

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FACULTY OF TECHNOLOGY

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

ANALYZING AND PROPOSING A SOLUTION FOR THE INCOMPATIBILITY PROBLEM OF TCP VEGAS

BY

KERIA WASSIE

A thesis submitted to the school of Graduate studies of Addis Ababa University in partial fulfillment of the requirements for the degree of Masters of Science in Electrical and Computer Engineering

(Computer Engineering)

March, 2008

Addis Ababa, Ethiopia



ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY
ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

ANALYZING AND PROPOSING A SOLUTION FOR THE INCOMPATIBILITY PROBLEM OF TCP VEGAS

By
KERIA WASSIE

ADVISORS

Ato Abyot Asalefew
Dr. V.N.V Manoj



ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY
ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

ANALYZING AND PROPOSING A SOLUTION FOR THE INCOMPATIBILITY PROBLEM OF TCP VEGAS

By

KERIA WASSIE

APPROVAL BY BOARD OF EXAMINERS

Dr. Mengesha Mamo Chairman, Department of Electrical and Computer Engineering	_____ Signature
Ato Abyot Asalefew Advisor	_____ Signature
Dr. V.N.V Manoj Advisor	_____ Signature
Prof. Sayed External Examiner	_____ Signature
Dr. Hailu Ayele Internal Examiner	_____ Signature

DECLARATION

I, the undersigned, hereby declare that this thesis is my original work performed under the supervision of Dr. V.N.V Manoj and Ato Abyot Asalefew has not been presented as a thesis for a degree program in any other university and all sources of materials used for the thesis are duly acknowledged.

Name: Keria Wassie Mussa .

Signature: _____

Place: Addis Ababa .

Date of submission: _____

This thesis has been submitted for examination with our approval as university advisors

Ato Abyot Asalefew _____
Advisor Signature

Dr. V.N.V Manoj _____
Advisor Signature

TABLE OF CONTENTS **PAGE**

LIST OF FIGURES iii
LIST OF TABLES iv
GLOSSARY OF ACRONYMS v
ACKNOWLEDGMENTS vi
ABSTRACT
.....vii

CHAPTER 1: Introduction 1
1.1 Statement of the Problem 3
1.2 Objectives 4
1.2.1 General Objective: 4
1.2.2 Specific Objectives 4
1.3 Related Words 5
1.4 Methodology
..... 6
1.5 Thesis Organization 7

CHAPTER 2: Introduction to TCP and Congestion Control Mechanisms 8
2.1 Transmission Control Protocol (TCP) 8
2.1.1 Characteristics of TCP 9
2.1.2 Overview of TCP Congestion a nd Control Methods 10
2.1.3 Congestion Control Algorithms 11
2.1.3.1 Slow Start and Congestion Avoidance 12
2.1.3.2 Fast Retransmit and Fast Recovery 13
2.2 Congestion Control Mechanisms of TCP 14
2.2.1 TCP Tahoe 14
2.2.2 TCP Reno 15
2.2.3 TCP Vegas 17
2.2.4 Comparison between TCP Reno and TCP Vegas ……..... 20
2.2.5 Problems of TCP Vegas 21

CHAPTER 3: Relaxed-Vegas algorithm and its Implementation 22
3.1 Design Phase 22
3.1.1 Relaxed-Vegas Algorithm 22
3.1.1.1 Existing Algorithm of TCP Vegas 23
3.1.1.2 Algorithm of Relaxed-Vegas 24
3.1.2 Design of Simulation Structure ……..... 28
3.1.3 Design of Network Topology 30
3.1.4 Simulation 31
3.1.4.1 Network Simulator Version 2 (NS2) 31
3.2 Implementation Phase 32
3.2.1 Network Animator (NAM) 34
3.2.2 Implementation of Trace Files 36
3.2.3 Data Filtering using "awk" 37

CHAPTER 4: Simulation Results 39
4.1 Performance Metrics 39
4.2 Simulation Results while TCP-Reno and TCP-Vegas working independently 40
4.2.1 Number of Received and Dropped Packets 40
4.2.2 Drops of TCP Reno and TCP Vegas 41
4.2.3 Stability of Window Management 42
4.2.4 Queue Management of TCP Reno and TCP Vegas 42
4.2.5 Throughput Comparison of TCP Reno and TCP Vegas 43
4.2.6 Overhead Cost 44
4.3 Simulation Results while TCP-Reno and TCP-Vegas working simultaneously 45
4.3.1 Simulation Results using the standard threshold values 46
4.3.1.1 Received Packets of Reno and Vegas 46
4.3.1.2 Dropped Packets from Reno and Vegas 47
4.3.1.3 Throughput Comparison of TCP Reno and TCP Vegas 48
4.3.1.4 Overhead Cost 49
4.4 Implementation and Simulation Results of Relaxed-Vegas 50
4.4.1 The Effect of Changing of alpha, Beta with constant Gamma 50
4.4.2 The Effect of Changing of Gamma 53
4.4.3 Throughput of Relaxed-Vegas 55
4.4.4 Dropped Packets of Relaxed-Vegas 56
4.5 Comparison of TCP Vegas and Relaxd-Vegas 57

CHAPTER 5: Analysis and Discussion of Results 58
5.1 Discussion of Results while TCP-Reno and TCP-Vegas working Independently 58
5.2 Discussion of Results while TCP-Reno and TCP-Vegas working Simultaneously 59
5.3 Discussion of Results of Relaxed-Vegas 60

CHAPTER 6:Conclusions and Recommendations 62
6.1 Conclusions 62
6.2 Recommendations 64

Appendix A: Awk Scripts for Metrics Calculations 65
Appendix B: Simulation TCL Script 68

REFERENCES: 71

List of Figures

Figure - 2.1 Three way handshake ---------- 9

Figure - 2.2 Flow chart of Congestion control of TCP Reno and TCP Vegas ------- 19

Figure - 3.1 Flow chart For Relaxed-Vegas Algorithm ---------- 27

Figure - 3.1 Block diagram of the over-all progress of the Simulation ---------- 28

Figure - 3.2 Topology layout used for Simulation ---------- 30

Figure - 3.3 Format of the trace file ---------- 36

Figure - 4.1 Packet drop comparison between Reno and Vegas ---------- 41

Figure - 4.2 Window management comparisons between Reno and Vegas --------- 42

Figure - 4.3 Queue management comparisons between TCP Reno and Vegas ----- 42

Figure - 4.4 Throughput comparisons of Reno and Vegas ---------- 44

Figure - 4.5 Screen captured pictures of Network Animator (NAM) ---------- 45

Figure - 4.6 Received packets using the Standard Threshold Values ---------- 46

Figure - 4.7 Dropped packets using the Standard Threshold Values ---------- 47

Figure - 4.8 Throughputs of TCP Reno and TCP Vegas
while working simultaneously ---------- 48

Figure - 4.9 Overhead cost while working simultaneously ---------- 49

Figure - 4.10 Received packets with constant gamma (G=1) ---------- 52

Figure - 4.11 Received packets with different values of Gamma ---------- 54

Figure - 4.12 Throughput with change of Gamma ---------- 55

Figure - 4.12 Throughput comparison of TCP Vegas and Relaxed Vegas ---------- 58

Figure 4.13 Throughput comparison of TCP Vegas and Relaxed Vegas ---------- 57

Figure 4.14 Throughput comparison of TCP Vegas and Relaxed Vegas ---------- 57

List of Tables

Table - 4.1 Received and dropped packets when working independently ---------- 40

Table - 4.2 Throughputs of Reno and Vegas while working independently -------- 43

Table - 4.3 Received packets with standard threshold values (α=1, β=3 and G=1) ---------- 46

Table - 4.4 Dropped packets with standard threshold values (α=1, β=3 and G=1) ---------- 47

Table - 4.5 Throughputs of TCP Reno and TCP Vegas while working simultaneously ---------- 48

Table - 4.6 Overhead cost of TCP Reno and TCP Vegas while working simultaneously ---------- 49

Table - 4.7 Simulation results with α=β=2, G=1 ---------- 50

Table - 4.8 Simulation results with α=β=3, G=1 ---------- 51

Table - 4.9 Simulation results with α=β=4, G=1 ---------- 51

Table - 4.10 Simulation results with α=β=5, G=1 ---------- 51

Table - 4.11 Simulation results with α=β=6, G=1 ---------- 51

Table - 4.12 Average Values of the Received Packets with α=β=6, G=1 ---------- 52

Table - 4.13 Simulation results with the change of Gamma ---------- 53

Table - 4.14 Throughput results with change of Gamma ---------- 55

Table - 4.15 Drops of Relaxed-Vegas ---------- 56

Table - 4.16 Comparison of Drops of TCP Vegas and Relaxed-Vegas ---------- 57

Glossary of Acronyms

ACK	Acknowledgment
AD	Additive Decrease
AI	Additive Increase
AIAD	Additive Increase Additive Decrease
AVI	Audio Video Interleave
FTP	File Transfer Protocol
GIF	Graphics Interchange Format
IP	Internet Protocol
NS2	Network Simulator Version 2
OSI	Open Systems Interconnection
RED	Random Early Detection
RTT	Round Trip Time
SMSS	Sender Maximum Segment Size
SMTP	Simple Mail Transfer protocol
TCP	Transmission Control Protocol

Acknowledgments

First and foremost, I express my deepest gratitude to my advisors Ato Abyot Asalefew and Dr. V.N.V Manoj for their unceasing support and suggestions that greatly enriched this thesis work.

Finally, my heartfelt appreciation and indebtedness goes to my family and my friends for their love, patience, support and encouragement throughout my study.

Abstract

TCP is a reliable, connection-oriented and sequence-delivery protocol. All applications which require a reliable delivery of data use TCP. TCP-Reno and TCP-Vegas are some of the TCP types created to give a reliable service in the Network.

According to different researchers and results of our simulation, it is observed that TCP-Vegas in isolation has better performance with respect to overall network utilization, stability, fairness, throughput and packet loss, but its performance degrades when TCP-Reno connection exists in the Network. So this thesis is trying to address this incompatibility problem of TCP Vegas while working with TCP-Reno and proposing a solution called **Relaxed-Vegas**.

Using a simulation tool (NS2) we examine the detail behaviours of TCP-Reno and TCP-Vegas while working independently and simultaneously. We propose modifications to the currently working algorithm of TCP-Vegas.

From the simulation results we compare the performance of the currently used TCP Vegas and the new proposed solution (Relaxed-Vegas). By using Relaxed-Vegas the number of received packets and the throughput (good-put) is raised by **56.87%** and the drop of packets is reduced by **17.0%.**

Keywords: TCP, TCP Reno, TCP Vegas, Congestion control, Relaxed-Vegas

Chapter 1

Introduction

Internet is different from a single network. Because many parts in the world have different topologies, bandwidths, delays, packet sizes, and other parameters.

TCP is a connection-oriented packet transfer protocol that ensures reliable and in order communication between two hosts [7]. The idea behind this reliability and ordered packet delivery is that the sender does not send a packet unless it has received acknowledgment from the receiver that the previous packet (or group of packets) sent has been received [2, 15]. Due to these good qualities of TCP, most networks approximately 90% of current traffic uses this transport service. It is used by such applications as telnet, World Wide Web (www), ftp, email [19, 20].

The Internet's expanding rate, as well as the widespread use of TCP by the majority of the network applications, leads to the need of improving the TCP's congestion detection and avoidance mechanisms. Starting from the series congestion collapse on October 1986, many researchers developed and implemented different versions of TCP such as TCP Tahoe and TCP Reno [2]. Short description for the history of the development of TCP types is displayed in Chapter 2.

Up to now several researches have been made by the scientific community, which study the behavior of the congestion control algorithms, detect problems and suggest improvements. One of them is TCP Vegas. TCP Vegas is a new TCP implementation, by Brakmo, O'Malley and Petersen which presented in 1994 [10], has a different mechanism of congestion control. It uses the difference between "***expected***" and "***actual***" flow rates to estimate the available bandwidth in the network, and using this

difference in flow rates, estimates the congestion level in the network and updates the congestion window size accordingly.

Unlike to TCP Reno instead of waiting till overflow of the buffer and packet losses happened, TCP Vegas uses a prediction mechanism to forecast the congestion in the network before it happens. Different researches assure that TCP Vegas achieves higher efficiency than TCP Reno, and causes less packet retransmissions. TCP Vegas also improves the throughput of TCP Reno by 37 to 71% [5, 9].

But when TCP Vegas is working with TCP Reno connections, its performance degrades due to the different mechanisms of congestion control they follow [5]. That is when TCP Vegas competes with TCP Reno there is unfair share of a bandwidth. TCP Reno increases continuously its window size till the loss of the packets occurs but TCP Vegas has a conservative nature; it tries to use small space and tries to leave more space in the buffer as mush as possible.

In this thesis the mechanisms used by TCP Reno and TCP Vegas, to control congestion on the network, are studied and analyzed using a Network Simulation (NS2). Also we study the behaviors of each TCP types when working independently and simultaneously. From the simulation results we observe that TCP Vegas has good qualities while working alone but, penalized while working with TCP Reno.

So we propose a new solution called Relaxed-Vegas to overcome the incompatibility problem of TCP Vegas while it is working with TCP Reno.

1.1 Statement of the Problem

From different research papers it is clearly seen that TCP Vegas achieves better performance than all other TCP types [9]. But even though TCP Vegas has good performance in isolation, researchers observed that when TCP Vegas is working with other connections that use TCP Reno, it does not receive a fair share of bandwidth [5].

