet loss, and more precisely the loss caused by wireless links.

**4.1 Necessity of Bandwidth and its Management**

The bandwidth [1] simply represents the capacity of the communication media to transfer data from source to destination. Wider the route/path for data transmission, more packets of information will be transmitted to the user's Internet enabled devices. Bandwidth is a gross measurement, taking the total amount of data transferred in a given period of time at a particular rate, without taking into consideration the quality of the signal itself [66]. Furthermore, the bandwidth is responsible for data transfer speed and commonly used in Internet connections. Bigger the bandwidth quota is, the higher the connection speed and hence quicker it will be to upload and download information. The basic measurement unit of bandwidth is bits per second (bps) and it can be kilobits per second (kbps), megabits per second (mbps) and gigabits per second (gbps). Various Internet connections are offering different bandwidth standards. For instance, the traditional dial-up
Internet Connection provides a very narrow bandwidth limit of about 56 kbps, while the current broadband connections allow data transfer at much higher speed ranging from 128 kbps to 2mbps. Bandwidth is both absolutely and relatively much more expensive for any institution. Many institutions are finding that they still do not have reliable, usable Internet Access for their students and staff despite considerable investment. Improving the performance of the information delivery chain is urgent if researchers and students are to be benefited from the Internet and take part in the international academic community. The performance of the existing Internet Connection can be enhanced by monitoring and controlling mechanism and this is known as bandwidth management. The bandwidth management means to improve the performance of an Internet Connection by removing unnecessary traffic. Bandwidth is like a pipe and if the flow of the material inside the pipe is not monitored and managed properly then it will clog up with unwanted traffic. Similar is the case for computer network bandwidth where it can be hijacked by viruses, spam, peer-to-peer file-sharing traffic, etc. Furthermore, the useful resource of any organisation will be eaten by unproductive applications and may be difficult to avail useful services by the needy ones.

Bandwidth management is a process of controlling and measuring communication (traffic, or packets) on a network link in order to avoid filling up the link either to its capacity or crossing the capacity which could result in network congestion and poor performance. If the university has a much slower connection, Internet Access will still function, however if the connection is increased and the management removed, useful access to the Internet will decrease immediately and soon become impossible.

A bandwidth management functions by sorting outbound network traffic into various classes according to service and application types. Traffic is then planned out accordingly to the minimum and maximum bandwidth that is
configured for each of the traffic types. Bandwidth management requires three activities:

i. Policy,
ii. Monitoring, and
iii. Implementation.

If any one of these activities is missing then the management of bandwidth is significantly compromised. These activities inform and reinforce each other. The Figure 5.1 shows the relationship among bandwidth management activities.

Figure 5.1: Bandwidth Management Activities.

- Monitoring is important for defining and enforcing policy. Network monitoring informs the process of creating an enforceable policy that reflects the actual needs of the user group. It is also the necessary part of enforcing policy. Furthermore, monitoring is also required to diagnose faults and troubleshooting of the network.
- Without an “Acceptable Internet Usage Policy” no amount of bandwidth is enough to satisfy the demands of an unrestricted user community. Individuals downloading music and other files for their personal use can absorb an institution’s bandwidth. Frequently it is the minority that consumes the majority bandwidth. In this situation, user education is far more productive than technical solutions. The institution’s policy needs to be
understood and enforced. It becomes the responsibility of the network administrators to find out which users are not adhering to the policy and to interact with them on a face-to-face level.

- There are number of tools and techniques that help network administrators to ensure that bandwidth is being managed properly and policy is adhered.

The key components are:

i. Network Analyzers—for monitoring traffic
ii. Firewalls—for blocking malicious and unwanted traffic;
iii. Anti-Virus—for protecting network;
iv. Caches—for efficiently using bandwidth,
v. Traffic Shapers—for prioritising and controlling traffic
vi. Quota Systems—for managing user behaviour.

The above discussion has made it clear that network management is the core concern but bandwidth being a limited and valuable asset attracts immediate attention of the network administrators and researchers to do efforts so that adequate network services may be provided to the genuine users.

As it is an expensive resource, organizations should ensure that they are purchasing the right amount of bandwidth, and test whether they are getting what they pay for. A university should also aim to use the available bandwidth as best it can, not only by limiting abuse but also by providing access to as many people as possible in order to maximize the use of the resource.

For example, if there are periods (such as during weekends) when there is low usage of bandwidth, a university could consider providing free or paid access to the local community. Unused bandwidth is wasted money.

**How much bandwidth is needed?**

Institutions will need to increase their bandwidth from time to time. Typical reasons include:
- Student numbers tend to grow, and universities increase the number of computers they own.
- The volume of resources on the Internet keeps growing, and tends to become ever more bandwidth-hungry.

New services on the Internet, such as streaming media, may present new opportunities for education, though it is also possible that streaming media will prove to be useful for entertainment only.

The price of bandwidth is falling, even in developing countries where access is only possible via satellite, so increasing bandwidth is more achievable. However, few institutions undertake studies on how much bandwidth is needed. Normally, an institution simply gets as much as it can afford, or increases its bandwidth because a new cheaper rate enables them to do so, or because Internet access simply becomes too slow. There is nothing wrong with this approach, but clear thinking on the subject enables authorities to decide whether they need more bandwidth or whether they need more control over usage (or both). What follows are some pointers from Hughes Network Systems, the MIMCOM network, and Moratuwa University.

A graph of bandwidth required (and bandwidth required per user) against number of users can be expected to look something like this in figure 5.2:

![Figure 5.2 the Bandwidth](image)

Two reasons why bandwidth per user reduces (but overall bandwidth demand increases) are as follows:
• The higher the available bandwidth, the less is needed per user because requests can be satisfied faster, leaving the ‘big pipe to the Internet’ open for the next user. Also, the waste of bandwidth due to retransmissions is reduced.

• Not all connected users use bandwidth all the time, because in between requesting Web pages they are also reading them.

4.2 Overview

In TCP Westwood (TCPW) the sender continuously computes the connection bandwidth estimate (BWE) that are defined as the share of bottleneck bandwidth used by the connection. Thus, bandwidth estimate (BWE) is equal to the rate at which data is delivered to the TCP receiver. The estimate is based on the rate at which ACKs are received and on their payload. After a packet loss indication, (i.e., reception of 3 duplicate ACKs, or timeout expiration), the sender resets the congestion window and the slow start threshold based on bandwidth estimate (BWE). To understand the rationale of TCP Westwood (TCPW), note that bandwidth estimate (BWE) varies from flow to flow sharing the same bottleneck; it corresponds to the rate actually achieved by each individual flow. Thus, it is an achievable rate by definition. Consequently, the collection of all the bandwidth estimate (BWE) rates, as estimated by the connections sharing the same bottleneck, is an achievable set. When the bottleneck becomes saturated and packets are dropped, TCP Westwood (TCPW) selects a set of congestion windows that correspond exactly to the measured bandwidth estimate (BWE) rates and thus reproduce the current individual throughputs. The solution is feasible, but it is not guaranteed to be “fair share”. Another important element of this procedure is the Round-Trip Time (RTT) estimation. Round-Trip Time (RTT) is required to compute the window that supports the estimated rate bandwidth estimate (BWE). Ideally, the Round-Trip Time (RTT) should be measured when the
bottleneck is empty. In practice, it is set to equal to the overall minimum Round-Trip Time delay (\(RTT_{min}\)) measured so far on that connection (based on continuous monitoring of ACK RTTs).

**Setting \(cwnd\) and \(ssthresh\) in TCP Westwood:**

Further details regarding bandwidth estimation are provided in following sections. For now, let us assume that a sender has determined the connection bandwidth estimate (BWE), BWE is used to properly set \(cwnd\) and \(ssthresh\) after a packet loss indication. First, we note that in TCP Westwood (TCPW), congestion window increments during slow start and congestion avoidance remain the same as in Reno, which are exponential and linear, respectively. A packet loss is indicated by (a): the reception of 3 DUPACKs, or (b): coarse timeout expiration. In case the loss indication is 3 DUPACKs, TCP Westwood (TCPW) sets \(cwnd\) and \(ssthresh\) as follows:

**After 3 DUPACKS**

\[
\text{If receiving 3 DUPACKS} \\
\text{Set } ssthresh = (BWE \times RTT_{min}) / \text{seg}_\text{size}; \\
\text{and if } cwnd > ssthresh; \\
\text{then set } cwnd = ssthresh; \\
\text{enter congestion avoidance}
\]

In the pseudo-code, \(\text{seg}_\text{size}\) identifies the length of a TCP segment in bits. In case a packet loss is indicated by timeout expiration, \(cwnd\) and \(ssthresh\) are set as follows:

**After Timeout**

\[
\text{If } RTO \text{ then set } ssthresh = (BWE \times RTT_{min}) / \text{seg}_\text{size}; \\
\text{if } (ssthresh < 2) ssthresh = 2; \\
\text{end if}; \\
cwin = 1; \\
\text{end if} \\
\text{enter slow start}
\]

The rationale of the algorithm above is that after a timeout, \(cwnd\) and the \(ssthresh\) are set equal to 1 and bandwidth estimate (BWE), respectively. Thus, the basic TCP Reno behaviour is still captured, while a speedy recovery is ensured by setting \(ssthresh\) to the value of bandwidth estimate (BWE).
4.3 Bandwidth Estimation

The TCPW sender uses ACKs to estimate BWE. More precisely, the sender uses the following information: (1) the ACK arrival times and, (2) the increment of data delivered to the destination. Let’s assume that an ACK is received at the source at time $t_k$, notifying that $d_k$ bytes have been received at the TCP receiver as shown in Figure 4.3. We can measure the sample bandwidth used by that connection as $b_k = d_k / (t_k - t_{k-1})$, where $t_{k-1}$ is the time the previous ACK was received. Letting $\Delta t_k = t_k - t_{k-1}$, then $b_k = d_k / \Delta t_k$. Since congestion occurs whenever the low-frequency input traffic rate exceeds the link capacity, we employ a low-pass filter to average sampled measurements and to obtain the low-frequency components of the available bandwidth.

![Figure 4.3 Bandwidth samples](image)

Let $b_k$ be the bandwidth sample. Let $\alpha_k$ be the time varying exponential filter coefficient at time $t_k$, The TCPW filter is then given by

<table>
<thead>
<tr>
<th>Bandwidth estimation (BWE) algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$BWE = b_k = \alpha_k b_{k-1} + (1 - \alpha_k) [(b_k + b_{k-1}) / 2]$</td>
</tr>
</tbody>
</table>

where

- $b_k =$ sample bandwidth $= d_k / (t_k - t_{k-1})$
- $d_k =$ amount of bytes acknowledged by ACK $k$,
- $t_k =$ arrival time of ACK $k$, and $\alpha_k = \frac{2\tau - \Delta(t_k - t_{k-1})}{2\tau + \Delta(t_k - t_{k-1})}$

where $\tau$ is the cut-off frequency of this Tustin filter

Notice the coefficients $\alpha_k$ depend on $t_k$ to properly reflect the variable inter-arrival times. A number of considerations must be taken into account while interpreting the information that a returning ACK carries regarding delivery of segments to the destination. Two of these considerations are:
1. An ACK received by the source implies that a transmitted packet was delivered to the destination. A DUPACK also implies that a transmitted packet was delivered, triggering the transmission by the receiver of the DUPACK. Thus, DUPACKs are considered in estimating bandwidth.

2. TCP ACKs can be “delayed,” i.e., receivers wait for a second packet before sending an ACK. These items are included in the implementation of TCP Westwood (TCPW) under Linux.
Chapter Six

Our Proposed Mechanism (TCP WestwoodNew)

With increases in the heterogeneity and the complexity of the Internet, many problems have emerged in TCP congestion control mechanism. The primary reasons for these problems are that the congestion signals are only indicated by packet loss and that TCP uses fixed Additive-Increase-Multiplicative-Decrease (AIMD) parameter values to increase and decrease window size, whereas the window size should be changed according to the network environment. The AIMD mechanism is triggered by the detection of packet losses in the network. The congestion control mechanism improves the throughput by adjusting the increasing and decreasing parameters statically and/or dynamically. Because window size indicates the maximum amount of packets that TCP can transmit for one Round Trip Time (RTT), an adequate window size for a TCP connection is equal to the product of the available bandwidth and the round-trip propagation delay between the sender and receiver hosts. TCP measures the RTTs of the network path between sender and receiver hosts by checking the departure times of the data packets and the arrival times of the corresponding ACK packets. However, TCP Westwood does not have an effective mechanism to recognize the available bandwidth. This is an explanation of the fundamental problem: TCP Westwood cannot
adjust window size to an adequate value under various network environments. In a sense, traditional TCP Westwood can be considered to be a tool that measures available bandwidth because of its ability to adjust the congestion window size to achieve a transmission rate appropriate to the available bandwidth. However, it is ineffective because it only increases the window size by static value. **Our objectives:** We want to maximize the bandwidth utilization and the throughput of the connection. The largest advantage of the algorithm is that we can obtain the accurate estimation results for physical bandwidth even when the network is highly loaded.

**6.1 Principle Idea**

The purposed mechanism is enhancement to TCP Westwood by make changes at congestion avoidance and Timeout algorithms, the proposed mechanism is called WestwoodNew [67]. TCP Westwood can be considered to be a tool that measures available bandwidth because of its ability to adjust the congestion window size to achieve a transmission rate appropriate to the available bandwidth. However, it is ineffective because it only increases the window size by static value. This is a major reason that makes TCP Westwood inefficient in term of network utilization. The idea of the proposed algorithm WestwoodNew is to determine the optimal congestion window size at a TCP sender. The congestion window size will be dynamically calculated and adjusted based on the network conditions to improve the behavior of TCP. The following sections introduce the proposed TCP congestion control mechanism in details.

**6.2 Enhanced TCP Westwood Congestion Avoidance Algorithm**

TCP Westwood is a rate based scheme extending the TCP Reno. In Transmission Control Protocol (TCP), the congestion window is one of the factors that determine the number of bytes that can be outstanding at any
time. Maintained on the sender, this is a means of stopping the link between two places from getting overloaded with too much traffic. The size of this window is calculated by estimating how much congestion there is between the two places. The sender maintains the congestion window. When a connection is set up, the congestion window is set to the maximum segment size (MSS) then the size doubled every ack until Cwnd > Ssthresh then go to congestion avoidance state where size of Cwnd=Cwnd+1/Cwnd until congestion occur we propose enhanced in congestion avoidance algorithm as follow: TCP WestwoodNew takes the data-receiving rate as a metric to predict the network conditions. TCP WestwoodNew estimated BW as BWcurrent (the current BW after receive new ACK) then divide it on BWprevious (BW before receive the same new ACK) the result is the BW ratio if the BW ratio < 1 this indicate that there is an increase in the network load therefore the Cwnd should be constant else if BW ratio > 1 indicates that there is an decrease in the network load therefore the Cwnd should be increased the Cwnd is adjusted based on the network conditions estimate. These modifications constitute the foundation for an efficient congestion avoidance strategy over heterogeneous environments with wire-line or wireless networks. The TCP WestwoodNew congestion avoidance algorithm:

<table>
<thead>
<tr>
<th>Congestion avoidance</th>
</tr>
</thead>
<tbody>
<tr>
<td>slow start is over */</td>
</tr>
<tr>
<td>/*cwnd &gt; ssthresh */</td>
</tr>
<tr>
<td>Every Ack:</td>
</tr>
<tr>
<td>Estimate BWE</td>
</tr>
<tr>
<td>Set BWE = BWcurrent</td>
</tr>
<tr>
<td>BWratio = BWcurrent/BWprevious</td>
</tr>
<tr>
<td>If (1.5&gt;BWratio &gt;= 1)</td>
</tr>
<tr>
<td>cwnd = cwnd + 1/cwnd</td>
</tr>
<tr>
<td>If (BWratio &gt;= 1.5)</td>
</tr>
<tr>
<td>cwnd = cwnd + 2/cwnd</td>
</tr>
<tr>
<td>Else if (BWratio &lt; 1)</td>
</tr>
<tr>
<td>cwnd = cwnd + 0</td>
</tr>
<tr>
<td>Until (timeout or 3 DUPACKs)</td>
</tr>
</tbody>
</table>

Where BWcurrent: the current BW after receive new ACK, and BWprevious: BW before receive the same new ACK.
6.3 Modified RTO Calculation Algorithm

In the recent years the variety of Internet links with different properties has increased dramatically. The high speed networks have reached Gigabit rates, whereas the increasing number of mobile wireless access networks have introduced a prolific number of mobile hosts attached to the Internet through a slow, wireless links. Moreover, the challenging characteristics of wireless links, in particular high packet loss rate or delays due to various reasons such as link-layer retransmissions or hand-off between the points of attachment to the Internet, have introduced a large set of problems for the Internet transport protocols. The TCP protocol is the dominant Internet transport protocol and its congestion control algorithms are essential for the stability of the Internet.

Because these algorithms have a strong effect on TCP performance, finding solutions to improve TCP performance over slow wireless links. The traditional problem regarding the use of TCP over wireless links or other challenging channels has concerned TCP congestion control. If a packet is lost, TCP interprets it as an indication of congestion and a TCP sender needs to reduce its transmission rate. Hence, TCP performance deteriorates with increasing packet loss rate. If the packet loss occurred due to corruption, reducing the TCP transmission rate is, however, the wrong action to take. TCP uses the fast retransmit mechanism to trigger retransmissions after receiving three successive duplicate acknowledgements (ACKs). If for a certain time period TCP sender does not receive ACKs that acknowledge new data, the TCP retransmission timer expires as a back-off mechanism. When the retransmission timer expires, the TCP sender retransmits the first unacknowledged segment assuming it was lost in the network. Because a retransmission timeout (RTO) can be an indication of severe congestion in the network, the TCP sender resets its congestion window to one segment and starts increasing it according to the slow start algorithm. However, if the RTO occurs spuriously and there still are segments outstanding in the network, a
false slow start is harmful for the potentially congested network as it injects extra segments to the network at increasing rate. TCP should be able to adjust its RTO value when needed. This is realized according to the identified loss model within the network. First of all, let us note that when congestion is detected within the network, the RTO estimation is not changed and remains similar to the one used by TCP New Reno. Alternatively, in the case of wireless channel errors, no RTO calculation or adjustment is necessary as the network conditions are supposed to be unvaried. In the case of link failure, the RTO value has to be modified based on the characteristics (length, load, and link qualities) of the new route discovered by the routing protocol. So, after link loss recovery by the ad hoc routing protocol, we may observe that both the propagation and queuing delays change suddenly. As RTT is one of the most direct TCP connection characterization parameter that reflects network links conditions, our estimation algorithm will be depending on it. It is obvious that the number of hops as well as the load of the route between the TCP sender and receiver affects the RTT value over that connection. Thus, the characteristics of the new discovered route could be represented by RTT values over that route. Thus, the RTO value would be updated as follows:

\[
RTO_{\text{new}} = \frac{RTT_{\text{new}}}{RTT_{\text{old}}} \times RTO_{\text{old}} \quad (6.1)
\]

where RTTnew is the new round trip time estimation after congestion recovery and RTTold the round trip time estimation before congestion.