The reason behind this is that both TCP types follow different mechanisms to control congestion. TCP Reno has an aggressive nature; it always tries to fill the buffer until a packet loss is observed. Unlike to Reno, TCP Vegas has a conservative nature, and it tries to use a small buffer size and tries to leave much space in the queue. Due to this opposite behavior incompatibility problem occurs.

Some researchers agree that this problem could be one of the reasons why TCP Vegas is not widely deployed despite its many desirable properties. [5]

So this thesis focuses on analyzing and proposing a solution to this incompatibility problem of TCP Vegas while working with other TCP Reno.

1.2 Objectives

1.2.1 General Objective:

- The main goal of this thesis is to investigate the current algorithms used in TCP Vegas and study why the incompatibility problem is happened while working with other TCP types and then propose a solution to the incompatibility problem.

1.2.2 Specific Objectives:

- Investigate the existing TCP Vegas's algorithm.
- Analyze and evaluate the loopholes in TCP Vegas congestion control algorithms.
- Study the incompatibility between TCP Reno and TCP Vegas
- Address appropriate evaluation metrics
- Design new algorithm to solve the incompatibility problem.
- Analyze the new algorithm using Simulation.

1.3 Related Works

The incompatibility of TCP Vegas while working with TCP Reno was studied and mentioned by different researchers and some of the papers also proposed solutions to this problem.

- The incompatibility problem of TCP Vegas is explained in [5], when a TCP Vegas user competes with other users that use TCP Reno, it does not receive a fair share of bandwidth due to the conservative congestion avoidance mechanism used by TCP Vegas. Different researchers conclude that it is possible to deploy TCP Vegas in connection with TCP Reno by adopting gateway control such as Random Early Detection (RED) gateways.

- The research papers [6, 7], also studied the unfairness problem of TCP Vegas when competing with TCP Reno. According to these papers the study is done for algorithms: Drop-tail and Random Early Detection (RED) algorithms and from their research they concluded that RED improves the fairness to some degree, but drop-tail can't keep the fairness between TCP Reno and TCP Vegas.

- Similarly in paper [3] the fairness of TCP Vegas while working with TCP Reno is studied. The paper proposed two approaches to improve the fairness. In the first solution a modified algorithm of TCP Vegas is developed by applying two modes for updating the window size. It behaves as TCP Vegas in the Moderate Mode and as TCP Reno in the Aggressive Mode. In the second solution a modified RED algorithm at the router is developed, and called as ZL-RED (Zombie Listed RED).

- Other researchers [14] did some studies to observe the behavior of fairness of TCP Vegas when the minimum threshold value (alpha) and the maximum threshold value (beta) varies. They studied the behavior of TCP Vegas

related with other TCP Reno and decided that the values of alpha and beta should be equal and not greater than 2.

- Different from that the standard TCP Vegas is working with alpha less than beta by two, i.e. α=1, and β=3. In all those previous Drop-Tail queue methods; the effect of the change of Gamma (the threshold value which decides to change from slow-start to congestion avoidance state), was not taken into account.

So the incompatibility problem needs a great focus and enhancement in order TCP Vegas to be implemented in the current Internet.

1.4 Methodology

- **Literature Review:** A lot of works has been done on TCP congestion avoidance. The thesis work requires a review of the available documents and literatures to find out the existing TCP congestion control algorithms and find out the problems associated with these algorithms.

- **Design of Algorithm:** After studying the current algorithms of TCP Vegas a new algorithm is proposed to solve the specific problem of incompatibility.

- **Design of Network Topology:** The network topology including the nodes, links and the set of protocol objects (agents), are designed.

- **Design of Simulation program and Simulation:** Network Simulator (NS2) is used to study the characteristics of the proposed algorithm. NS2 is widely available and commonly used network simulation software which can help to analyze and predict actual network characteristics without investing any cost.

- **Analysis of Simulation Results:** Performance metrics are used to show and compare the results of the existing congestion algorithms and the new proposed solution.

- **Conclusion and Recommendations:** The summery of the whole thesis work and observed results will be written on the conclusion.

1.5 Thesis Organization

The thesis is organized as follows: Chapter two presents introduction to TCP and Congestion control mechanisms, literature survey related to TCP Vegas, and detailed description to congestion control algorithms that are used in TCP Reno and TCP Vegas. The proposed algorithm (Relaxed-Vegas) and implementation are presented in chapter three. Chapter four presents the simulation results of the two TCP types while working independently and simultaneously, and the results of the proposed algorithm. Analysis and discussions of the results are described in chapter 5. Conclusion and recommendations are presented in chapter 6.

Chapter 2

Introduction to TCP and Congestion Control Mechanisms

2.1 Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP) is a packet transfer protocol which is commonly used on the Internet today. TCP is one of the main protocols, operating at Layer 4 (Transport Layer) of the OSI reference model, designed for the Internet. TCP is a **connection oriented** and specially designed to provide a **reliable** end-to-end connection between two hosts. It is also a **secure** and sequential delivery of packets. Due to that 90% of the total traffic on the Internet today is using this protocol. TCP is used by those applications needing reliable, connection-oriented and secure transport service: such as mail (SMTP), file transfer (FTP), and virtual terminal service (Telnet) [19, 20].

TCP also provides a flow control mechanism to prevent overflowing of the buffers at the receiving host, and allows the sending and receiving host to communicate. The two hosts then establish a data-transfer rate that is agreeable to both. The flow control mechanism implemented in TCP is window based control mechanism. Window based protocol refers to the fact that the window size is negotiated dynamically during the TCP session. Hence, windowing is a flow-control mechanism. Windowing requires that the source device receives an acknowledgment from the destination after transmitting a certain amount of data. The receiving TCP process reports a “window” to the sending TCP. This window specifies the number of packets, starting with the acknowledgment number, that the receiving TCP process is currently prepared to receive.

2.1.1 Characteristics of TCP

- **TCP is connection-oriented** means that a connection must be established before hosts can exchange data. When the request for set up a connection from a sender side is accepted by the receiver, the authorization of the session should be verified by exchanging messages between the two hosts. This is called Three-way handshake as can be seen in figure 2.1.

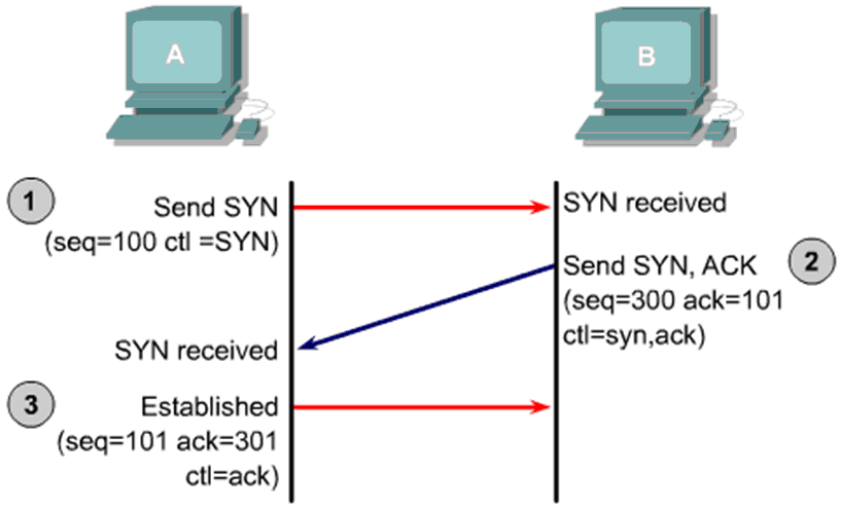


Figure2.1: Three way handshake

- **TCP provides a reliable delivery service** Reliability is achieved by assigning a sequence number to each transmitted segment. An acknowledgment is used to verify that the segment is received by the other host. For each segment or group of segments sent, the receiving host must return an acknowledgment (ACK) within a specified period of time. If an ACK is not received for the transferred, the source host retransmits the data again. [11]
- **All TCP connections are full duplex.** This means that a TCP connection supports the simultaneous transfer of data in both directions. [2]

- **TCP is a Sequence Delivery service.** When the source sends two messages along a connection, one after the other, and these two packets arrived out of order, TCP has a method for rearranging the segments on their correct order [11]

2.1.2 Overview of TCP Congestion and Control Methods

Congestion can occur when data arrives on a big pipe (a fast LAN) and ***gets sent*** out a smaller pipe (a slower WAN). Congestion can also occur when multiple input streams arrive at a router whose output capacity is less than the sum of the inputs.

Because of the fast increasing and uncontrolled use of applications on Internet, a serious congestion collapse was happened in October of 1986. During this period, the Internet had the first of what became a series of 'congestion collapses' and the data throughput was dropped from 32Kbps to 40bps [15].

After the collapse of the Internet many researchers suggested that a special method is needed to control and prevent congestion on the network.

On the late 1980's and early 1990, different versions of congestion control mechanisms of TCP are developed and released. The first congestion control mechanism, which is named TCP Tahoe, developed by Jacobson and Karels [5]. Starting from that, different versions of TCP with many modifications are implemented and used in the present Internet. The history of the development of TCP congestion control mechanisms are summarized in [2] as follows:

- TCP Tahoe, the first implementation in 4.3 BSD Tahoe TCP in 1988, initiated support for three of the key algorithms: ***slow-start, congestion avoidance,*** and ***fast retransmit***. These algorithms are originally proposed by Van Jacobson in 1988.

- TCP Reno, the first implementation in 4.3 BSD Reno TCP in 1990, supports all the van Jacobson enhancements introduced in TCP Tahoe and extends the fast retransmit algorithm to support fast recovery. By supporting fast recovery, TCP Reno overcomes the throughput performance limitation of TCP Tahoe that occurs when a single packet is lost from a window of data.
- TCP Vegas, discussed in a number of research papers in 1994, enhances the congestion avoidance algorithm of TCP Tahoe and TCP Reno by dynamically increasing and decreasing the transmission window size according to the observed Round Trip Time (RTT) of the packets that it has previously sent. If the observed RTT becomes large, the network is experiencing congestion, causing TCP Vegas to reduce its window size. Likewise, if the observed RTT becomes small, the network is not experiencing congestion causing TCP Vegas to increase its window size. Another modification introduced by TCP Vegas is that during slow-start the rate of *cwnd* increase is half that of the TCP Tahoe and TCP Reno - *cwnd* is doubled with the receipt of every other ACK instead of every ACK.

2.1.3 Congestion Control Algorithms

TCP uses four congestion control algorithms to manage the congestion window size "*cwnd*" and hence, control congestion in the network:

- Slow-start,
- Congestion avoidance,
- Fast retransmission, and
- Fast recovery

2.1.3.1 Slow Start and Congestion Avoidance

When a TCP connection is first established, the TCP source does not transmit a full receiver's advertised window (***rwnd***) of segments. Instead, the TCP source avoids exceeding the capacity of the network by transmitting only a few packets at the beginning, waiting for the ACKs to those sent packets, and then gradually increasing its transmission rate [2]. This allows the source TCP to probe the network to determine the amount of bandwidth that is currently available for the connection. So at beginning of each new connection, to keep the network from flooding of packets, TCP uses a mechanism called ***slow-start***. Slow start mechanism is also used; when an existing TCP connection is restarted after a long idle period, and wh en an existing TCP connection is restarted after the retransmission timer expires.

TCP uses another mechanism called ***congestion-avoidance*** when it realizes the network is in congestion. To control the amount of outstanding packets which are injected in to the network, the ***slow start*** and the ***congestion avoidance*** algorithms are used in combination, and this is implemented by stating three variables as follows:

(a) ***cwnd***: The congestion window (***cwnd***) is a variable stated at the sender side to control the amount of data to be transmitted into the network before the acknowledgement (ACK) is received.

(b) ***rwnd***: The receiver's advertised window (***rwnd***) is the receiver-side limit which is used to control the amount of the outstanding data to be received.

- ❖ The minimum of ***cwnd*** and ***rwnd*** governs the data transmission.

(c) ***ssth***: The slow start threshold (***ssth***), is a variable used to determine whether the slow start or the congestion avoidance algorithm is to be used.

The slow-start and congestion-avoidance algorithms in combination operate as follows:

- During slow start, a TCP sender increments its ***cwnd*** by at most one Sender Maximum Segment Size (SMSS) [11] bytes for each received ACK that acknowledges the previous data.
- The TCP sender never sends more than the minimum of ***cwnd*** and ***rwnd***.
- When a TCP sender detects segment loss using retransmission timer, the value of ***ssth*** is halved, and the ***cwnd*** is reset to 1 segment.
- The slow start ends when ***cwnd*** exceeds ***ssth***, or when congestion is observed, and congestion avoidance takes place.

In general this TCP response function to congestion is a direct consequence of TCP's *Additive Increase and Multiplicative Decrease* **(AIMD)** mechanisms [11] of increasing the congestion window by roughly one segment per round-trip time in the absence of congestion, and halving the congestion window in response to a round-trip time with a congestion event.

2.1.3.2 Fast Retransmit and Fast Recovery

A TCP-receiver immediately sends a duplicate ACK when an out-of-order segment arrives. The TCP-sender uses fast retransmit to detect and repair loss based on incoming duplicate ACKs.

The fast retransmit and fast recovery algorithms are implemented together in [11] as follows.

- When the third duplicate ACK is received, set **ssth** to half of the cwnd and **cwnd** to **ssth**.
- The lost segment must be retransmitted and set cwnd to ssth+3*SMSS. This inflates the congestion window.