**6.4 WestwoodNew Mechanism Descriptions**

This section describes the Proposed Mechanism for TCP Congestion Control: TCP WestwoodNew. The TCP WestwoodNew uses the four algorithms: Slow Start, Congestion Avoidance which it enhances, Fast Retransmit to contain just retransmitting the loss packet, and enhances the Fast Recovery
Algorithm. Table 6.1 shows Pseudo code of the WestwoodNew congestion control algorithms.

| Slow Start: | Initial: cwnd = 1;  
|             | For (each packet Acked)  
|             | cwnd++;  
|             | Until (congestion event, or, cwnd > ssthresh) |
| Congestion Avoidance: | /* slow start is over */  
|                      | /*cwnd > ssthresh */  
|                      | Every Ack:  
|                      | Estimate BWE  
|                      | Set BWE = BWcurrent  
|                      | BWratio = BWcurrent/BWprevious  
|                      | If (1.5>BWratio >= 1)  
|                      | cwnd = cwnd * 1/cwnd  
|                      | If (BWratio >= 1.5)  
|                      | cwnd = cwnd * 2/cwnd  
|                      | Else if (BWratio < 1)  
|                      | cwnd = cwnd * 1  
|                      | Until (timeout or 3 DUPACKs) |
| Fast Retransmit: | After receive 3 DUPACKs  
|                   | Send that packet;  
|                   | Invoke Fast Recovery algorithm |
| Fast Recovery: | i-With 3 DUPACKs:  
|                | Estimate RTTnew  
|                | Set RTOnew=(RTTnew/RTTold)*RTOold  
|                | Estimate BWE  
|                | ssthresh = (BWE*RTTmin) /seg_size; and if  
|                | cwnd > ssthresh  
|                | then set  
|                | cwnd = ssthresh  
|                | Invoke Congestion Avoidance Algorithm;  
| ii-When TimeOut: | ssthresh = (BWE*RTTmin) /seg_size;  
|                 | if (ssthresh < 2)  
|                 | ssthresh =2  
|                 | cwnd = 1;  
|                 | Invoke Slow Start Algorithm; |

Table 6.1: The WestwoodNew Congestion Control algorithms Pseudo code.

5.5 Implementation
In this section, the implementation of the proposed mechanism will be outline. The proposed mechanism is implemented in the Network Simulator.
Chapter 6  Proposed Mechanism (TCP WestwoodNew)

(NS-2)[68] under UNIX environment, to be available for examining under different network topologies. Figure 6.1 shows the architecture of the proposed mechanism implemented in the Network Simulator (NS-2) under UNIX environment, to be available for examining under different network topologies.

![Figure 6.1: The architecture of the proposed mechanism](image)

As in Figure 6.1 when new data is generated at the application, the data is passed to the TCP layer. The data is passed to the IP layer after TCP protocol processing, and the resulting IP packets are injected into the network. Conversely, an ACK packet that arrives at the IP layer of the sender host is passed to the TCP layer to process. The congestion window size of a TCP connection is updated when an ACK packet is passed to the tcp recv() function. Therefore, the control program for the congestion window size for the proposed mechanism should be implemented in the tcp recv() function. The tcp recv() function calls the open cwnd() function and updates the congestion window size when an ACK packet arrives.. the degree of the
increase of the congestion window size based on the bandwidth value. Figure 6.2 shows the flow chart of the proposed mechanism.

First, the mechanism compares the congestion window size (cwnd) and the slow start threshold (ssthresh). When cwnd is smaller than ssthresh, the congestion window size is updated by the slow start algorithm as TCP Reno. On the other hand, when cwnd is larger than ssthresh, the congestion window size is determined based on the algorithm of the proposed mechanism. The increase degree of the congestion window size is determined on consideration of k. Finally, cwnd is updated by equation (6.2) as follow:

$$Cwnd = cwnd + \frac{k}{cwnd}$$

(6.2)
For the implementation of the timeout:

- When Loss detection then timeout will be determined we find that
- Long timeout cause Long latency in detection loss
- While short timeout cause unnecessary packet transmission
- TCP timeout is estimated based on the round trip time (RTT)
- The proposed mechanism should be implemented in the rtt_timeout() function which Computes the bounded RTO value

PROCEDURE 1: shows the implementation for the new Congestion Avoidance Algorithm at opencwnd() function which is called within recv() function to update cwnd value each time a new ACK is received.

PROCEDURE 2: shows the implementation of the Fast Recovery Algorithm. The implementation depend on developing the function, rtt_timeout().

PROCEDURE 1:

```
1 void TcpAgent::opencwnd()
2 { double increment ;
3   double past, current ;
4   past=1_current_bwe_ ;
5   current=current_bwe_ ;
6   double ratio=current/past ;
7   if (cwnd_ < ssthresh_){
8     cwnd_ += 1 ;
9   } else { /* linear */
10     double f;
11     switch (wnd_option_) {// here is equal to 1
12       case 0:
13           if (++count_ >= cwnd_) {
14               count_ = 0; ++cwnd_; }
15           break;
16       case 1:
17           if(ratio>=1.5)
18               increment= 2*(increase_num_ / cwnd_); //increase_num_ =1
19           else if (1.5>ratio>=1)
20               increment = (increase_num_ / cwnd_);
21           else if(ratio<1)
22               increment=0;
23           if ((last_cwnd_action_ == 0 ||
```
last_cwnd_action_ == CWND_ACTION_TIMEOUT) && max_ssthresh_ > 0){
    increment = limited_slow_start(cwnd_, max_ssthresh_, increment); }
  cwnd_ += increment;
  l_ current_bwe_ = current_bwe_ ;
  break;

PROCEDURE 2:
1 double TcpAgent::rtt_timeout()
2 {double timeout ;
3  double past, current ;
4   past = l_t_rtt ;
5   current = t_rtt ;
6 double ratio=current / past
7 timeout = timeout_prev * ratio
8   if (timeout > maxrto_)
9      timeout = maxrto_; 
10     if (timeout < minrto_)
11        timeout = minrto_; 
12    if (timeout < 2 * tcp_tick_)
13       if (timeout < 0 ){
14           fprintf(stderr, "TcpAgent: negative RTO! (%f)\n" timeout);
15              exit(1);
16      }
17     timeout = 2 * tcp_tick_; 
18   } } 
19   l_timeout = timeout_prev ;
20  Return timeout() ;
Chapter Seven

Simulation Results and Discussions

As mentioned previously, several mechanisms were proposed by researchers to improve congestion control. These mechanisms include TCP Tahoe, Reno, Vegas, SACK, NewReno, Westwood and WestwoodNew. In this chapter the current congestion control mechanisms are evaluated by simulation considering throughput, losses, delay, and congestion window that provided by each mechanism. This study is done using the well known network simulator NS-2 and a realistic topology generator called GT-ITM.

Generally, the performance evaluation of algorithms and policies are often investigated by simulation or analysis. The reason is clear; networks that are large enough to be interesting are also expensive and difficult to control. Therefore they are rarely available for experimental purposes. Moreover, it is generally more efficient to assess solutions using analysis or simulation providing the model is a "good" abstraction of the real network and application. But with the modern data communication networks which are extremely complex and do not lend well to theoretical analysis. It is common that network analysis can be rigorously made after leaving out several, sometimes subtle, details that cannot be easily captured in the analysis.

Moreover, simulations are complementary to analysis, not only for providing a check on the correctness of the analysis, but also allowing exploration of complicated scenarios that would be either difficult to analyze. Simulations
allow researchers to test scenarios that might be particularly difficult or expensive to emulate using real hardware for instance. Simulations can also play a vital role in helping researchers to develop intuition. In particular, the complexities of Internet topologies and traffic, and the central role of adaptive congestion control, make simulation the most promising tool for addressing many of the questions about Internet traffic dynamics [69].

7.1 Simulation Environments
In this study, the well known Network Simulator NS-2 has been used to evaluate the different variants of TCP congestion control mechanisms. In addition, the evaluation is done considering different network topologies that are generated by the GT-ITM generator. The simulation is done under UNIX environment.

The NS-2 is an event driven, object oriented network simulator enabling the simulation of different network (LAN, WAN) considering different protocols, traffic sources, and router queue managements [70]. The NS-2 is written in both OTCL and C++ languages. That is for efficiency reason, where NS separates the data path implementation from control path implementations. In order to reduce packet and event processing time (not simulation time), the event scheduler and the basic network component objects in the data path are written and compiled using C++ programming language. These compiled objects are made available to the OTCl interpreter through an OTCl linkage. It is also possible to Add New Application and Agent using C++. While C++ is fast to run but slower to change, making it suitable for detailed protocol implementation, OTCl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. NS-2 can be built and run under UNIX and Windows. For more detail about NS-2 network simulator, please refers to appendix A.
In investigating the TCP mechanisms by simulation, the selected topology often influences simulations outcomes; hence, realistic topologies are needed to produce realistic results. It have been shown that the first principle approach for modelling the Internet connectivity at the router level provides direct and immediate insight into the approaches that based on topology generation [71]. The development of abstract, yet informed, models for network topology evaluation and generation has been largely empirical in nature, for example, GT-ITM [72], BRITE [73], and TIERS [74]. The networks that generated by the topology generators are pseudo-random in the sense that they are randomly generated within the constraints of various properties that have been identified as existing in many real world networks. From these wide ranges of topology generators, GT-ITM (Georgia Tech Inter-network Topology Models) topology generator has been used to produce realistic topology. The GT-ITM is a collection of software tools for creation, manipulation, and analysis of graph models of internet topology. It has been used by networking researchers in a variety of ways, most often to create topologies for use in simulation studies. GT-ITM has been enhanced to include visualization capabilities, a routing and forwarding module for use with large graphs, and support for modelling of inter-domain routing policies [75]. For more detail about GT-ITM topology generator please refer to appendix B.