- For each additional dupACK received (after the third), cwnd is incremented by SMSS.
- Transmit a segment, if allowed by the new value of cwnd and rwnd.
- When next ACK arrives that acknowledges new data, set cwnd to ssth. This is deflating of the window.

2.2 Congestion Control Mechanisms of TCP

Starting from the Internet-collapse period different versions of TCP congestion control mechanisms are released and implemented. The common ones are:

1. **TCP Tahoe** (Slow start, congestion avoidance and Fast Retransmit),
2. **TCP Reno** (TCP Tahoe and Fast Recovery), and
3. **TCP Vegas** (More sophisticated bandwidth estimation scheme)

2.2.1 TCP Tahoe

TCP Tahoe is released in 1988 by Jacobson after modifying and adding a number of new algorithms to the previous version of TCP. The new algorithms include: The *Slow-Start, Congestion Avoidance, and Fast Retransmit.*

Tahoe [17] assumed that congestion signals are represented by lost segments. It was assumed by Jacobson that losses due to packet corruption are much less probable than losses due to buffer overflows on the network. Therefore, on a loss, the sender should lower its share of the bandwidth. This is done by reducing its cwnd to half of the size at which the loss was found.

The reasoning behind this value of a half is that the decrease in throughput should be equal to the multiplicative increase of queue length in the network upon congestion.

The implementation of this multiplicative decrease is through the use of a tcp variable called ***ssth***. Upon a loss, half of the value of ***cwnd*** just before the loss is recorded in ***ssth*** (eq-1). The connection then resorts back to slow start by setting cwnd to 2 segments (eq-2). Slow start grows the ***cwnd*** exponentially until it reaches ***ssth*** from which it will do congestion-avoidance until the same thing happens again.

$$\textbf{\textit{ssth}} = \frac{cwnd}{2} \qquad \textbf{(1)}$$

$$\textbf{\textit{cwnd}} = 2 * \text{SMSS} \qquad \textbf{(2)}$$

This forces TCP to enter slow-start again. TCP Tahoe does not deal well with multiple packet drops within a single window of data [17].

2.2.2 TCP Reno

TCP Reno [17], introduced major improvements over Tahoe by changing the way in which it reacts to detecting a loss through duplicate acknowledgements. 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 - This would be an indication of heavy congestion.

The Reno TCP implementation retained the enhancements incorporated into Tahoe TCP but modified the Fast Retransmit operation to include a new algorithm known as **Fast Recovery** [16] which should be activated after a Fast Retransmit. The new algorithm prevents the communication channel from going empty after Fast Retransmit, thereby avoiding the need fall to Slow-Start to re-fill it after a single packet loss. This is the case when a loss is signaled by the receipt of dupacks rather than a timeout. The congestion experienced is not heavy, so the sender can keep on sending since the flow

still exists (as the equilibrium has not been completely destroyed). But, the sender should be sending with less vigor to utilize less resource.

In the TCP Reno algorithm [7], the window size is cyclically changed in a typical situation. The window size continues to be increased until packet loss occurs. TCP Reno has two phases in increasing its window size: slow start phase and congestion avoidance phase. When an ACK (acknowledgment) packet is received by TCP at the sender side at time $t+t_A$[sec], the current window size $cwnd(t+t_A)$ is updated from $cwnd(t)$ using eq.(3).

$$cwnd(t+t_A) = \begin{cases} \text{slow start phase :} \\ \quad cwnd(t)+1, & \text{if } cwnd(t) < ssth(t); \\ \text{congestion avoidance phase :} \\ \quad cwnd(t)+\dfrac{1}{cwnd(t)}, & \text{if } cwnd(t) \geq ssth(t); \end{cases} \quad \textbf{(3)}$$

where cwnd(t) is window size of the sender and $ssth(t)$[packets] is a threshold value at which TCP changes its phase from slow start phase to congestion avoidance phase. When packet loss is detected by retransmission timeout expiration, $cwnd(t)$ and $ssth(t)$ are updated using eq.(4).

$$cwnd(t) = 1; \qquad ssth(t) = \frac{cwnd(t)}{2} \quad \textbf{(4)}$$

And when TCP detects packet loss by a duplicate ACK, it changes $cwnd(t)$ and $ssth(t)$ using eq.(5).

$$ssth(t) = \frac{cwnd(t)}{2}; \qquad cwnd(t) = ssth(t) \quad \textbf{(5)}$$

TCP Reno then enters a fast recovery phase and resends the lost data.

2.2.3 TCP Vegas

New TCP implementation, called Vegas is presented in 1994 by Brakmo, O'Malley and Petersen. [8]. TCP Vegas adopts a more sophisticated bandwidth estimation scheme. It uses the difference between expected and actual flow rates to estimate the available bandwidth in the network. The idea is that when the network is not congested, the actual flow rate will be close to the expected flow rate. Otherwise, the actual flow rate will be smaller than the expected flow rate. TCP Vegas, using this difference in flow rates, estimates the congestion level in the network and updates the window size accordingly.

TCP Vegas [7] controls its window size by observing RTTs (round-trip times) of packets that the sender host has sent before. If observed RTTs become large, TCP Vegas recognize that the network begins to be congested. If RTTs become small, on the other hand, the sender host of TCP Vegas determines that the network is relieved from the congestion, and increases the window size again. Hence, the window size in an ideal situation is expected to be converged to an appropriate value. The details of the algorithms are:

1. First, the sender computes the expected flow rate using eq.(6) as follows.

$$\textbf{\textit{Expected}} = \frac{cwnd(t)}{base_rtt}, \qquad \textbf{(6)}$$

   where ***cwnd(t)*** is the current window size and ***base_rtt*** is the minimum round trip time.

2. Then, the sender estimates the current flow rate by using the actual round trip time in eq.(7).

$$\textbf{\textit{Actual}} = \frac{cwnd(t)}{rtt}, \qquad \textbf{(7)}$$

   where ***rtt*** is the actual round trip time of a packet.

3. Then, using eq.(8), the sender computes the estimated backlog in the queue.

$$diff = \left| \frac{cwnd\ (t)}{base\ _\ rtt} - \frac{cwnd\ (t)}{rtt} \right| \qquad \textbf{(8)}$$

4. Based on *diff*, the sender updates its window size as follows by using eq.(9).

$$cwnd\ (t) = \begin{cases} cwnd\ (t) + 1, & \text{if } diff < \alpha \\ cwnd\ (t), & \text{if } \alpha \le diff \le \beta \\ cwnd\ (t) - 1, & \text{if } \beta < diff \end{cases} \qquad \textbf{(9)}$$

where, ***diff = Expected – Actual***

TCP Vegas tries to keep at least 'α' packets but not more than 'β' packets in the queues. The reason behind this is that TCP Vegas attempts to detect and utilize the extra bandwidth whenever it becomes available without congesting the network.

This mechanism is fundamentally different from that used by TCP Reno. TCP Reno always updates its window size to guarantee full utilization of available bandwidth, leading to constant packet losses, whereas TCP Vegas does not cause any oscillation in window size once it converges to an equilibrium point.

The flow chart in Figure 2.2 shows the algorithms of the two TCP types (TCP Reno and TCP Vegas).

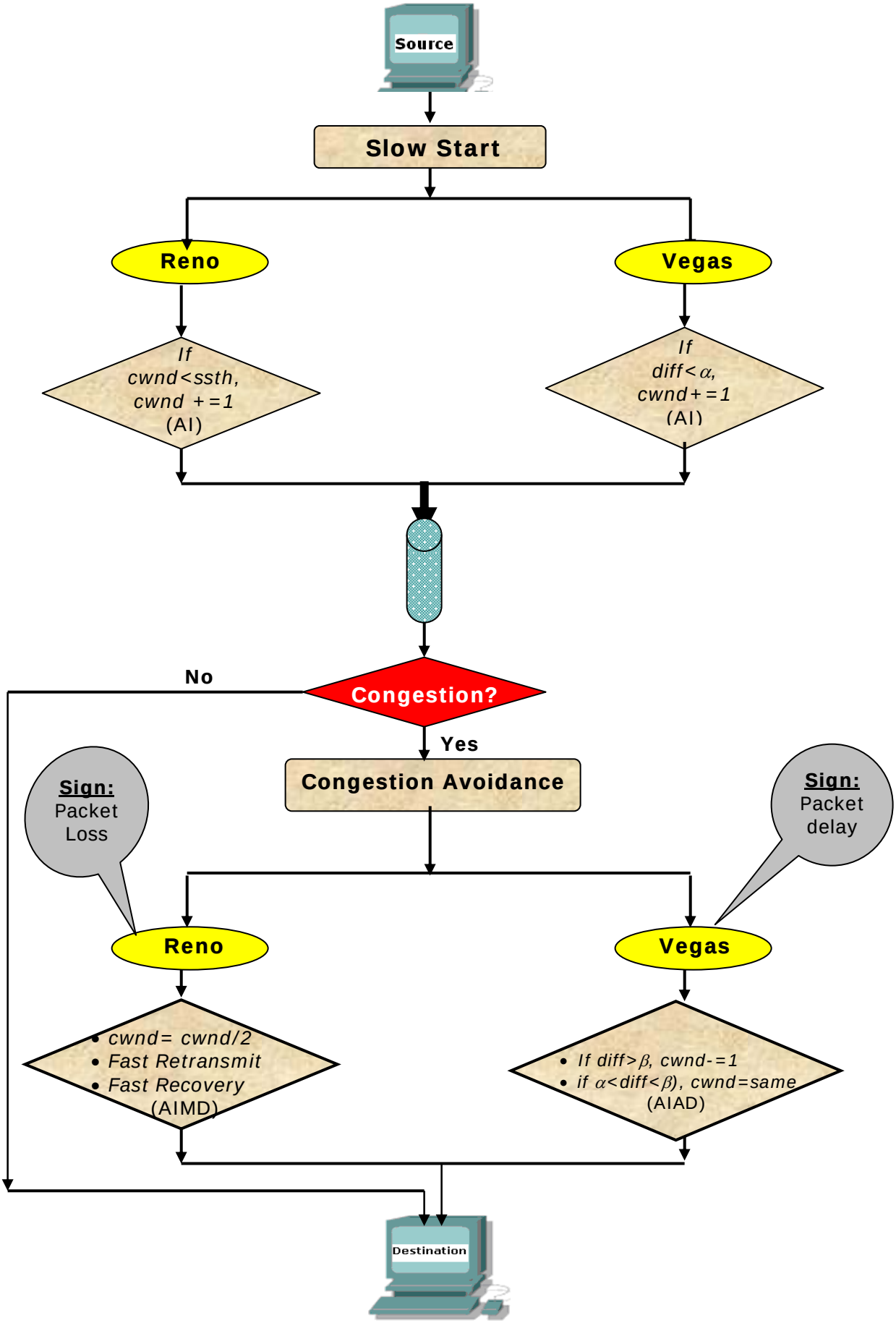


Figure 2.2 Flow chart of Congestion control of TCP Reno and TCP Vegas

2.2.4 Comparison between TCP Reno and TCP Vegas

TCP Reno and TCP Vegas have similarities and differences. The concept of the research papers [5-10], about the similarities and differences between TCP Reno and TCP Vegas is presented as follows:

1. **Packet timeout:** instead of only relying on a retransmission timer as with Tahoe and Reno, Vegas checks the RTT of a 'distinguished' packet. If the RTT of this packet is greater than expected, it will retransmit the packet before the retransmission timer expires.
2. **Slow start:**
   a. Reno can experience packet losses of up to half a ***cwnd*** due to its exponentially increasing reactive nature. Vegas attempts to predict when it should go from slow start (still exponential) to the congestion avoidance algorithm.
   b. TCP Vegas has another feature in its slow start. The rate of increasing its window size in slow start phase is one half of that in TCP Reno. Namely, the window size is incremented every other time an ACK is received.
3. **Congestion Avoidance mechanism:** The most important difference between TCP Vegas and Reno lies in this bandwidth estimation scheme. TCP Reno views packet losses as a signal of network congestion, while TCP Vegas implements a proactive approach to congestion, it calculates rates of what is expected to what it actually achieved.
4. **Efficiency:** The bandwidth estimation scheme enables TCP Vegas to utilize the available bandwidth more efficiently.
5. **Packet Retransmission:** The congestion avoidance method of Vegas causes much less packet retransmission than TCP Reno.

6. **Throughput:** Different research papers show that TCP Vegas achieves 37 to 71 percent higher throughput than TCP Reno.
7. **Bandwidth usage:** Reno has ***aggressive*** nature. It is trying to increase its window till packet is lost, so trying to use fill the buffer to the maximum. But in contrast TCP Vegas has a ***conservative*** nature that it is trying to use small buffer size.

2.2.5 Problems of TCP Vegas

Some studies showed that TCP Vegas has some problems that could have a serious impact on the performance of the network. These are:

1. **Rerouting:** Since TCP Vegas uses an estimate of the propagation delay, base-RTT, to adjust its window size, it is very important for a TCP Vegas connection to be able to have an accurate estimation. Rerouting a path may change the propagation delay of the connection, and this could result in a substantial decrease in throughput [13].
2. **Persistent Congestion:** Since TCP Vegas uses baseRTT as an estimate of the propagation delay of a route, its performance is sensitive to the accuracy of baseRTT. Therefore, if the connections overestimate the propagation delay due to incorrect baseRTT, it can have a substantial impact on the performance of TCP Vegas [13].
3. **Incompatibility:** When TCP Vegas competing with other TCP Reno connections, TCP Vegas gets penalized due to the aggressive nature of TCP Reno and TCP Vegas does not receive a fair share of bandwidth due to its conservative congestion avoidance mechanism [5]. This thesis work is focusing and trying to solve on the third problem.

Chapter 3

Relaxed-Vegas algorithm and its Implementation

3.1 Design Phase

3.1.1 Relaxed-Vegas Algorithm

Before implementing the new proposed solution, we have studied and analyzed the currently used detail algorithms of TCP Reno and TCP Vegas, their characteristics and why incompatibility problem occurs. As can be seen in the following chapter (4), simulations are done using NS2 and from the simulation results we verify the theory discussed in the literature review.

- From the previous discussions we can conclude that TCP Reno and TCP Vegas have different congestion control mechanisms. TCP Reno has a corrective nature, i.e. it has no any mechanism for the prediction of the congestion on the network. So it tries to solve the congestion problem after it happens. But TCP Vegas have a preventive nature, i.e. instead of congestion control, it tires to avoid and prevent before it happens.

- TCP Reno uses packet loss as a sign of congestion; it tries to fill the buffer till a packet loss is occurred. But TCP Vegas does not use drops as signal of congestion. It uses delay to adjust its thresholds and prevents the congestion before happening. It also tries to use very small part of the buffer and tries to leave more space in the buffer. So TCP Vegas is not relaxed to send more packets.
- From the simulation results it is clearly observed that TCP Vegas does not utilize the currently available bandwidth in the network. So to exploit the buffer to the maximum, without changing the basic behaviours of TCP Vegas, it is

possible to make TCP Vegas little-aggressive to some extent, and hence possible to send more packets to the queue. So we proposed **Relaxed-Vegas** as a solution for the incompatibility problem.

- But to do this, it is very important to keep the advantages of TCP Vegas. So we should measure the current and previous throughputs before changing the thresholds to make Vegas more relaxed.

To do so we have studied the detail algorithms of TCP Vegas and where the improvements should be done to get a better performance.

The Core congestion control algorithms of TCP Vegas and the new proposed solution Relaxed-Vegas are displayed as follows.

3.1.1.1 Existing Algorithm of TCP Vegas

Starting Points:

- 0<alpha<gamma<beta
- diff = Expected – Actual

o If (diff > gamma)
{Slow-down to ensure the net is not congested}

o If (diff > beta) {
{Slow down, back to previous “cwnd”, and
don’t increase in the next rtt}

o else if (diff < alpha)
{Increase cwnd}

3.1.1.2 Algorithm of Relaxed-Vegas

- Starting Points:
  - 0<alpha<gamma<beta
  - diff = Expected – Actual
- If (diff < alpha){
  - Calculate Throughput
  - If (current_throughput > previous_throughput)
    Increase cwnd and alpha }
- If (alpha ≤ diff ≤ gamma) {
  - Calculate Throughput
  - If (current_throughput > previous_throughput)
    - Increase cwnd, alpha & gamma }
- If (gamma < diff ≤ beta) {
  - If (current_throughput > previous_throughput)
    - Increase in slow rate (cwnd/2), gamma & beta }
- Else
  - If (current_throughput < previous_throughput) ,
  - If (diff > beta) ,
    - Slow down cwnd, gamma & beta

The congestion control algorithm of TCP Vegas, as displayed in the section **(3.1.1.1)**, when (diff<alpha), it considers the actual throughput is very close to the expected and hence things are working fine, and continues on that line to probe the network and increases the size of “cwnd”.

But all other cases lead TCP Vegas to a conclusion of "occurrence of congestion" and hence reduce the size of “cwnd” and corresponding thresholds values. Since every time Vegas want to leave minimum number of packets in the queue, while it tires to restrains

itself to maintain minimum number of packets in the queue other TCP implementations become aggressive and beat Vegas.

So one possible area is to look onto the possibility of telling Vegas not to worry on the number of packets it should leave in the queue. And this would help Vegas to be slightly more aggressive and probe the network for congestion.

So in our solution we are trying to exploit all the available bandwidth as much as possible, and we are exploring all the possibilities where the algorithm can be modified to get a better result as displayed in the algorithm of Relaxed-Vegas **(3.1.1.2).**

- In all cases Relaxed-Vegas compares the previous and current throughputs. If the current_throughput > the previous_throughput, the implication is that we are within the limit of actual and expected, and at the same time we have progress in throughput so we can be more aggressive and use the chance. So we increase the cwnd and the values of alpha and beta so that the possible ranges are a little bit wide.
- Still we have to entertain the network more and use the available bandwidth to the maximum, but we should not put the network also to the risk of congestion, so we use the intermediate threshold gamma (G) which is the point where Vegas turns from slow start to congestion avoidance stage to maintain the window.
- So instead of comparing directly (alpha<diff<beta), we first compare if (alpha≤diff≤gamma), then on the second step we compare if (gamma<diff≤beta), and in both cases we checked if the current throughput is greater than the previous throughput. If this condition is true the cwnd and the corresponding thresholds also updated. But in the second step since we are approaching to the maximum threshold we should increase in a slow rate that (cwnd/2).

- Finally if (diff > beta) or if the current throughput is smaller than the previous throughput, leading to the conclusion of occurrence of congestion, so the size of cwnd will be reduced. Since we have also modified the values of alpha, gamma and beta in the previous steps we reduce them also. The flow chart for Relaxed-Vegas is displayed in Figure 3.1.

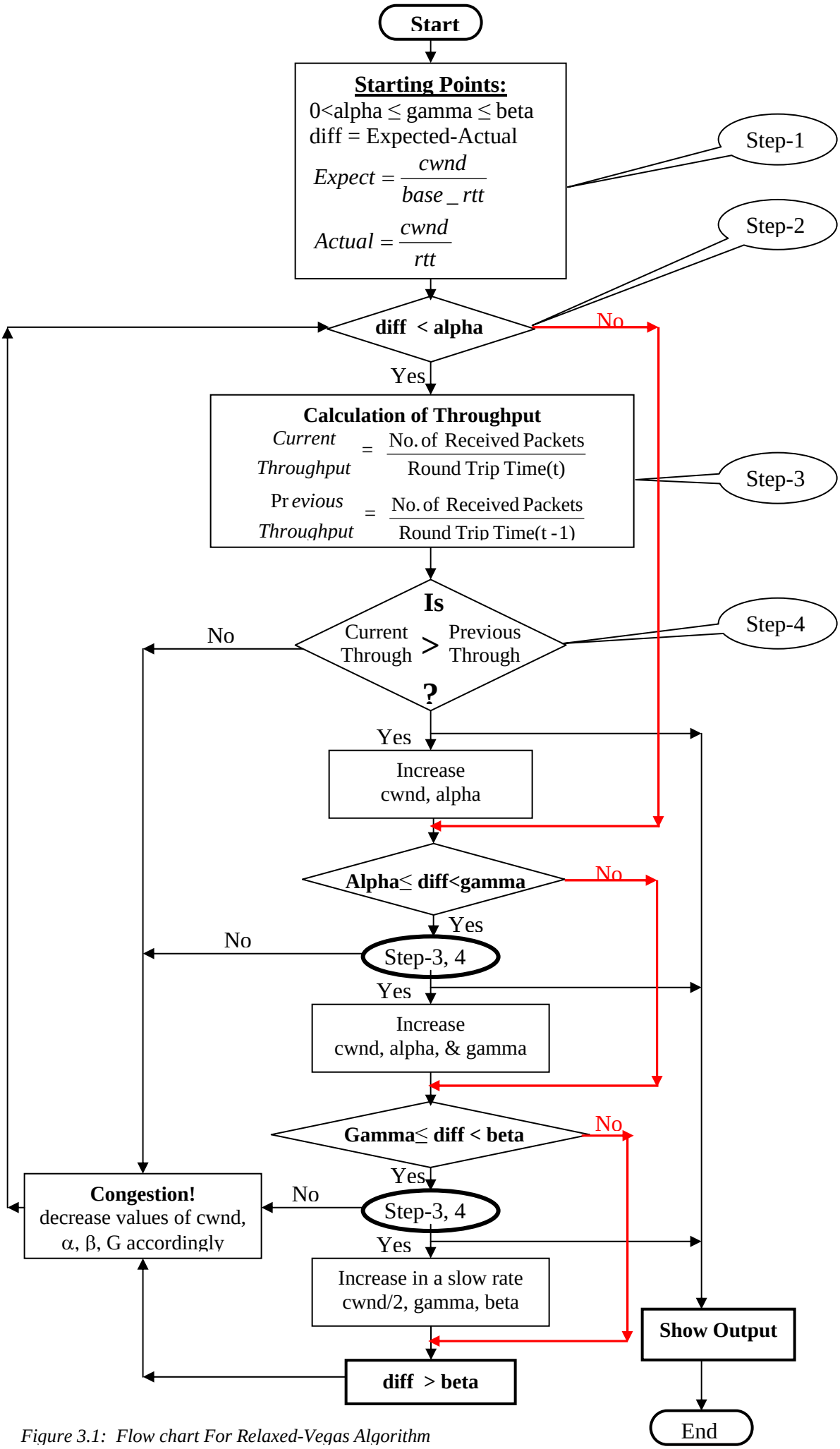


Figure 3.1: Flow chart For Relaxed-Vegas Algorithm

3.1.2 Design of Simulation Structure

In this part we design the general process of the simulation; including the topology of the Network, the simulation and the filtering programs to be used. Figure 3.2 shows the over-all simulation progress of the thesis work.

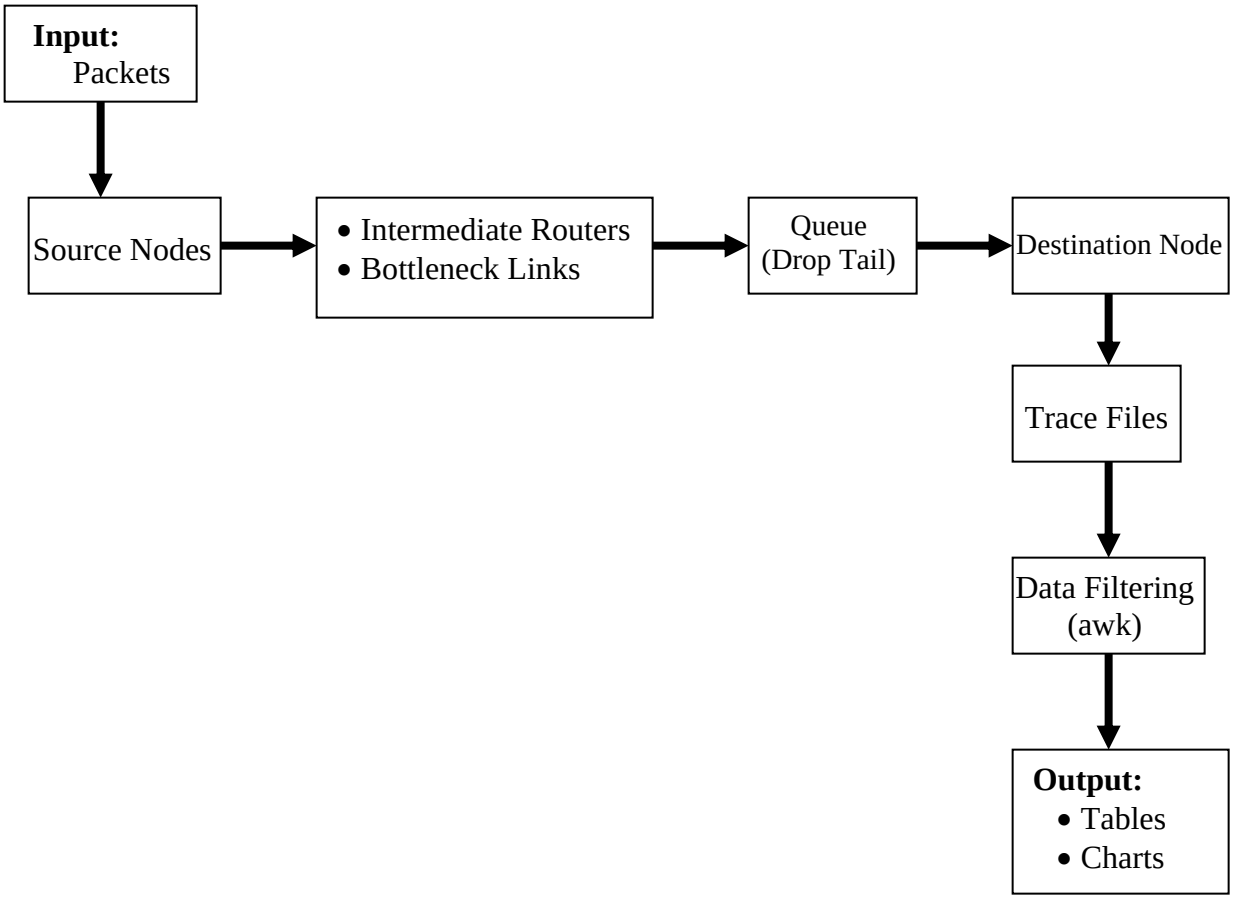


Figure 3.2 Block diagram of the over-all progress of the Simulation

Input Packets: Two types of traffics of packets are created here. That is traffics created by TCP Reno and TCP Vegas agents.

Source Nodes: Two source nodes are created here, one for TCP Reno and another for TCP Vegas. Then these nodes are attached to the above traffic sources.

Intermediate Routers and Bottleneck links: To observe the characteristics of the two TCP types, intermediate routers and bottlenecks should be created with a link which has less bandwidth.