The GT-ITM topology generator creates a graph representation of a network based on a model and a set of parameters specified by the user. These representations are often used as input to a simulation. If the graph approximates characteristics of a real network, then the simulation results predict the behaviour of a network protocol or application if it were to be deployed. It have been proved that the level of similarity for the created topology using GT-ITM generator is high and indicate that the hierarchical
topology generators are able to produce realistic router level topology, with a similarity to the real network reaching the order of 96.6%.

7.2 Evaluation Criteria

Criterion is one of the most important issues for evaluating the network traffic control mechanisms and algorithms. Due to multiple performance objectives of the network traffic control, evaluation criteria must include multiple performance metrics [76]. In this study, four performance metrics are used in evaluating the TCP congestion control mechanisms. These include network throughput under different protocols, packet delay, and packet drop rates or simply packet losses, and congestion window. While the major aim of enhancing the congestion control algorithms is to provide the user with high quality of services, so the used metrics are selected from the used point of view. In order to measure the performance of our proposal with the Standard TCP and other protocols in a Network Environment, We need to define performance Metrics. The following metrics:

1- Throughput
2- Delay
3- Packet loss
4- Congestion Window (Cwin)

The following section briefly describes the metrics that will be considered in evaluating the TCP congestion control mechanisms.

7.2.1 Throughput

Throughput we consider it as the most significant metric. It is defined as the rate at which a system can process a given computation [77]. In communication networks, the throughput is measured in packets per second (pps) or bits per second (bit/s), refers to the number of packets sent from the source and correctly received by the destination. It may be the amount of data that is delivered to a certain network terminal or host computer, or between
two specific computers. The throughput is usually measured in bits per second (bits/s or bps), occasionally in data packets per second or data packets per timeslot. It have been noted that, maximizing throughput is of concern in a wide range of environments, from highly congested networks to underutilized ones. Hence, it is a clear goal of most congestion control mechanisms to maximize throughput, subject to application demand and to the constraints of the other metrics. In some contexts, the distribution of per flow throughput is being considered. Where some researchers evaluate transport protocols in terms of maximizing the aggregate user utility, where a user’s utility is generally defined as a function of the user’s throughput. Individual applications can have application specific needs in terms of throughput. For example, real-time video traffic can have highly variable bandwidth demands; VoIP traffic is sensitive to the amount of bandwidth received immediately after idle periods. Thus, user metrics for throughput can be more complex and rarely to be used.

7.2.2 Delay
The delay time [78] is significant in systems that require two-way "interactive" communication, such as voice telephony, or ACK/NAK data systems. It may range from a very few microseconds for a short line-of-sight (LOS) radio system to many seconds for a multiple link circuit with one or more satellite links involved. This includes the node delays as well as the media transit time. Delay can be measured as per packet transfer times. For reliable transfer, the per packet transfer time includes the possible delay of retransmitting a dropped packet. Users of bulk data transfer applications might care about per packet transfer times as so far as they affect the per connection transfer time. On the other hand, for users of streaming media, per packet delay can be a significant concern, while some users might care about the worst case delay or about the delay distribution.
Chapter 7

Simulation Results and Discussions

7.2.3 Packet Losses

Packet losses (packet drop rates) [79] can be measured as the number of dropping packets with the time metric. It is also defined as the packets that are retransmitted from the sender whatever is corrupted or lost. Some users might care about packet drop rates only in so far as they affect per connection transfer times, while other users might care about packet drop rates directly. It is highly important reason to avoid high packet dropping rate, which will lead to avoid congestion collapse in environments containing paths with multiple congested links. In such environments, high packet dropping rate could result in congested links wasting the bandwidth by carrying packets that will only be dropped downstream, before being delivered to the receiver.

7.2.4 Congestion Window

As mentioned previously, window based mechanisms control the data transmission rate based on both the congestion window size and the receiver’s advertised window size that reflect the available channel capacity and receiver buffer space. The receiver’s advertised window is a receiver’s base adjustment, depending on the resources of the receiver, while congestion window size is window that reflects the network status. So congestion window size is the main target for congestion control mechanisms.

7.3 Simulated Network Topologies & Simulation Results

We have used in our simulations two network scenarios, called network topology 1 and network topology 2. The latter is an evolution of the former.

6.3.1 Network Topology 1 (NT1)

Figure 7.1 shows the network topology (NT1) that is used for the simulation. The topology has five nodes connected to each other via four TCP connections; each link is labelled with its bandwidth capacity and its delay.
In the following text, the metrics (throughput, delay, packet losses, and congestion window) are evaluated with the time using ns-2 simulator. The performance evaluation is discussed for each metric as shown in the following subsections.

**Throughput:**

In computer networks, the cumulative or aggregate throughput is the sum of the data that are delivered to all terminals over the time while the instant throughput is the throughput value at each time. In most cases it might be sufficient to consider the aggregate (cumulative) throughput.

During this evaluation the cumulative or aggregate throughput is used and is calculated as the collection of the received data over the time. The throughput is determined in kilobit per second (Kb/s), where the amount of the receiving is calculated every Time Interval length (TIL) and divided by the time. Define Recdata as the data received, the throughput is:

$$\text{throughput} = \frac{((\text{Recdata} \times 8)/\text{time})}{1000}$$

Figure 7.2 shows the throughput for all TCP congestion control mechanisms; Tahoe, Reno, NewReno, SACK, Vegas, Westwood and WestwoodNew. The results are given using TIL=1 second. As shown in the figure, for all TCP variant, the throughput increases exponentially in slow start phase. Where, the slow start algorithm duplicates the congestion window every RTT.
and duplicated the sending rate. Entering in congestion avoidance phase depends on the ssthresh value which is determined from the first flow packet in the transmission. By entering the congestion avoidance phase, the congestion window size increases linearly until packet losses which are translated as congestion indication. Adding of fast retransmit algorithm improves the performance of TCP over its earlier implementation.

In Tahoe TCP, the connection always goes to slow start after a packet loss. However, if the window size is large and packet losses are rare, it would be better for the connection to continue from the congestion avoidance phase, since it will take a while to increase the window size from 1 to ssthresh value. It has been found that, with TCP Tahoe, the sender may retransmit packets which have been received correctly and that decreases throughput. The purpose of the fast recovery algorithm in Reno TCP is to overcome this behaviour. So as it could be seen, Reno’s Fast Recovery algorithm is optimized for the case when a single packet is dropped from a window of data. However, Reno can suffer from performance problems when multiple packets are dropped from a single data window. With multiple packet losses TCP Reno is timeout and trigger slow start phase. So Reno has the same performance of TCP Tahoe with multiple packets losses.

TCP NewReno is improved over Reno when multiple packet losses from single window. As shown in the Figure 7.2, the NewReno increases the overall throughput than Reno. The throughput of NewReno is higher than that of Reno. It should be noted that TCP NewReno retransmitting one lost packet per round-trip time until all of the lost packets from that window have been retransmitted. TCP SACK tries to eliminate the problem of multiple packets lost from one window using other technique than NewReno. The TCP SACK adds selective acknowledgement and selective retransmit that the TCP sender can detect single window multiple packet losses using the SACK option within the acknowledgment and selectively retransmit lost packets.
Figure 7.2: Throughput vs. Simulation Time for NT1.

Even the adding SACK option to the acknowledgement increase the load in the network and need improvement in both TCP sender and receiver to enable SACK option. It could be noted from the figure that TCP Sack throughput is higher than Reno due to the little changes. TCP Vegas attempts to increase its congestion window (cwnd) according to the available bandwidth. It is important to note that the TCP Vegas tries to avoid congestion more than just treating with it. From this figure, Vegas get the lowest throughput value. The reasoning is that when the actual throughput of a connection approaches the value of the expected maximum throughput, it may not be utilizing the intermediate routers’ buffer space efficiently, and hence it should increase the flow rate.

On the other hand, when the actual throughput is much less than the expected throughput, the network is likely to be congested and hence the sender should reduce the flow rate. It is noted from this figure that TCP WestwoodNew has highest throughput on the steady state time. This is because TCP Westwood
new interested by the network status in now and the past therefore it
determines the network status by a large percentage of accuracy. TCP
Westwood determines the network status by estimate the current BW without
looking to the previous BW which determine the previous network status,
therefore the rate of the sending packets in TCP Westwood and base on that
TCP Westwood sending the same rate of packets whether the network status
is heavy or not heavy by TCP WestwoodNew determine the rate of packets
sending based on the network status if it not heavy TCP WestwoodNew
increase the rate of the sending packets which is increase the throughput
performance in the network if network status is heavy.

![Throughput](image)

**Figure 7.3:** Throughput versus time for Westwood and WestwoodNew for
NT1.

TCP WestwoodNew remains the rate of sending packets constant, and when
comparing the Westwood with WestwoodNew, it noted from the zoom of
Figure 7.2, in Figure 7.3 that depicts the throughput WestwoodNew is
improved because of taking the network status in now and the past to
adapting the congestion window.
Delay:
Communication delay is comprised of two elements: propagation delay, and queuing delay. The propagation delay is the physical delay in transmission media where the data along a length of link. The queuing delay is the delay experienced by data waiting to be served at resources (routers) within the network.

![Delay of mechanisms](image)

Figure 7.4: Delay vs. Simulation Time for NT1.

So the queuing delay is the primary source of communication delay in the network, while the propagation delay is constant for the same link.

In this evaluation the delay have been measured as per packet transfer times, like throughput the delay measured cumulatively with the time. Figure 7.4 shows the cumulative delay for different TCP congestion control mechanisms. As shown in the figure, TCP WestwoodNew has a better delay behave than the others variants, unlike Westwood which have higher delay than it. That is because cwnd of WestwoodNew determined by more accurate
than cwnd of Westwood this determination based on the degree the network get congested so WestwoodNew could send more new packets depending on the network status that reduces the delay time.

![Delay of mechanisms](image)

Figure 7.5: Delay versus the time for Westwood and WestwoodNew for NT1
Also WestwoodNew determine RTO based on new RTT and old RTT this make it avoid to send the packets once again without necessary to that and avoid a false slow start. When comparing the Westwood with WestwoodNew, it noted from the zoom of figure 7.4, in Figure 7.5 that depicts the delay of WestwoodNew is improved as the minimum delay.

**Packet Losses:**
Figure 7.6 shows the cumulative sum of all dropped packets plotted with time, where the number of dropped packets is summed cumulatively and the summation is determined each time interval length (one second in the graph).
Figure 7.6: Packet Losses vs. Time for NT1.

This figure shows that, the dropped packets by TCP Westwood have the highest dropping rate over the simulation time cause in fast recovery Westwood send more number of packets than other mechanisms which reduce the no. of sending packets in fast recovery, WestwoodNew improved Westwood by more adapting to the number of sent packets in fast recovery depending on the past and current network status. When comparing the Westwood with WestwoodNew, it noted from the zoom of Figure 7.6, in Figure 7.7 that depicts the packet losses of WestwoodNew is improved more than Westwood as less packet losses with highest throughput and minimum delay.
Figure 7.7: Packet Losses versus the time for Westwood and WestwoodNew for NT1

**Congestion Window:**

Figure 7.8 shows the behaviour of congestion window size \( cwnd \). As shown in the figure, the congestion window grows, exponentially as in slow start, and linearly as in congestion avoidance, until some errors occur. With Tahoe TCP, the sender retransmits the lost packet and then sets the \( ssthresh \) to the half of the congestion window size and \( cwnd \) to 1 segment. The Reno TCP reduces its \( cwnd \) to the \( ssthresh \). It could be noted from the figure that it return again to one packet. On the other hand, NewReno and SACK get over multiple packet losses using their Fast Recovery algorithm, which differentiate between fall ACK and partial ACK. In TCP Vegas, \( cwnd \) behaviour is different from that of Tahoe, Reno, NewReno and SACK. It could be noted that Vegas congestion window does not increase as the others, but much more slower, Vegas try to keeps track of the optimal window size,
With TCP Westwood and TCP WestwoodNew, we find that when congestion detected TCP Westwood and TCP WestwoodNew determines the cwnd based on the bandwidth estimation in fast recovery then go to congestion avoidance state with new cwnd in TCP Westwood congestion avoidance.

![Congestion Window Graph](image)

**Figure 7.8:** Behaviour of Congestion Window (cwnd) for NT1

We find that when new ACK is received cwnd is increased by 1/cwnd in TCP WestwoodNew congestion avoidance if new ACK is received, the rate of cwnd increasing determine based on network status if heavy cwnd remain constant, else if network status not heavy cwnd multiplying by 1/cwnd or 2/cwnd, so we find that the rate of TCP WestwoodNew Cwnd larger than the rate of TCP Westwood cwnd. Figure 7.9 depicts the Behaviour of the Congestion Window for Westwood and WestwoodNew. We note that the cwnd of WestwoodNew is growing than cwnd of Westwood, because WestwoodNew exploits the available bandwidth by more precious to enlarge its congestion window and sending rate.
Figure 7.9: Behaviour of the Congestion Window for Westwood and WestwoodNew for NT1

6.3.2 Network Topology 2 (NT2)

Figure 7.10 shows the network topology NT2 that is used for the simulation.

The topology has 189 nodes connected to each other, the topology generated by the GT-ITM generator and used for evaluating the TCP variants. While on creating this topology, the GT-ITM is based on model like flat random, N-level hierarchical and transit-stub hierarchical and a set of parameter. In the following text, the metrics (throughput, delay, packet losses, and congestion
window) are evaluated with the time for NT2 using ns-2 simulator. The performance evaluation is discussed for each metric as shown in the following subsections.

**Throughput:**

![Throughput vs. Simulation Time for NT2](image)

Figure 7.11: Throughput vs. Simulation Time for NT2

Figure 7.11 shows the throughput for all TCP congestion control mechanisms; Tahoe, Reno, NewReno, SACK, Vegas, Westwood and WestwoodNew versus simulation time with network NT2. It is noted that the WestwoodNew has the highest throughput because of the reasons that mentioned before in the description of Figure 7.2. With comparing the Westwood and WestwoodNew, we find the WestwoodNew has higher throughput than that in Westwood, as shown in Figure 7.12, because of the same reasons mentioned before.
Figure 7.12: Throughput versus time for Westwood and WestwoodNew for NT2

Delay:
Figure 7.13 shows the delay for all TCP congestion control mechanisms; Tahoe, Reno, NewReno, SACK, Vegas, Westwood and WestwoodNew versus simulation time with network NT2. It is noted that the WestwoodNew has the lowest delay because of the reasons that mentioned before in the description of Figure 7.4. With comparing the Westwood and WestwoodNew, we find the WestwoodNew has lower delay than that in Westwood, as shown in Figure 7.14, because of the same reasons mentioned before.
Figure 7.13: Delay vs. Simulation Time for NT2

Figure 7.14: Delay versus the time for Westwood and WestwoodNew for NT2
Packet loss:
Figure 7.15 shows the packet losses for all TCP congestion control mechanisms; Tahoe, Reno, NewReno, SACK, Vegas, Westwood and WestwoodNew versus simulation time with network NT2.

It is noted that the WestwoodNew has a little more packet losses than that of Vegas, Tahoe, and Reno and lower packet losses than that of Westwood, NewReno, and sack. It is because of the same reasons mentioned for Figure 7.6. With comparing the Westwood and WestwoodNew, we find the WestwoodNew has higher throughput than that in Westwood, as shown in Figure 7.16, because of the same reasons mentioned before.

Figure 7.15: Packet Losses vs. Time for NT2.
Chapter 7  
Simulation Results and Discussions

Figure 7.16: Packet Losses versus the time for Westwood and WestwoodNew for NT2.

**Congestion Window:**

Figure 7.17 shows the congestion window for all TCP congestion control mechanisms; Tahoe, Reno, NewReno, SACK, Vegas, Westwood and WestwoodNew versus simulation time with network NT2.

It is noted that the WestwoodNew has a more congestion window than that of Vegas, Tahoe, Reno, Westwood, NewReno, and SACK. It is because the WestwoodNew has highest throughput for the same reasons mentioned before in Figure 7.9. With comparing the Westwood and WestwoodNew, we find the WestwoodNew has higher congestion window than that in Westwood, as shown in figure 7.18, because of the same reasons mentioned before.
Figure 7.17: Behaviour of Congestion Window (cwnd) for NT2

Figure 7.18: Behaviour of the Congestion Window for Westwood and WestwoodNew for NT2.
Chapter Eight

Conclusions and Future Work

The study of protocol TCP Westwood enabled us to raise the problem of congestion window evolution. To improve the transfer of data, it is necessary to resolve this problem. We proposed a new mechanism based on TCP Westwood to reduce the margin in which can evolve the congestion window and modify the window evolution according to time. Our contributions made it possible to increase the throughput without increasing the packet loss rate. And reduce the delay. After that, we compared this modification with other mechanisms and fine that he has a throughput higher than of them that is best suited for the Internet than other mechanisms. We come to a conclusion that TCP WestwoodNew, among other protocols, is the one with a better performance, with regards to throughput and delay. The following sections summarize in brief our contributions, concluding remarks, and some directions for future work.

8.1 Overview

One of the main problems that face the Internet performance and the continuing success of the Internet is the congestion. Hence, congestion control is currently a large area of research and concern in the network community, to avoid this problem. Generally, Congestion control can be implemented in two ways: network-assisted congestion controls (approaches
taken by routers) and end-to-end congestion controls (approaches taken by TCP at end nodes). In this thesis, we are precisely concerned with the end-to-end congestion control: Evaluation and Enhancement. The end-to-end congestion control has several implementation versions of mechanisms which intend to improve congestion control. These mechanisms include TCP Tahoe, Reno, NewReno, SACK, Vegas and Westwood. The performance of the existing mechanisms is evaluated and compared with each other over a real network topology. The purpose of this investigation is to show the performance of the existing mechanisms and to proof the best of them. Furthermore, this investigation is important for declaring the weakness points in the current mechanisms. We propose an adaptive dynamic algorithm, called WestwoodNew, was developed. The proposed algorithm is based on the enhancement of both the Congestion Avoidance and the Fast Recovery algorithms of the current TCP Westwood mechanism. Indeed, the enhancements are considered to address some important issues that affect the Internet growth and the deployment of its applications. Through this thesis, the performance of both the current mechanisms and the proposed mechanism are evaluated on different environments by simulation, using the well known network simulator NS-2 and a realistic topology generator called GT-ITM. The performance of the current mechanisms is evaluated over a real network topology, i.e., AT&T network topology, generated by the GT-ITM generator. On the other hand, the performance of the proposed mechanism is evaluated and compared with the existing mechanisms over two scenarios of network topologies. The first Scenario uses the AT&T realistic network topology that generated by the GT-ITM. The second scenario uses a network topology with link error.
8.2 Concluding Remarks

As a rough conclusion from the survey, several congestion control mechanisms are developed and refined by researchers aiming to improve congestion control. These mechanisms include TCP Tahoe, Reno, NewReno, SACK, Vegas and Westwood. The Tahoe is first implementation. It works in three phases: Slow Start, Congestion Avoidance and Fast Retransmit. The Reno implementation retained the enhancements incorporated into Tahoe, but, the Fast Retransmit phase is modified to include Fast Recovery. The NewReno enhances the TCP Reno to include multiple packets dropped from a single window. It adds small changes to the Reno mechanism at the sender in order to eliminate the wait for a retransmit timer when multiple packets are lost from a window. The TCP SACK is a conservative extension to Reno to support Selective Acknowledgement (SACK) option by both ends. The TCP Vegas extends the Reno’s retransmission mechanisms. TCP Westwood enhances the NewReno by using the bandwidth to estimate the congestion window more precious. This survey is very important to other researchers who will work in the same field. This is because; it gives the reader a general overview and a brief description of different mechanisms.