Queue: Due to less bandwidth of the links all the packets from the source nodes can not transmit immediately to destination. So a Drop-Tail queue system is implemented here in the intermediate router.

Destination Node: Since our main aim is to receive all the packets sent from the source node to destination node, we create here the end of the path with a TCP sink agent where the results to be collected.

Trace Files: Here we get all the data from both sources together in a tabular form

Data Filtering: Since all packets are collected from both TCP sources filtering should be done here. To do so, two virtual nodes are used for collecting the sorted data for each TCP type separately.

Output: Output of the filtered data is finally changed into table and graph formats and used for analysis.

3.1.3 Design of Network Topology

The two algorithms of TCP congestion control mechanisms (TCP-Reno and TCP-Vegas) are represented and evaluated using the following network topology.

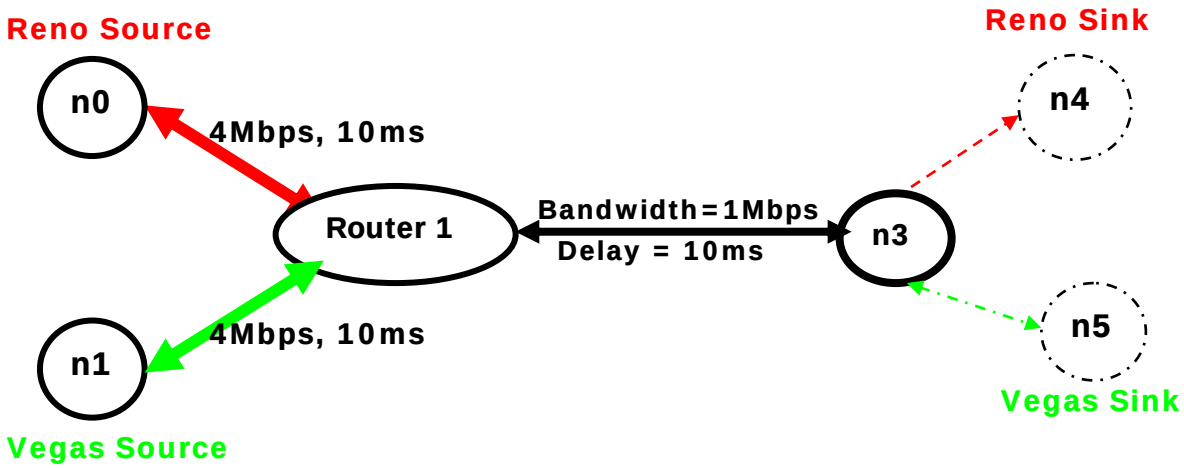


Figure 3.3: Topology layout used for Simulation

Using the above topology layout, two different sources of TCP are allowed to send FTP traffic to the destination. Each of the TCP source has a connection link of **4Mbps**, (Mega bits per second), bandwidth and **10ms,** (milli-seconds), delay to its nearest router, Router1. The bottleneck link from Router1 to node 3 (n3) has a bandwidth of **1Mbps** and **10ms** delay, which all the two algorithms are going to experience. The bottleneck is suitable for evaluating the algorithms on their performance of congestion control and congestion avoidance mechanisms.

Because all the sources are to pass through Router1, the queue size is bounded to a limit of **10**(not fixed), above which packets are to be discarded based on Drop-Tail queuing mechanism. Each TCP source is allowed to send FTP traffic with packet size of **460**bytes and a maximum queue size **45** packets for duration of **30**sec. Such traffic setup is suitable for collecting simulation output data on a chosen interval over a given bandwidth configuration at the same time by experiencing congestion.

3.1.4 Simulation

The word simulation is a generic term describing many different types of activities including: role playing in a social psychology experiment, complex video games, scale models built by engineers to describe behaviour of bridges or aircrafts etc.

Engineering approach while deploying a task is usually to begin with modelling the actual task and then simulating it with computer software. Simulating a given task by using some input factors and a few simplifying assumptions is finally to end up in proper deployment and installation of the real task. Networking is one major real problem which needs proper modelling and simulation before actual network configuration and installation. Simulation of a network problem is very important because of the unpredictable bursty nature of network traffic which leads to unwise and unfair purchase of network equipments. Before purchasing and deploying the actual network equipments one has to properly model and simulate the given task so that a clear picture of the real problem can be obtained. NS2 is the commonly used network simulation software which can help to analyze and predict actual network characteristics without investing any cost. Using NS2 one can properly study the characteristics of both wired and wireless networks.

3.1.4.1 Network Simulator Version 2 (NS2)

NS2 (Network Simulator Version 2) is an object-oriented, discrete event driven network simulator developed at UC Berkeley for simulating local and wide area networks.

NS2 implements different network protocols (TCP, UDP), traffic sources (FTP, web, CBR, Exponential on/off), queue management mechanisms (RED, DropTail), routing protocols (Dijkstra) over wired and wireless (local and satellite) networks etc. NS2 is written in C++ and OTcl languages to separate the control and data path

implementations. The reason why NS2 uses two languages is that different tasks have different requirements: For example simulation of protocols requires efficient manipulation of bytes and packet headers making the run-time speed very important where C++ is used. On the other hand, in network studies where the aim is to vary some parameters and to quickly examine a number of scenarios the time to change the model and run it again is more important, OTcl best suits for such a task. A well documented tutorial on how to install, configure and use NS2 can be found from [18].

Operating System Used: The software has been designed to work on Linux, but it is also made to run on Windows XP by using the Cygwin tool [12]. So we used the Cygwin tool on the Window XP Operating System.

Trace Files: Results of the simulations are shown in a tabular form in what ns-2 calls trace files. One line in a trace file is produced for each data packet that travels from the initiating node to the terminating node. Since the trace files record every parameter of a data packet, including its size, start time, type, time to live, starting node, and ending node, ns-2 generates huge trace files.

Data Filtering: Filtering out required data from the trace files presents another issue. The data in trace files is in tabular form, with each column separated by a white space. You need to employ a programming language to filter out the required data. Perl or awk can handle the white space issue well [12]. In this thesis work also we used the awk language to filter the simulation data from the trace files. A Tutorial and Introduction to Awk can be found from [1].

Gnuplot: For analysis purpose NS2 has some good tools such as Gnuplot and Xgraph for plotting the data in a graph. It is also possible to convert the data into tabular format and plotting charts in Microsoft Excel. In this thesis work the Gnuplot and Excel tools are used. The simulation results achieved while the algorithms working independently are immediately plotted using the Gnuplot. But since the simulation results achieved,

while working the algorithms simultaneously, are mixed data, it needs to be filtered into different files and putted in a tabular format. So in this case Microsoft Excel is used for plotting such kind of data.

Network Animator: Though ns-2 provides limited ability to view the animations after the simulation, its sister program Network Animator (NAM) makes it possible. NAM also can record the animation in the form of graphics as the simulation progresses. These graphics can then be converted to GIF or AVI format for later viewing. NAM also provides options for adjusting the step size of the animation in milliseconds, zooming in and out the animation [12].

In general we found that NS2 is very useful Simulator especially for simulations related to Networks. So we choose NS2 to be used in this thesis work.

3.2 Implementation Phase

- To implement the new algorithm, Relaxed-Vegas some tcl cods are modified in the ns2 library files.
- The currently working algorithm of TCP Vegas is found in the **"tcp-vegass.cc"** file of "ns2-2.29\tcp" folder. And the new algorithm, Relaxed-Vegas is also implemented in the same file.
- The standard threshold values are found in the library file of ns2 **"ns-default.tcl"** file of the "tcl\lib" folder. So the new values of Relaxed-Vegas also implemented there.
- **"Compilation"** process for each simulation is done to remove the old and to create the new object files so that the effects of the new values will be observed by the simulation.

3.2.1 Network Animator (NAM)

The network topology designed above will be represented using a Network Animator (NAM). So a simulation program is designed for the animation and collection of results of the simulation as follows.

As it is indicated in the following segment of code, the nodes created are *n0, n1, rtr1*, and *n3. n0* and *n1* represent the source nodes, *rtr1* represents the intermediate router and *n3* represents the destination node.

#Create nodes

```
set n0   [$ns node]
set n1   [$ns node]
set rtr1 [$ns node]
set n3 [$ns node]
```

Both the links from the sources to the router are defined with a bandwidth of 4Mb, but the bottleneck between the router1 and the destination node3 is designed with the bandwidth of 1Mb, and all follows the drop-tail queue management mechanism as indicated in the segment of code below.

#Create links between the nodes

```
$ns duplex-link $n0 $rtr1 4Mb 10ms DropTail
$ns duplex-link $n1 $rtr1 4Mb 10ms DropTail
$ns duplex-link $rtr1 $n3 1Mb 10ms DropTail
```

Here the queue size (queue limit) is number of packets to be queued. And all the packets more than that value are dropped. The queue size is only fixed for 30sec (i.e.) till one simulation time to be completed. Since observations of the results should be done using different queue limits, the values used in this simulation are:

2,4,6,10,15,20,25,30,35, 40 and 45. For example the second line below shows the queue-limit=40, (i.e.) maximum number of packets to be queued is 40 and any packet coming after the 40th packet is immediately discarded.

#limit queue size

$ns queue-limit $rtr1 $n3 40

In the following segment of code, source and destination agents are created and attached to the source and destination nodes respectively. *reno_tcp* is the sender defined as Reno class and attached to the source node, *n0*. And *reno_sink*, is the recipient defined for traffic sink and is attached to the destination node, *n3*.

#TCP Reno traffic source

#Create a TCP agent and attach it to node n0

set reno_tcp [new Agent/TCP/Reno]

$ns attach-agent $n0 $reno_tcp

#Create a TCP sink agent and attach it to node n3

set reno_sink [new Agent/TCPSink]

$ns attach-agent $n3 $reno_sink

#Connect both agents

$ns connect $reno_tcp $reno_sink

Similarly TCP Vegas is defined as follows:

#TCP Veags traffic source

#Create a TCP agent **and attach it to node n1**

set vegas_tcp [new Agent/TCP/Vegas]

$ns attach-agent $n1 $vegas_tcp

#Create a TCP sink agent and attach it to node n3

set vegas_sink [new Agent/TCPSink]

$ns attach-agent $n3 $vegas_sink

#Connect both agents

$ns connect $vegas_tcp $vegas_sink

To observe the characteristics of the algorithms the same applications type is used in both sides. And the application type used here is File Transmission Protocol (FTP) as can be seen in the following pieces of codes.

 # Create a Reno FTP source "application":

set reno_ftp [new Application/FTP]

$reno_ftp attach-agent $reno_tcp

 # Create a Vegas FTP source "application":

set vegas_ftp [new Application/FTP]

$vegas_ftp attach-agent $vegas_tcp

3.2.2 Implementation of Trace Files

The simulation output data is placed in trace files in tabular format with 12 columns as indicated below.

event	time	from node	to node	packet type	packet size	flags	fid	source address	destination address	sequence number	packet id

Figure3.4: Format of the trace file

The first column describes the events such as (r, +, - , d, for receive, enqueue, dequeue, and drop respectively). The second column is for simulation time of the event, then the next two columns record the link from and to be connected. The fifth column has the

information of type of packets such as "ack", tcp, udp, etc. The sixth column records the packet size in Bytes and then flags are located on the next column.

The next field is flow id (fid) of IPv6 that a user can set for each flow at the input OTcl script. Even though fid field may not used in a simulation, users can use this field for analysis purposes. The fid field is also used when specifying stream color for the NAM display. The next two fields are source and destination address in forms of "node**.**port". The next field shows the network layer protocol's packet sequence number. The last field shows the unique id of the packet [4].

3.2.3 Data Filtering using "awk"

On the first phase of the simulation since the data for each TCP type are recorded separately the results are filtered and written to files using NS2 tcl scripts as can be seen in the following segment of codes.

 # Calculate and write to the files

```
puts $wn_file "$now \t [$reno_tcp set cwnd_] \t [$vegas_tcp set cwnd_]"
puts $dr_file "$now \t $pktdropped"
puts $qu_file "$now \t [expr ($pktarrived - $pktdeparted -$pktdropped)]"
```

But for the second phase since the data are achieved in a mixed format it is necessary to use some different filtering methods. So the programming language (awk) is used to filter these data as can be seen from the next sample of the scripts.

```
# Filtering of mixed data
if(strEvent == "r" && to_node == 3)
      {if(fid == 2 && pkttype == "tcp")
            nReceivedPkts ++ ;}

if(strEvent == "+" && to_node == 3)
```

```
      {if(fid == 2 && pkttype == "tcp")
            nEnqueuedPkts ++ ; }

if(strEvent == "-" && to_node == 3)
      {if(fid == 3 && pkttype == "tcp")
            nDequeuedPkts ++ ; }

if(strEvent == "d" && to_node == 3)
      {if(fid == 3 && pkttype == "tcp")
            nDroppedPkts ++ ; }

END {
      printf("\n Enqueued Pakets  =  %d\n Dequeued Pakets  =  %d\n
      Received Packets =  %d\n  Dropped Pakets   =
      %d\n",nEnqueuedPkts, nDequeuedPkts, nReceivedPkts,
      nDroppedPkts);
    }
```

As can be seen from the above samples, the required data can be filtered from the trace files using awk program and used for analysis purpose.

Chapter 4

Simulation Results

This phase is where the above designed methods to be applied for collecting and testing of results and this is done in three steps.

1. Testing of Results while the two TCP types working **Independently**
2. Testing of Results while the two TCP types working **Simultaneously**
3. Implementing and Testing of the New Solution **(Relaxed-Vegas)**

4.1 Performance Metrics

Testing of the simulation results should be measured and analyzed using some performance metrics. So in this simulation the following performance evaluation metrics are used.

- **Queue Management**
- Stability of **Window Management**
- Number of **Received** Packets
- Number of **dropped** packets
- **Throughput** and
- **Overhead cost**

4.2 Simulation Results while TCP-Reno and TCP-Vegas Working Independently

From "tcp-vegas.cc" file we found the standard threshold values which are currently used in TCP Vegas as:

- α=1,
- β=3, and
- **G**=1,

  Where alpha **(α)** and beta **(β)** are the minimum and maximum threshold values of TCP Vegas respectively. And gamma **(G)** is a threshold value which decides the transition from the slow start to congestion avoidance stage.

To observe the currently used congestion control behaviours of the algorithms, the above threshold values are used and both TCP sources are allowed to use the link independent of each other. So each TCP type is running for 15sec. (i.e.) Packets sent from TCP Reno source starting at 0sec and stops at 15sec, and then TCP Vegas source node starts sending packets at 15sec and stops at 30sec. And the number of received and dropped packets from the simulation is displayed in table 4.1.

4.2.1 Number of Received and Dropped Packets

Table4.1: Received and dropped packets when working independently

TCP Type	Number of Received Packets	Number of Dropped Packets
Reno	**3580**	**36**
Vegas	**4012**	**0**

Here the number of packets to be queued is limited to 10. The reason why only one sample of the simulation with this queue limit is showed is, because all the simulation

with different queue limits shows similar results. So it is not need to display repetition of the values. In all outputs it is observed that many packets are dropped from Reno but no packets are dropped from Vegas.

The above simulation result shows that TCP Vegas received=4012 packets but TCP Reno received 3616 packets. And number of dropped packets from Reno=36 but none of the TCP Vegas's packets are dropped.

4.2.2 Drops of TCP Reno and TCP Vegas

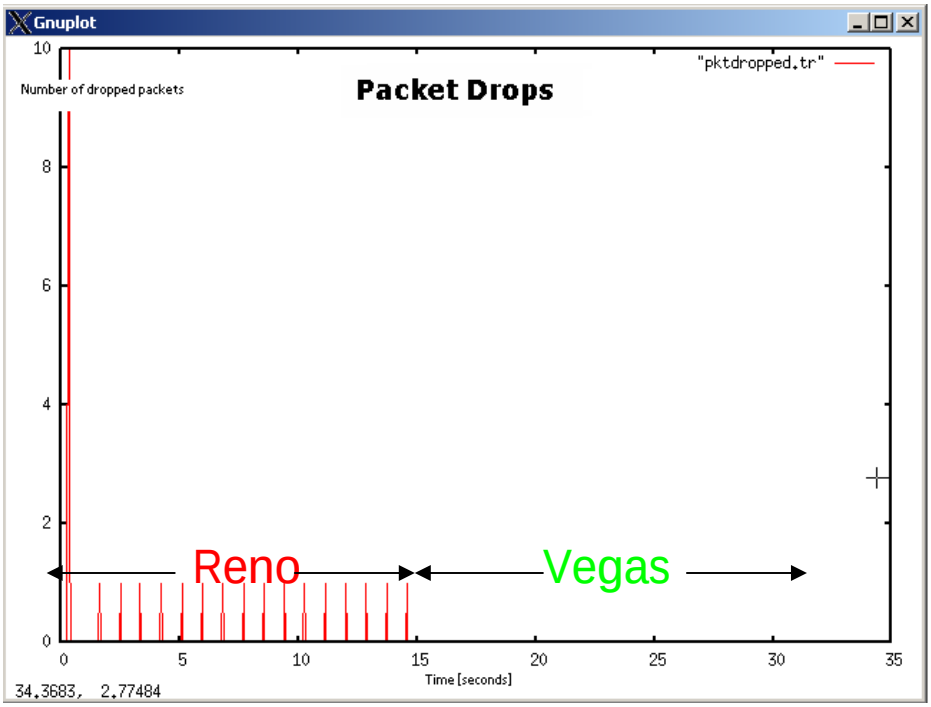


Figure4.1: Packet drop comparison between Reno and Vegas

Comparing TCP Reno and TCP Vegas for their packet losses, the above graph also assures that TCP Vegas experiences no packet loss. But Reno, as it tries to send more packets it faces for many packet drops.

Performance of the two TCP versions is also measured based on window and queue management in a router.

4.2.3 Stability of Window Management

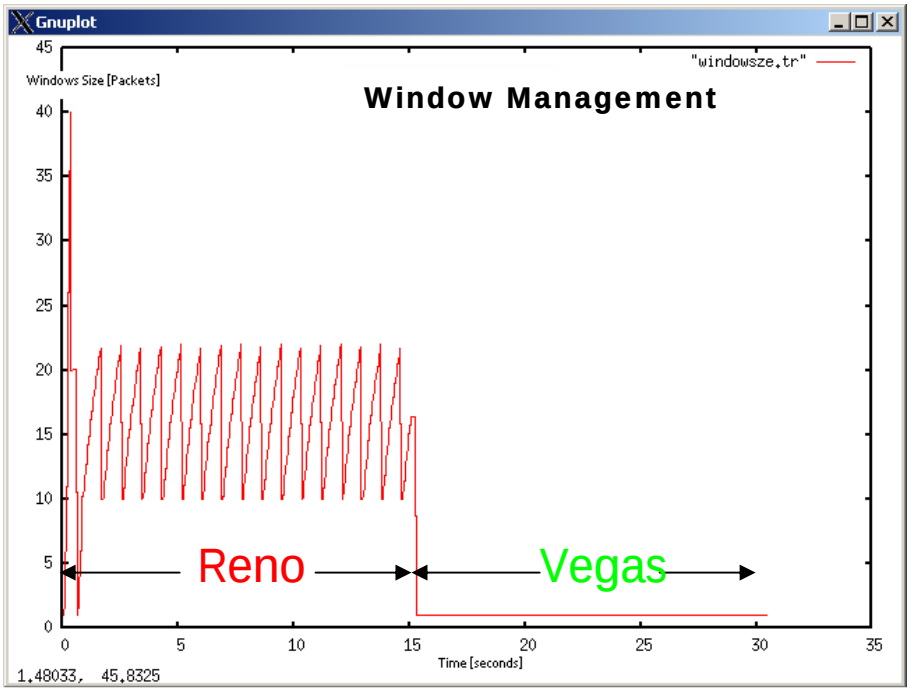


Figure4.2: Window management comparison between Reno and Vegas

4.2.4 Queue Management of TCP Reno and TCP Vegas

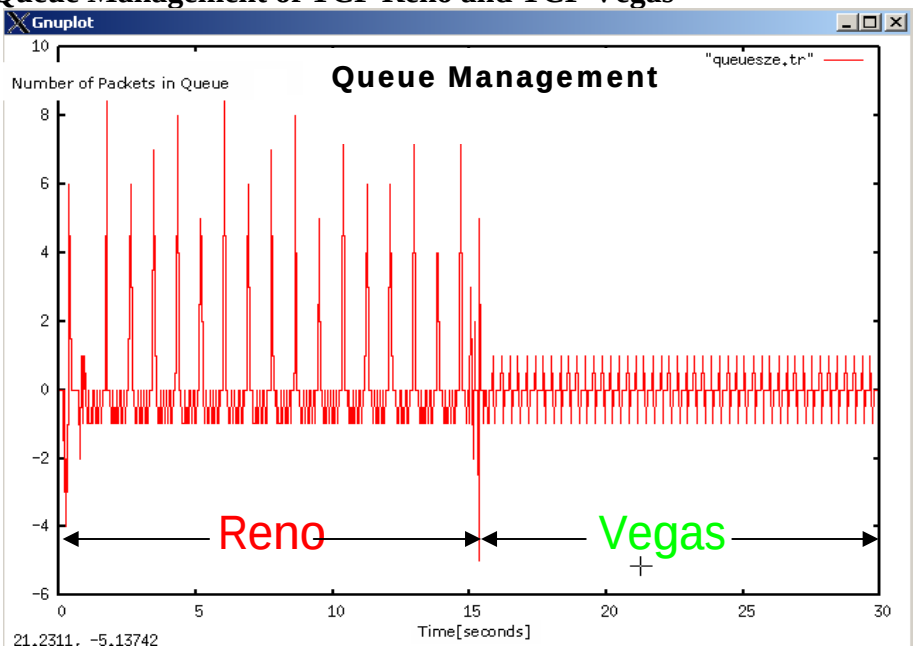


Figure 4.3: Queue management comparison between TCP Reno and Vegas

4.2.5 Throughput Comparison of TCP Reno and TCP Vegas

Different research papers show that TCP Vegas improves 31% to 71% of Reno's throughput [5]. Here also simulations are done to observe the throughput of TCP Vegas and TCP Reno while working independently. Throughput (Good-put) is the number of packets transferred within a specific period of time and the calculation is done using eq.(10).

$$\text{Throughput(Good-put)} = \frac{\text{Total Number of Received P}}{\text{Total Time (sec)}} \quad \textbf{(10)}$$

Throughput (Good-put) results are displayed in table and graph formats as follows:

Table 4.2 Throughput of Reno and Vegas while working independently

Queue-limit (pkts)	Throughput of Reno	Throughput of Vegas
2	23.57	56.80
4	104.30	127.10
6	111.53	133.33
10	119.33	133.53
15	121.06	133.36
20	120.63	133.00
25	119.46	133.66
30	125.00	109.06
35	125.00	132.63
40	125.00	132.63
45	125.00	132.60
Average	**110.90**	**120.70**

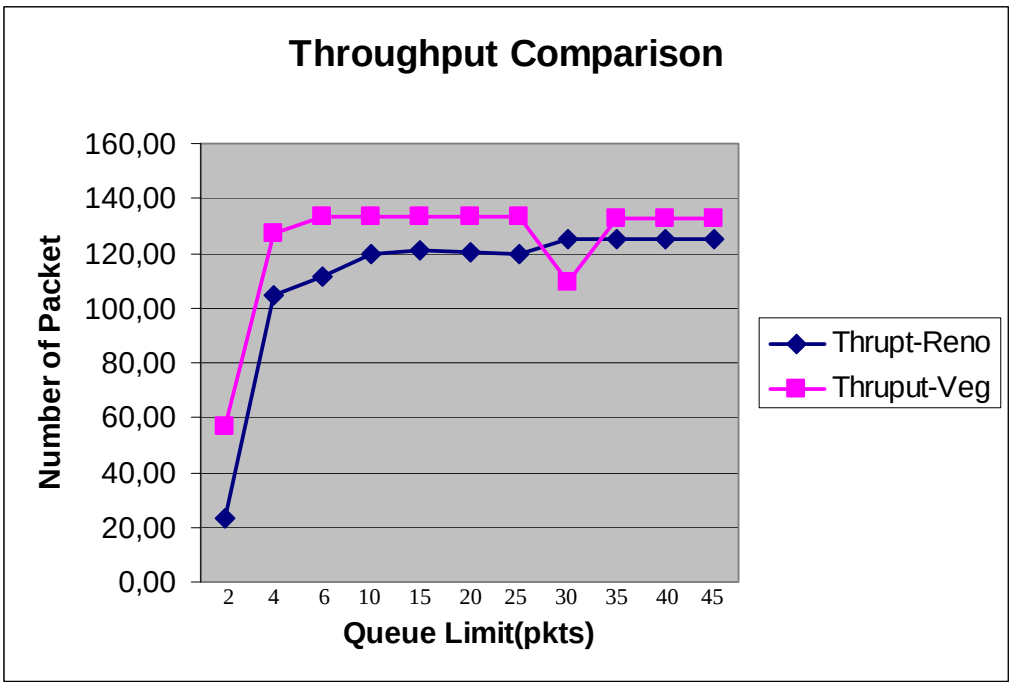


Figure 4.4: Throughput comparison of Reno and Vegas

4.2.6 Overhead Cost

Overhead cost is another performance metrics used to measure the number of Acknowledgments exchanged between the two hosts.

Number of ACKs

- Reno ACK = 28,640
- Vegas ACK = 32,030

4.3 Simulation Results while TCP-Reno and TCP-Vegas Working Simultaneously

In this part also the same topology and measurements as the above are used, the difference is here the two algorithms are running simultaneously. (i.e.) both algorithms start to send their traffic at the same time (at 0sec), and stopped at the same time (at 30sec). This simulation shows that how the two algorithms affect to each other.