Through this thesis, the current congestion control mechanisms are evaluated and compared with each other on different environments. This comparative study can help other researchers and developers to know the strong and weakness of each mechanism.

Based on the weakness in the behaviour of the TCP Westwood, a new adaptive dynamic algorithm, called WestwoodNew, is developed. The Proposed mechanism modifying both the Congestion Avoidance and the Fast Recovery algorithms of the current TCP Westwood mechanism so as to allow the window size to change based on the network situation.

A number of simulation studies have been performed on different environments to evaluate the performance of the proposed mechanism and
compare its performance with the existing mechanisms. The simulation results showed that, the introduced congestion control scheme significantly improved the TCP performance compared to that of existence TCP congestion control schemes. Specifically, it improves the throughput. TCP WestwoodNew controls its window to transmit data by a rate that increasing the throughput. Hence, this approach may be used by the network community to overcome the congestion problem from the end nodes.

8.3 Future Research Directions

There are several interesting directions for future work.

- In future studies, it is intended to evaluate the interaction between multiple connections of the proposed mechanism and the fairness among UDP.

- Implementation experiments are also remaining task.

- Further work should focus on whether WestwoodNew performs well with actual applications like, www, e-mail, etc.

- Finally, applications which do not use TCP are becoming more prevalent in the Internet, and many of these applications pay little or no attention to congestion control issues. The more predictable behaviour and better understanding of TCP congestion control may be a step toward standardized transport layer congestion behaviour for use by all Internet applications.
Appendix A

Network Simulator

A network simulator is a piece of software that predicts the behavior of a network, without an actual network being present. Network simulators serve a variety of needs. Compared to the cost and time involved in setting up an entire test bed containing multiple networked computers, routers and data links, network simulators are relatively fast and inexpensive. Networking simulators are particularly useful in allowing designers to test new networking protocols or changes to existing protocols in a controlled and reproducible environment. Network simulators, as the name suggests are used by researchers to design various kinds of networks, simulate and then analyze the effect of various parameters on the network performance. A typical network simulator, like NS2, encompasses a wide range of networking technologies and helps the users to build complex networks from basic building blocks like variety of nodes and links. With the help of simulators one can design hierarchical networks using various types of nodes like computers, hubs, bridges, routers, optical cross-connects, multicast routers, mobile units, etc.

There are a wide variety of network simulators, ranging from the very simple to the very complex. Minimally, a network simulator must enable a user to represent a network topology, specifying the nodes on the network, the links between those nodes and the traffic between the nodes. More complicated systems may allow the user to specify everything about the protocols used to handle network traffic.
Graphical applications allow users to easily visualize the workings of their simulated environment. Text-based applications may provide a less intuitive interface, such as NS2, but may permit more advanced forms of customization. Also it is programming-oriented, and providing a programming framework that the user then customizes to create an application that simulates the networking environment to be tested.

NS2 is an open-source simulation tool that runs on Linux. It is a discreet event simulator targeted at networking research and provides substantial support for simulation of routing, multicast protocols and IP protocols, such as UDP, TCP, RTP and SRM over wired and wireless (local and satellite) networks. It has many advantages that make it a useful tool, such as support for multiple protocols and the capability of graphically detailing network traffic. Additionally, NS2 supports several algorithms in routing and queuing. LAN routing and broadcasts are part of routing algorithms. Queuing algorithms include fair queuing, deficit round-robin and FIFO.

Network Simulator NS2

NS2 started as a variant of the REAL network simulator in 1989 [81] (see Resources). REAL is a network simulator originally intended for studying the dynamic behavior of flow and congestion control schemes in packet-switched data networks? Currently NS2 development by VINT group is supported through Defense Advanced Research Projects Agency (DARPA) with SAMAN and through NSF with CONSER, both in collaboration with other researchers including ACIRI (see Resources). NS2 is available on several platforms such as FreeBSD, Linux, SunOS and Solaris. NS2 also builds and runs under Windows. NS (version 2) is an object-oriented, discrete event driven network simulator developed at UC Berkeley. NS is primarily useful for simulating local and wide area networks. Although NS is fairly easy to use once you get to know the simulator, it is quite difficult for a first time user, because there are few user-friendly manuals. Even though there is a lot
of documentation written by the developers which has in depth explanation of the
simulator, it is written with the depth of a skilled NS user.

NS is an object oriented simulator, written in C++, with an OTcl interpreter as
front-end. The simulator supports a class hierarchy in C++ (also called the
compiled hierarchy), and a similar class hierarchy within the OTcl interpreter (also
called the interpreted hierarchy in this document). The two hierarchies are closely
related to each other; from the users' perspective, there is a one-to-one
correspondence between a class in the interpreted hierarchy and one in the
compiled hierarchy. The root of this hierarchy is the class TclObject. Users create
new simulator objects through the interpreter; these objects are instantiated within
the interpreter, and are closely mirrored by a corresponding object in the compiled
hierarchy. The interpreted class hierarchy is automatically established through
methods defined in the class TclClass. User instantiated objects are mirrored
through methods defined in the class TclObject. There are other hierarchies in the
C++ code and OTcl scripts; these other hierarchies are not mirrored in the manner
of TclObject.

But, why two languages? NS uses two languages because simulator has two
different kinds of things it needs to do. On one hand, detailed simulations of
protocols require a systems programming language which can efficiently
manipulate bytes, packet headers, and implement algorithms that run over large
data sets. For these tasks run-time speed is important and turn-around time (run
simulation, find bug, fix bug, recompile, re-run) is less important. On the other
hand, a large part of network research involves slightly varying parameters or
configurations, or quickly exploring a number of scenarios. In these cases, iteration
time (change the model and re-run) is more important. Since configuration runs
once (at the beginning of the simulation), run-time of this part of the task is less
important. ns meets both of these needs with two languages, C++ and OTcl. C++ is
fast to run but slower to change, making it suitable for detailed protocol
implementation. OTcl runs much slower but can be changed very quickly (and
interactively), making it ideal for simulation configuration. NS (via tclcl) provides glue to make objects and variables appear on both languages.

**Source code**

The ns2 project is released under the BSD is as license. Due to the project being under this license, the source code is distributed as well, enabling for the user community to participate in the development of this simulator.

**TCL/OTCL and C++ programming**

NS uses two languages because simulator has two different kinds of things it needs to do. On one hand, detailed simulations of protocols require a systems programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important. On the other hand, a large part of network research involves slightly varying parameters or configurations, or quickly exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulation), run-time of this part of the task is less important. ns meets both of these needs with two languages, C++ and OTcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. NS (via tclcl) provides glue to make objects and variables appear on both languages.

The simulator makes a distinction between two layers of code. One layer is C++ and is the core of the simulator; it has the modules implementing the protocols like TCP and IP, the actual simulator, queuing systems, etcetera. This could be called the framework which is the basis of this simulator. On the other hand a TCL file is used for the more dynamic changing part of the program, namely the configuration.
file of the actual network. Although it is possible to write complete programs or
scripts in this TCL file, Tcl, the Tool Command Language, is an interpreted
scripting language. It is used by the NS-2 network simulator to build simulation
scenarios. The following section provides enough information on Tcl to understand
a simple simulation scenario for NS-2. The main intent is small scripts a creation of
objects from the core program. There is an interface between the TCL and C++ to
exchange values of object parameters. The idea behind this setup is to have a fast
(precompiled) C++ core of the simulator and still be able to make and use
intelligent configuration files which are parsed and built run-time.

TCL Programs

This is a small TCL script that used simulation topology from RFC 2415-1998, (A
Simulation-Based Study of HighSpeed TCP and its Deployment MAY2003). 14
node, forward path 4 connections and reverice path 2 connections, using TCP
protocol. The script:

```
########setup new simulator###
set ns [new Simulator ]
$ns color 1 Red
$ns color 2 Green
$ns color 3 Blue
$ns color 4 Brown
$ns color 5 Orange
$ns color 6 Yellow
set lrate [lindex $argv 0]

###setup nam& trace files##########
set nam1 [open tout1.nam w]
$ns namtrace-all $nam1
set trace1 [open tout1.tr w]
$ns trace-all $trace1

########### finish procedure #########
proc finish {} {
    global ns nam1 trace1
    #finishing traceing
    $ns flush-trace
    #closing nam&trace files
    close $nam1
    close $trace1
    #executing nam
    exec nam tout1.nam &
```
Note: The code contains logical syntax errors and is not fully documented. The code appears to be a script for defining a network topology using the Network Simulator (ns-2). Here is a step-by-step explanation of the code:

1. **Creating Topology**
   - A loop is used to create nodes labeled 'server' and connect them with links of varying speeds and delay.
   ```tcl
   for { set i 0} {$i < 14} { incr i} {
       set n$i [ns node]
   }
   ```
   - Each node is set to be a hexagon, blue in color, and labeled 'server'.
   ```tcl
   $n0 shape hexagon
   $n0 color Blue
   $n0 label server
   $n1 shape hexagon
   $n1 color Blue
   $n1 label server
   ```
   - Links are created between these nodes, with specified bandwidth, delay, and drop tail policies.
   ```tcl
   $ns duplex-link $n0 $n1 1.5Mb 50ms DropTail
   $ns duplex-link $n0 $n2 10Mb 2ms DropTail
   $ns duplex-link $n0 $n3 10Mb 1ms DropTail
   $ns duplex-link $n0 $n4 10Mb 3ms DropTail
   $ns duplex-link $n0 $n5 10Mb 1ms DropTail
   $ns duplex-link $n0 $n10 10Mb 3ms DropTail
   $ns duplex-link $n0 $n11 10Mb 1ms DropTail
   $ns duplex-link $n1 $n6 10Mb 2ms DropTail
   $ns duplex-link $n1 $n7 10Mb 2ms DropTail
   $ns duplex-link $n1 $n8 10Mb 1ms DropTail
   $ns duplex-link $n1 $n9 10Mb 2ms DropTail
   $ns duplex-link $n1 $n12 10Mb 1ms DropTail
   $ns duplex-link $n1 $n13 10Mb 2ms DropTail
   ```
   - Buffer sizes are set for the links.
   ```tcl
   $ns queue-limit $n0 $n1 25
   $ns queue-limit $n1 $n0 25
   ```
   - Node positions are defined for visualization.
   ```tcl
   $ns duplex-link-op $n0 $n1 orient right
   $ns duplex-link-op $n0 $n2 orient right-up
   $ns duplex-link-op $n0 $n3 orient up
   $ns duplex-link-op $n0 $n4 orient right-down
   $ns duplex-link-op $n0 $n5 orient down-right
   $ns duplex-link-op $n1 $n6 orient right-up
   $ns duplex-link-op $n1 $n7 orient right
   $ns duplex-link-op $n1 $n8 orient right-down
   $ns duplex-link-op $n1 $n9 orient down
   ```