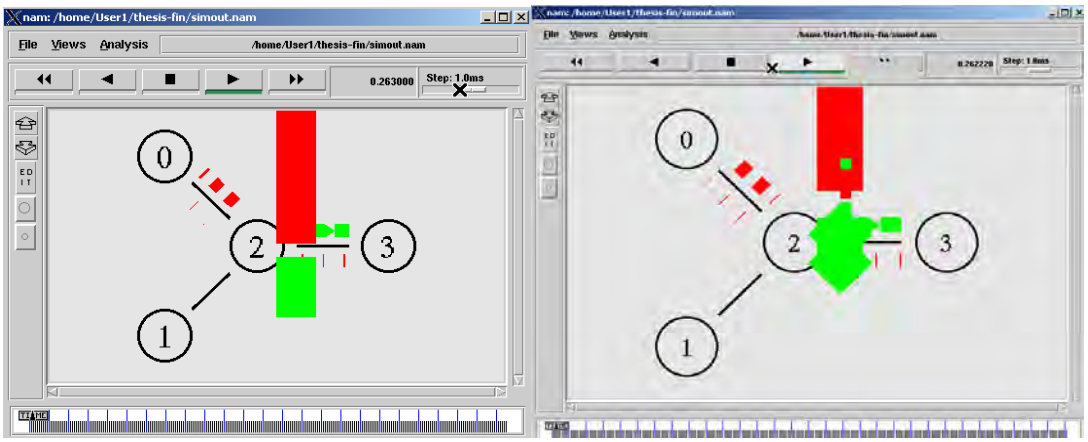


Figure 4.5: Screen captured of NAM while running simultaneously

As can be seen from the above two figures; both algorithms are suffering from dropped packets (red colour represents Reno packets and green represents the Vegas packets). Due to the behaviour of TCP Vegas, it tries to keep fewer packets in a queue to maintain the window size and it tries to occupy very minimum space of the buffer. But TCP Reno has an opposite behaviour that it tries to keep many packets in the queue till the buffer is full and a packet is lost. So it doesn't left any space that can be used by TCP Vegas. This opposite behaviour challenges Vegas and many of its packets are dropped down.

4.3.1 Simulation Results using the standard threshold values

In this phase also different simulations are done using different threshold parameters and the results are measured using the performance metrics. In this case the data is found mixed of the two types, so we can't use exactly the same metrics as in the previous work. The performance and the characteristics of the algorithms are measured using some different metrics.

4.3.1.1 Received Packets of Reno and Vegas

In this step first the standard (currently used) thresholds are used to observe the behaviour of the two TCP types when working together, and the results are displayed in the following table and graph.

The table and graph below show the simulation results using the standard threshold values of TCP-Vegas (α=1, β=3 and G=1).

Table 4.3: Received packets with standard threshold values (α=1, β=3 and G=1)

Queue Limit(pkt)	Received of Reno	Received of Vegas
2	**1434**	**1797**
4	**3451**	**3246**
6	**3199**	**4676**
10	**5240**	**2420**
15	**5350**	**2313**
20	**5385**	**2142**
25	**6129**	**1477**
30	**6372**	**1138**
35	**6838**	**723**
40	**6863**	**701**
45	**6863**	**701**
Average	**5193.09**	**1939.45**

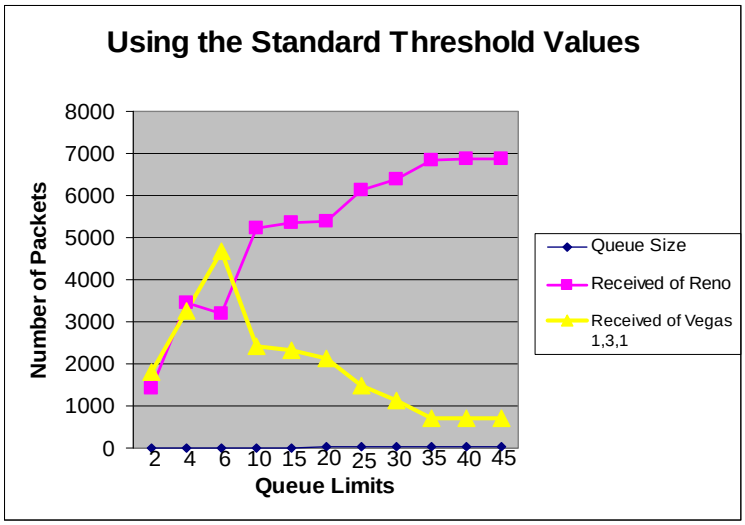


Figure 4.6: Received packets using the Standard Threshold Values

4.3.1.2 Dropped Packets from Reno and Vegas

Table 4.4: Dropped packets with standard threshold values (α=1, β=3 and G=1)

Queue Limit(pkts)	Dropped of Reno	Dropped of Vegas
2	117	149
4	79	146
6	86	4
10	58	6
15	46	2
20	35	1
25	23	1
30	16	0
Average	**57.5**	**38.6**

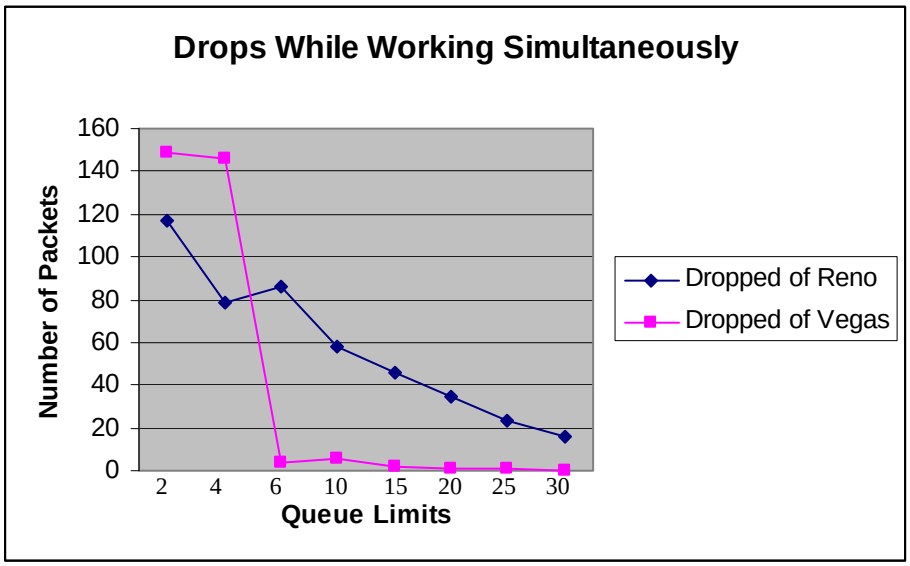


Figure4.7: Dropped packets using the Standard Threshold Values

4.3.1.3 Throughput Comparison of TCP Reno and TCP Vegas

Table 4.5: Throughput of TCP Reno and TCP Vegas while working simultaneously

Queue Limit(pkts)	Throughput of Reno	Throughput of Vegas
2	47.80	59.90
4	115.03	108.20
6	106.63	155.87
10	174.67	80.67
15	178.33	77.10
20	179.50	71.40
25	204.30	49.23
30	212.40	37.93
35	227.93	24.10
40	228.77	23.37
45	228.77	23.37
Average	**173.10**	**64.65**

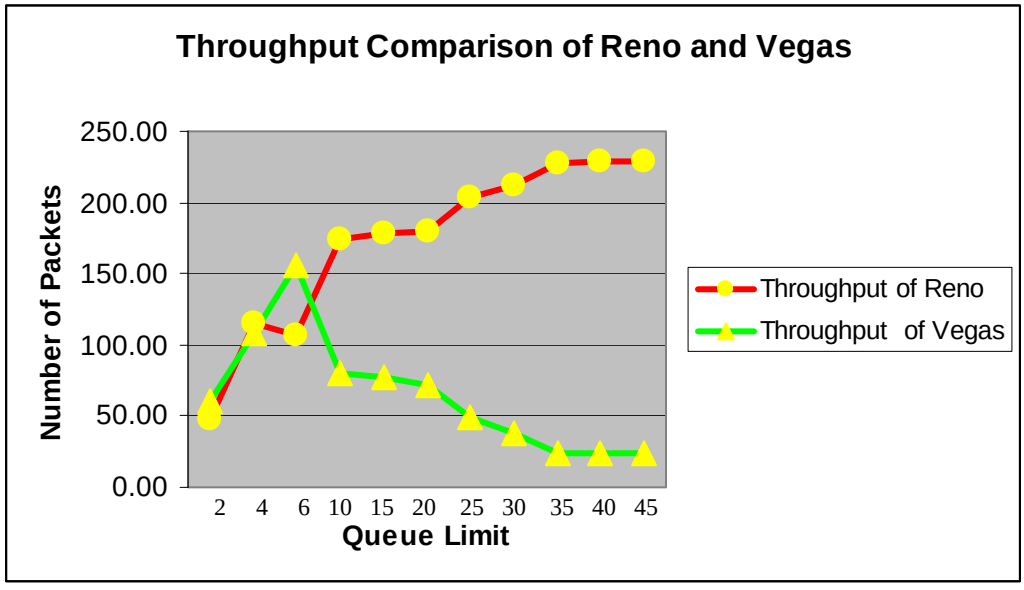


Figure4.8: Throughput of TCP Reno and TCP Vegas while working simultaneously

4.3.1.4 Overhead Cost

Table 4.6: Overhead cost of TCP Reno and TCP Vegas while working simultaneously

Queue Limit(pkts)	ACK of Reno	ACK of Vegas
2	10536	13192
4	26974	24753
6	24891	37315
10	41431	19272
15	42323	18440
20	42670	17088
25	48656	11775
30	50699	9080
35	54416	5763
40	54627	5564
45	54627	5584
Average	**41077.27**	**15256.91**

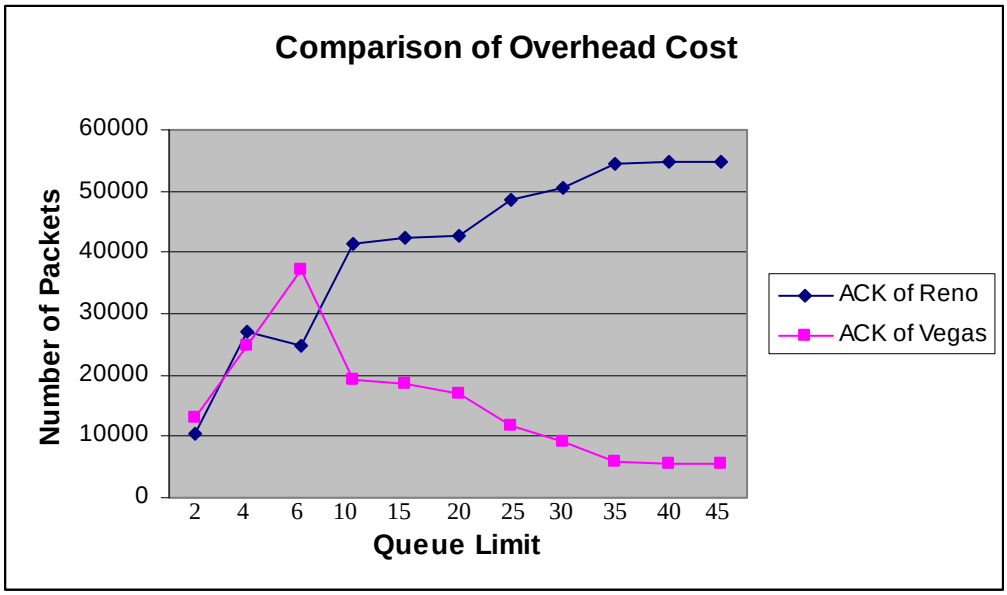


Figure4.9: Overhead cost while working simultaneously

4.4 Implementation and Simulation results of Relaxed-Vegas

The next simulations are done by changing the threshold values of Vegas as per of the stated in the Relaxed-Vegas algorithm. The simulations are categorized in to two steps. The first category displayed the effect of changing of alpha and beta with constant gamma, and the second displayed the effect of changing of gamma.

4.4.1 The Effect of Changing of alpha (α), Beta (β) with constant Gamma (G=1)

From previous research papers [14], they stated that better values are achieved when **(α=β)**. We agreed with this idea because from the above simulations we already observed the effects when the values of alpha and beta are different values. And as can be seen from the following simulations better outputs are achieved when alpha and beta are equal.

To study the output of these values 11 simulations for each category, (total 55 in this phase only), are done and the outputs are achieved and displayed in the following tables. In this category the simulation is done by fixing the value of gamma to 1, because first we have to observe the effect of changing of alpha and beta with constant gamma.

Table 4.7: Simulation results with α=β=2, G=1

Queue Limit(pkts)	Received of Reno	Received of Vegas 2,2,1
2	**1473**	**2362**
4	**1753**	**6106**
6	**2496**	**5275**
10	**4838**	**2881**
15	**5824**	**1816**
20	**6150**	**1459**
25	**6452**	**1092**
30	**6514**	**946**
35	**6976**	**571**
40	**7009**	**542**
45	**7009**	**542**

Table 4.8: Simulation results with $\alpha=\beta=3$, G=1

Queue Limit(pkts)	Received of Reno	Received of Vegas 3,3,1
2	**1450**	**710**
4	**2424**	**5091**
6	**3246**	**4484**
10	**4301**	**3444**
15	**5081**	**2641**
20	**5287**	**2260**
25	**6043**	**1567**
30	**6237**	**1285**
35	**6817**	**746**
40	**6863**	**701**
45	**6863**	**701**

Table 4.9: Simulation results with $\alpha=\beta=4$, G=1

Queue Limit(pkts)	Received of Reno	Received of Vegas 4,4,1
2	**1450**	**710**
4	**3987**	**2768**
6	**4150**	**3246**
10	**2150**	**5771**
15	**4425**	**3347**
20	**5294**	**2397**
25	**5744**	**1903**
30	**5935**	**1669**
35	**6228**	**1356**
40	**6593**	**997**
45	**6593**	**997**

Table 4.10: Simulation results with $\alpha=\beta=5$, G=1

Queue Limit(pkts)	Received of Reno	Received of Vegas 5,5,1
2	**1450**	**710**
4	**1949**	**5496**
6	**3405**	**3246**
10	**2561**	**5402**
15	**3828**	**4002**
20	**4804**	**2931**
25	**5338**	**2346**
30	**5711**	**1944**
35	**6012**	**831**
40	**6465**	**1137**
45	**6465**	**1137**

Table 4.11: Simulation results with $\alpha=\beta=6$, G=1

Queue Limit(pkts)	Received of Reno	Received of Vegas 6,6,1
2	**1450**	**710**
4	**1949**	**5896**
6	**3405**	**3635**
10	**2377**	**3653**
15	**3782**	**4043**
20	**4319**	**3460**
25	**5022**	**2687**
30	**5372**	**2093**
35	**5675**	**831**
40	**6343**	**1270**
45	**6343**	**1270**

The above results are plotted in the following graph. And the average values of each category are displayed in table 4.12.