2. **Bottleneck Link Errors**
   - A check is performed to determine if the link should be set to have loss errors.
   ```tcl
   if {$lrate < 0} { set lossy-link 1 }
   if { $lossy-link == 1} {
       set lossy-module [new ErrorModel]
       $lossy-module unit pkt
       $lossy-module set rate $lrate
       $lossy-module ranvar [new RandomVariable/Uniform]
       $lossy-module drop-target [new Agent/Null]
       $ns lossmodel $lossymodule $n0 $n1
   }
   ```

The code sets up a network topology and includes logic to add bottleneck link errors based on a specific rate.
puts " ErrorModel: lossy-link: $lossy-link - rate: $lrate" }

###setup tcp connection #######
for { set j 0 } { $j ¡ 6 } { incr j } {
set tcp$j [ new Agent/TCP/Newreno]
set ftp$j [ new Application/FTP]
set sink$j [ new Agent/TCPSink ] }

########attach tcp with node ############
$ns attach-agent $n2 $tcp0
$ns attach-agent $n3 $tcp1
$ns attach-agent $n4 $tcp2
$ns attach-agent $n5 $tcp3
$ns attach-agent $n12 $tcp4
$ns attach-agent $n13 $tcp5

###########attach sink with node ####
$ns attach-agent $n6 $sink0
$ns attach-agent $n7 $sink1
$ns attach-agent $n8 $sink2
$ns attach-agent $n9 $sink3
$ns attach-agent $n10 $sink4
$ns attach-agent $n11 $sink5

########### connecting tcp wih sink & udp with CBR ########
$ns connect $tcp0 $sink0
$ns connect $tcp1 $sink1
$ns connect $tcp2 $sink2
$ns connect $tcp3 $sink3
$ns connect $tcp4 $sink4
$ns connect $tcp5 $sink5

##################
$ftp0 attach-agent $tcp0
$tcp0 set fid 1
$tcp0 set packetSize 2000
$ftp1 attach-agent $tcp1
$tcp1 set fid- 2
$tcp1 set packetSize- 2000
$ftp2 attach-agent $tcp2
$tcp2 set fid- 3
$tcp2 set packetSize- 2000
$ftp3 attach-agent $tcp3
$tcp3 set packetSize- 2000
$tcp3 set fid- 4
$ftp4 attach-agent $tcp4
$tcp4 set fid- 5
$tcp4 set packetSize- 2000
$ftp5 attach-agent $tcp5
$tcp5 set packetSize- 2000
$tcp5 set fid- 6

################## schedule the event ############
$ns at 1.1 "$ftp0 start"
$ns at 2.15 "$ftp1 start"
To conclude, NS2 is probably a great tool for doing research, especially when this research takes up a longer period of time and many simulations. In such a situation it is worth of digging into the source code and finding out how the internals of the program exactly work. However, for smaller research with non-sufficient knowledge of ns2, there are just too many options and got to know how to do simulations with conclusive results.

Perl language

Perl [90] is a dynamic programming language created by Larry Wall and first released in 1987. Perl borrows features from a variety of other languages including C, shell scripting (sh), AWK, sed and Lisp. Perl was widely adopted for its strengths in text processing and lack of the arbitrary limitations of many scripting languages at the time.

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features include support for multiple programming paradigms (procedural, object-oriented, and functional styles), reference counting memory management (without a cycle detecting garbage collector), built-in support for text processing, and a large collection of third-party modules.

Perl Script
perl Scripts used for extracting interesting information from the trace files generated by the ns-simulations. The following script is an example for perl scripts (called throughput with extension (.pl)). It used to calculate the throughput run with the following scripts: "perl throughput.pl tracefile.tr tcp(the protocol) 0.1(granularity) throughput.data”. It can be applied to a arbitrary ns-tracefile.

**Throughput Script**

```perl
$infile=$ARGV[0];
$protocol=$ARGV[1];
$granularity=$ARGV[2];

# compute how many bytes were transmitted during time interval specified by granularity in ms
$sum=0;
$clock=0;
$count=1;
$cumul=0;
open (DATA, "$infile")
// die "Can't open $infile: $!";
while (<DATA>) {
    @x = split(' ');
    if ($x[1]-$clock >= $granularity) {
        if ($x[0] eq 'r') {
            if ($x[4] eq $protocol) {
                $sum=$sum+$x[5];
            }
        } else {
            $throughput = (($sum*8)/$x[1])/1000;
            $cumul = $cumul + $throughput;
            $cumul-average = $cumul / $count;
            $count ++;
            print STDOUT "$x[1] $throughput $cumul-average
        }
    } else {
        $clock=$clock+$granularity;
    }
}
close DATA;
exit (0);
```

**awk programming**

awk [91] is a simple and elegant pattern scanning and processing language. awk is a programming language that gets its name from the 3 people who invented it (Aho, Weinberger, and Kernighan). Because it was developed on a UNIX operating
system, its name is usually printed in lower-case ("awk") instead of capitalized ("Awk"). awk is distributed as free software.

awk is also the most portable scripting language in existence. It was created in late 70th of the last century. The name was composed from the initial letters of three original authors Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. It is commonly used as a command-line filter in pipes to reformat the output of other commands.

It’s the precursor and the main inspiration of Perl. Although originated in Unix it is available and widely used in Windows environment too. awk takes two inputs: data file and command file. The command file can be absent and necessary commands can be passed as augments. As it could be noted awk is very underappreciated language. GAWK, Gnu’s version of Aho, Weinberger, and Kernighan’s old pattern scanning language isn’t even viewed as a programming language by most people. Like PERL and TCL, most prefer to view it as a ”scripting language.” It has no objects; it is not functional; it does no built-in logic programming. Their surprise turns to puzzlement when I confide that (a) while the students are allowed to use any language they want; (b) with a single exception, the best work consistently results from those working in GAWK. Programmers in C, C++, and LISP haven’t even been close. The main advantage of awk is that unlike Perl and other ”scripting monsters” that it is very slim without feature creep so characteristic of Perl and thus it can be very efficiently used with pipes. Also it has rather simple, clean syntax and like much heavier TCL can be used with C for “dual-language” implementations. Generally Perl might be better for really complex tasks, but this is not always the case. In reality awk much better integrates with Unix shell and until probably in 2004 for simple scripts there was no noticeable difference in speed due to the additional time to load and initialize huge Perl interpreter (but Perl 5 still grows and soon might be too fat even for the typical PC or server). Unfortunately, Larry Wall then decided to throwing in the kitchen sink, and as a side effect sacrificed the simplicity and orthogonally. I would agree that Perl added some nice things, but it probably added too much nice things. Perl4 can probably be used as
AWK++ but it’s not that portable or universally supported. Like I mentioned above, awk is the most portable scripting language in existence.

awk progrm

The script:

BEGIN {
for (i in send) {
  send[i] = 0 }
for (i in recv) {
  recv[i] = 0 }
delay = avg-delay = 0 }
# Trace line format: normal
  event = $1
time = $2
if (event == "+" —— event == ",") node-id = $3
if (event == "r" —— event == "d") node-id = $4
flow-id = $8
pkt-id = $12
# Store packets send time
if (flow-id == flow & node-id == 2 & send[pkt-id] == 0 & (event == "+" ——
  event == "s")) {
  send[pkt-id] = time
  #printf("send[% g] = % g/n",pkt-id,time)
}
# Store packets arrival time
if (flow-id == flow & node-id == 1 & event == "r") {
  recv[pkt-id] = time
  printf("% g % g/n",time, (recv[pkt-id]-send[pkt-id]))
}
END {
# Compute average delay
for (i in recv) {
  if (send[i] == 0) {
    printf("/n Error % g/n",i)
  } else {
    delay += recv[i] - send[i]
    num ++
  }
  printf("% 10g ",flow)
  if (num != 0) {
    avg-delay = delay / num
  } else {
    avg-delay = 0
  }
  printf("% 10g/n",avg-delay*1000)
}

Gnuplot
Gnuplot was originally developed by Colin Kelley and Thomas Williams in 1986 to plot functions and data files on a variety of terminals. gnuplot is a command-driven interactive function plotting program. It can be used to plot functions and data points in both two and three dimensional plots in many different formats. It is designed primarily for the visual display of scientific data. gnuplot is copyrighted, but freely distributable; you don’t have to pay for it. Gnuplot can read in files containing additional commands during an interactive session, or it can be run in batch mode by piping a pre-existing file or a stream of commands to stdin. Gnuplot is used as a back-end graphics driver by such higher-level mathematical packages as Octave, and can easily be wrapped in a cgi script for use as a web-driven plot generator. Gnuplot offer:

- Plotting two-dimensional functions and data points in many different styles (points, lines, error bars)
- Plotting three-dimensional data points and surfaces in many different styles (contour plot, mesh)
- Interactive command line editing and history (most platforms)
- TEX-like text formatting for labels, titles, axes, data points
- Support for a large number of operating systems, graphics file formats and output devices

Although there are some subtle differences, any command-line arguments are assumed to be names of files containing gnuplot commands, with the exception of standard X11 arguments, which are processed first. Each file is loaded with the load command, in the order specified. Gnuplot exits after the last file is processed. When no load files are named, gnuplot enters into an interactive mode. The special filename "-" is used to denote standard input. See help for batch/interactive for more details. Many gnuplot commands have multiple options. Version 4 is less sensitive to the order of these options than earlier versions, but some order-dependence remains. If you see error messages about unrecognized options, please try again using the exact order listed in the documentation. Commands may extend
over several input lines by ending each line but the last with a backslash. The
backslash must be the last character on each line. The effect is as if the backslash
and newline were not there. That is, no white space is implied, nor is a comment
terminated. Therefore, commenting out a continued line comments out the entire
command. But note that if an error occurs somewhere on a multi-line command, the
parser may not be able to locate precisely where the error is and in that case will not
necessarily point to the correct line. In this document, curly braces ({})) denote
optional arguments and a vertical bar, separates mutually exclusive choices.

gnuplot keywords or help topics are indicated by back quotes or boldface (where
available). Angle brackets (<>) are used to mark replaceable tokens. In many cases,
a default value of the token will be taken for optional arguments if the token is
omitted, but these cases are not always denoted with braces around the angle
brackets.
Appendix B

Topology Generator

Effective engineering of the Internet is predicated upon a detailed understanding of issues such as the large-scale structure of its underlying physical topology, the manner in which it evolves over time, and the way in which its constituent components contribute to its overall function. Unfortunately, developing a deep understanding of these issues has proven to be a challenging task, since it in turn involves solving difficult problems such as mapping the actual topology, characterizing it, and developing models that capture its emergent behavior. Consequently, even though there are a number of topology models, it is an open question as to how representative the topologies they generate are of the actual Internet. The idea of using Topology Generator is to have a topology generation framework which improves the state of the art and is based on design principles which include representative-ness, inclusiveness, and interoperability. Representative-ness leads to synthetic topologies that accurately reflect many aspects of the actual Internet topology (e.g. hierarchical structure, degree distribution, etc.). Inclusiveness combines the strengths of as many generation models as possible in a single generation tool. Interoperability provides interfaces to widely-used simulation applications such as Network Simulator NS2, SSF and OmNet++ as well as visualization applications. Such a tool is called a universal topology generator [92]. Most simulations from literature in the field have used representations of a real topology (e.g. the Arpanet or a well-known provider’s backbone), simple models (e.g. a ring or a star), or randomly-generated flat
topologies using a variation of edge-probability function. Recently, more complex randomly-generated hierarchical models have been used to better approximate the Internet’s hierarchical structure. The following section will represent a real topology generator GT-ITM.

**GT-ITM**

A topology generator creates a graph representation of a network based on a model and a set of parameters specified by the user. These representations are often used as input to a simulation. If the graph approximates characteristics of a real network, then the simulation results predict the behavior of a network protocol or application if it were to be deployed. GT-ITM, Georgia Tech Internetwork Topology Models, is a collection of routines to generate and analyze graphs using a wide variety of models for internetwork topology. The graphs are generated in Don Knuth’s SGB format. A routine is provided to convert to an alternative format that may be easier to parse.

**GT-ITM Overview**

GT-ITM is a widely used facility for creation and analysis of graph models of network topology. Such models are widely used in simulation based evaluation and comparison of new protocols and algorithms, and also as a tool to understand the topological characteristics of the Internet. A new release of GT-ITM have been developed, which includes a number of new features and enhancements, including: a visualization capability; support for routing/forwarding in very large graphs (up to hundreds of thousands of nodes), and the ability to specify and use inter-domain routing policies on a per domain basis. The next section presents a brief overview of GT-ITM, followed by a section describing each of the major enhancements.
Models

The GT-ITM topology generator can be used to create flat random graphs and two types of hierarchical graphs, the N-level and transit-stub.

- **Flat Random**

  Methodology: Randomly place nodes at locations in a plane, then connect nodes with edges according to the probability given by the edge mode.

  Parameters: Number of nodes, scale, edge method, alpha, beta, gamma. See a comment from the code for an explanation of how each method uses alpha, beta, and gamma.

- **N-Level Hierarchical**

  Methodology: Start with the top level and create a flat random graph. Recursively replace each node in current level with a connected flat random graph. Resolve edges between levels by selecting nodes at random from within the replacement graph.

  Parameters: Number of levels, for each level, a set of parameters describing the graph.

- **Transit-Stub Hierarchical**

  Methodology: Create a 2-level hierarchy, with each top-level node representing a transit domain, and with the second level giving each intra-domain graph. Next, attach stub domains to each transit domain. Finally add extra transit-stub and stub-stub edges.

  Parameters: Number of stubs per transit, Number of transit-stub edges, Number of stub-stub edges, Top-level, transit-domain, and stub domain parameters.

Generating Topology
GT-ITM has been used by many researchers in comparisons of routing, caching, peer-to-peer and other algorithms and services. It has been included as part of the "ns2" network simulator package for several years. For example, if it is wanted to create a transit-stub graph with 200 nodes. So the specification file create, say ts200, that goes like this:

```plaintext
## Comments : 
## [method keyword] [number of graphs] [initial seed] 
## [stubs/exit] [t-s edges] [s-s edges] 
## [n] [scale] [edge-method] [alpha] [beta] [gamma] 
## number of nodes = 1*8* (1 + 4*6) = 200 

10 47 
4 0 0 
1 20 3 1.0 
8 20 3 0.8 
6 10 3 0.5
```

On running 'itm ts200', 10 transit-stub graphs with 200 nodes each will be created starting with initial seed 47. Line 4 0 0 means each transit node will have 4 stub domains connected to it. There shall be no extra transit-stub edges and no extra stub-stub edges. Line 1 20 3 1.0 means there will 1 transit domain in each graph. Line 8 20 3 0.8 says each transit domain will have, on an average, 8 transit nodes with an edge between each pair of node with a probability of 0.8 and line 6 10 3 0.42 says every stub domain shall have, on an average, 6 nodes each with an edge between every pair with a probability of 0.5. For a more complete description of the parameter specification file see models.ps file under doc’s subdirectory. The graphs produced are named as ts200-[0 to 9].gb and are in Stanford Graph Base Output format. This needs to be converted to tcl format for ns-2 to interpret it correctly.

**Conversion of GT-ITM output to ns-2 format**

Download the sgb2ns conversion program. Follow instructions in the README file and create the sgb2ns executable. Note that sgb2ns should be expanded in the GT-ITM directory, and its executables will be placed in gt-itm/bin subdirectory. This distribution also contains ts2ns and sgb2hier as described below. Modify sgb2ns’s Makefile to make macro GT-ITM to point to the right path and correct
Stanford Graphics Base library name installed in your system (e.g libgd.a, libgb4.a or libgb5.a etc). The library also comes as a part of gt-itm distribution. Under any circumstance it should reside under gt-itm/lib. Sgb2ns converts the sgb output to tcl output. Usage: sgb2ns ts200-0.gb ts200.tcl a modification to sgb2ns contributed separates transit nodes from stub nodes when using transit-stub networks. The file (ts2ns.c) is included in sgb2ns distribution.

Source distribution

Over the next few years, important decisions will be made regarding the adoption of algorithms and placement of facilities in the Internet to support scaling to tens of thousands of administrative domains. The inputs to these decisions will include simulation and analyses based on models of networks and applications. Unfortunately, with the current state of the art it is very difficult to draw quantitative conclusions based upon such models; indeed, there is presently no theoretical basis for assessment of the accuracy of conclusions drawn from models. GT-ITM supports large internet-works through scalable, realistic models of internet-work structure and applications. An additional applying and demonstration the utility of our modeling the development of novel multicast routing algorithms. Multicast routing is a critical and difficult problem within large scale internetworking.

To conclude, GT-ITM is a collection of software tools for creation, manipulation, and analysis of graph models of internet topology. It has been used by networking researchers in a variety of ways, most often to create topologies for use in simulation studies. This paper describes the features of a new release of GT-ITM, including enhanced visualization capabilities, a routing and forwarding module for use with large graphs, and support for modeling of inter-domain routing policies.
References


References


References


[72] GT-ITM "Georgia Tech Internetwork Topology”, http://www.cc.gatech.edu/project/gtitm

[73] BRITE "Boston university Representative Internet Topology Generator”, http://www.cs.bu.edu/brite


[77] Haowei Bai, David Lilja and Mohammed Atiquzzaman,"Improving TCP Throughput over Lossy Links Using Protocol-Level Speculations ", 2005


[79] J`orgen Ols´, " On Packet Loss Rates used for TCP Network Modeling " Department of Mathematics, Uppsala University, 2004
تحسين آلية للتحكم في ازدحام شبكات الحاسب

رسالة مقدمة طبقاً لمتطلبات جامعة المنوفية للحصول على درجة الماجستير في الهندسة الالكترونية تخصص هندسة وعلوم الحاسبات

المهندسة/ شيماء إبراهيم مصطفى حاج
مهندسة شبكات بشركة كهرباء جنوب الدلتا

تحية إلهام

أ.د. نوال احمد الفيشاوي
أستاذ ورئيس قسم هندسة وعلوم الحاسبات كلية الهندسة الالكترونية جامعة المنوفية

أ.م.د. أيمن السيد أحمد السيد عبيره
أستاذ مساعد بقسم هندسة وعلوم الحاسبات كلية الهندسة الالكترونية جامعة المنوفية

2012