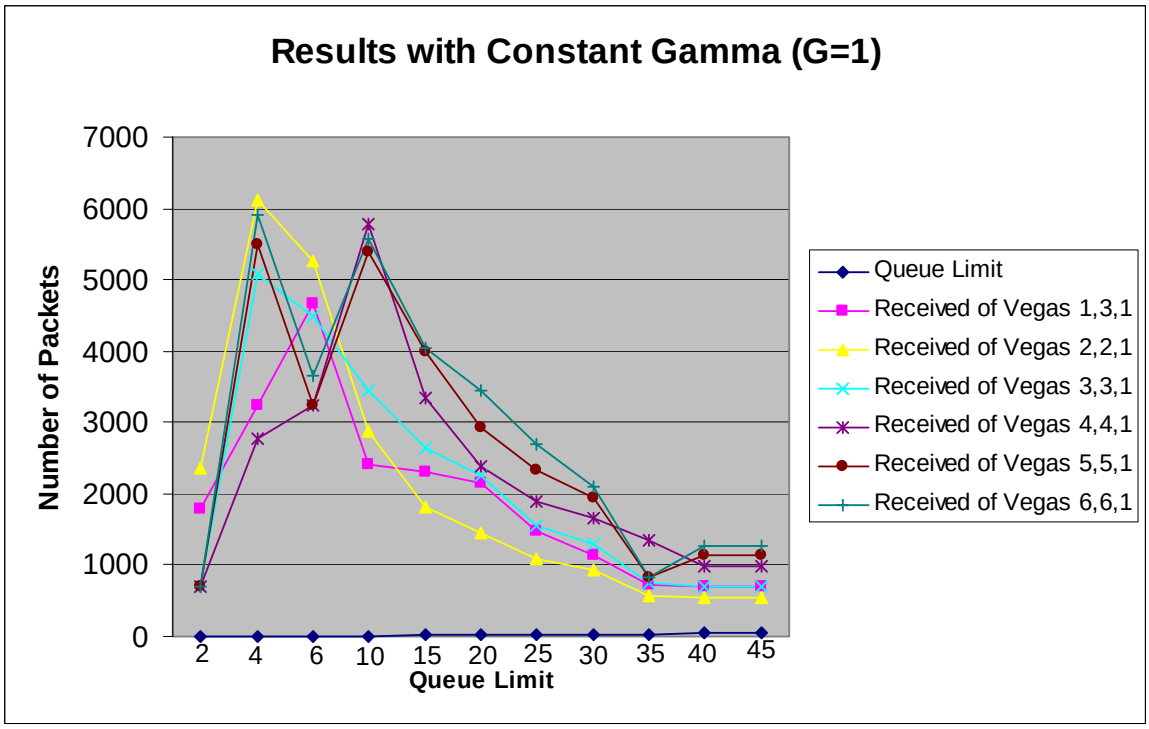


Figure4.10: Received packets with constant gamma (G=1)

Table 4.12: Average Values of the Received Packets with constant gamma (G=1)

Received of Vegas 1,3,1	Received of Vegas 2,2,1	Received of Vegas 3,3,1	Received of Vegas 4,4,1	Received of Vegas 5,5,1	Received of Vegas 6,6,1
1939.45	**2144.73**	**2148.18**	**2287.36**	**2652.91**	**2686.18**

4.4.2 The Effect of Changing of Gamma (G)

In all the above simulations and all the research papers we found, Gamma is fixed (**Gamma=1**). But Gamma is the threshold value of Vegas that change from slow-start to congestion-avoidance stage. So it is very important to study the effect of the change of Gamma in the network. There fore in this thesis simulations are done by changing the values of Gamma and the results are displayed below.

As discussed before, since the better result is achieved using the **α=β=6, Gamma=1**, we fix the values of alpha and beta to be 6, and only the gamma varies.

Table 4.13: Simulation results with the change of Gamma

Queue Limit (pkts)	Received of Vegas 6,6,1	Received of Vegas 6,6,2	Received of Vegas 6,6,4	Received of Vegas 6,6,5
2	**710**	**710**	**710**	**710**
4	**5896**	**5920**	**5920**	**5920**
6	**3635**	**3653**	**3653**	**4241**
10	**3653**	**3653**	**5581**	**5581**
15	**4043**	**4043**	**4043**	**4043**
20	**3460**	**3460**	**3460**	**3460**
25	**2687**	**2687**	**2687**	**2687**
30	**2093**	**2093**	**2093**	**2306**
35	**831**	**831**	**831**	**1969**
40	**1270**	**1271**	**1271**	**1271**
45	**1270**	**1279**	**1279**	**1279**
Average	**2686.18**	**2690.91**	**2866.18**	**3042.45**

The results are verified by calculating the average values of each output.

The results achieved when Gamma varies are plotted in the following graph.

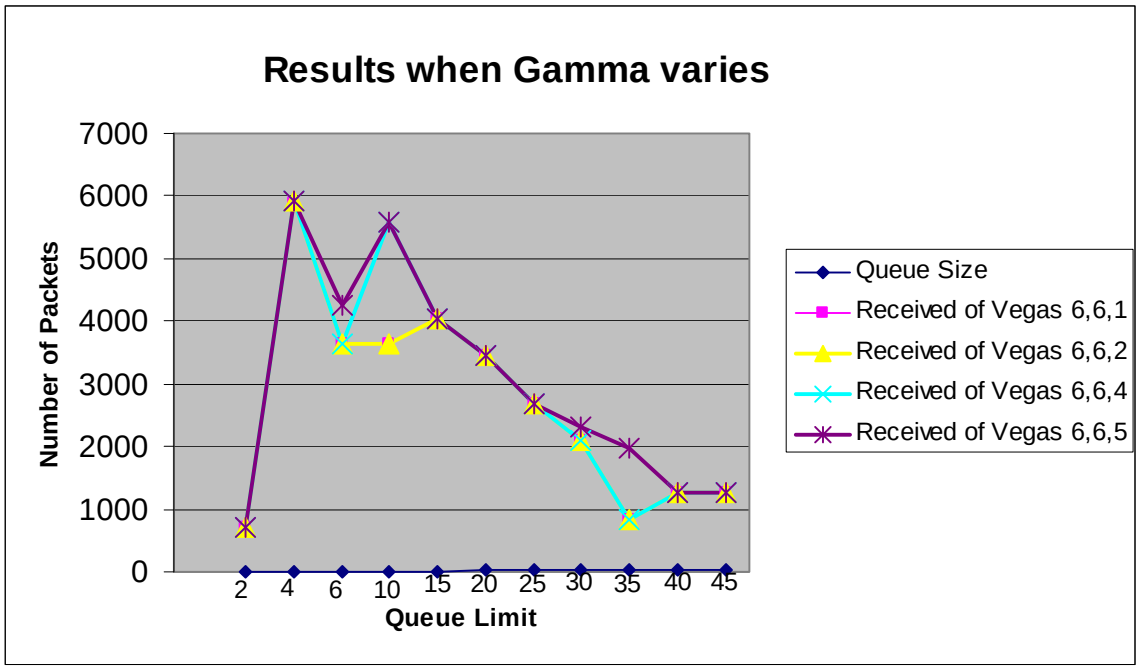


Figure4.11: Received packets with different values of Gamma

4.4.3 Throughput of Relaxed-Vegas

Table 4.14: Throughput results with change of Gamma

Queue Size	Throughput of Vegas 6,6,1	Throughput of Vegas 6,6,2	Throughput of Vegas 6,6,4	Throughput of Vegas 6,6,5
2	**23.67**	**23.67**	**23.67**	**23.67**
4	**196.53**	**197.33**	**197.33**	**197.33**
6	**121.17**	**121.77**	**121.77**	**141.37**
10	**121.77**	**121.77**	**186.03**	**186.03**
15	**134.77**	**134.77**	**134.77**	**134.77**
20	**115.33**	**115.33**	**115.33**	**115.33**
25	**89.57**	**89.57**	**89.57**	**89.57**
30	**69.77**	**69.77**	**69.77**	**76.87**
35	**27.70**	**27.70**	**27.70**	**65.63**
40	**42.33**	**42.37**	**42.37**	**42.37**
45	**42.33**	**42.63**	**42.63**	**42.63**
Average	**89.54**	**89.70**	**95.54**	**101.42**

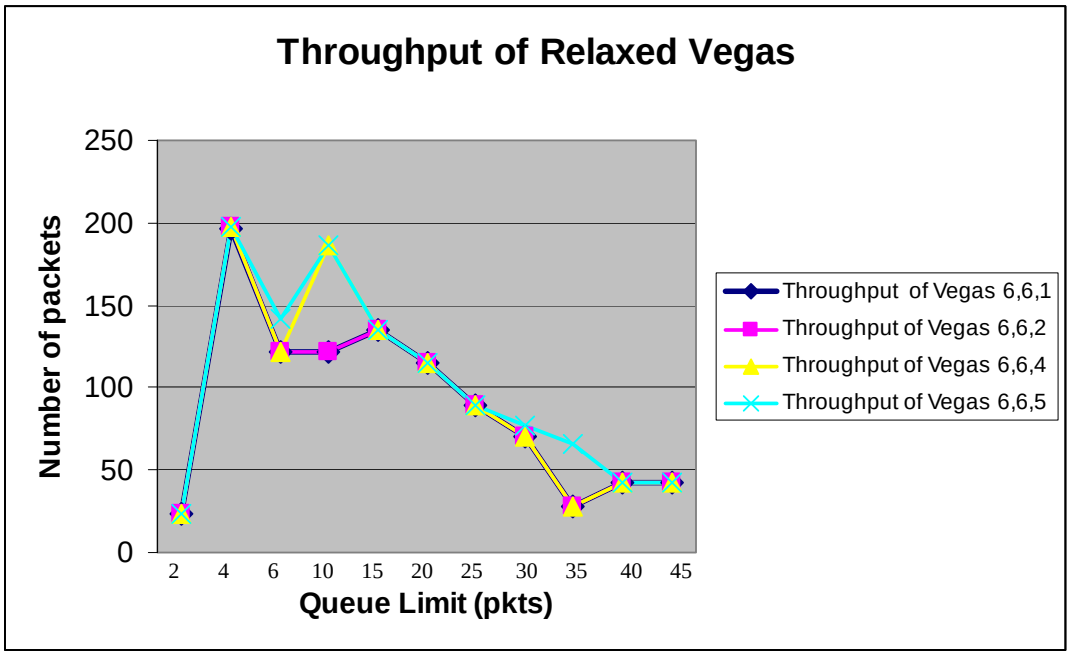


Figure 4.12 Throughput with change of Gamma

4.4.4 Dropped Packets of Relaxed-Vegas

Table 4.15: Drops of Relaxed-Vegas

Number of Dropped Packets		
Threshold	**Sum**	**Average**
Vegas(2,2,1)	**496.98**	**45.18**
Vegas(3,3,1)	**531.96**	**48.36**
Vegas(4,4,1)	**496.98**	**45.18**
Vegas(5,5,1)	**440.00**	**40.00**
Vegas(6,6,1)	**367.95**	**33.45**
Vegas(6,6,2)	**321.97**	**29.27**
Vegas(6,6,4)	**388.96**	**35.36**
Vegas(6,6,5)	**336.05**	**30.55**

4.5 Comparison of TCP Vegas and Relaxed-Vegas

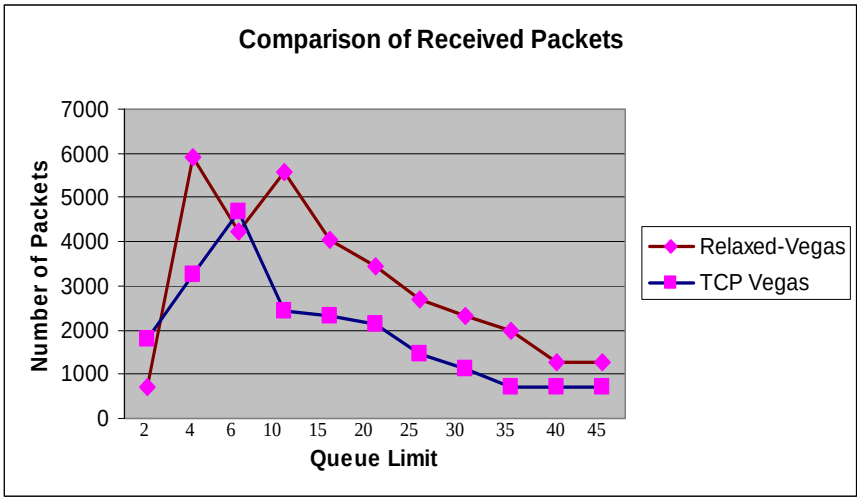


Figure 4.13 Throughput comparison of TCP Vegas and Relaxed Vegas

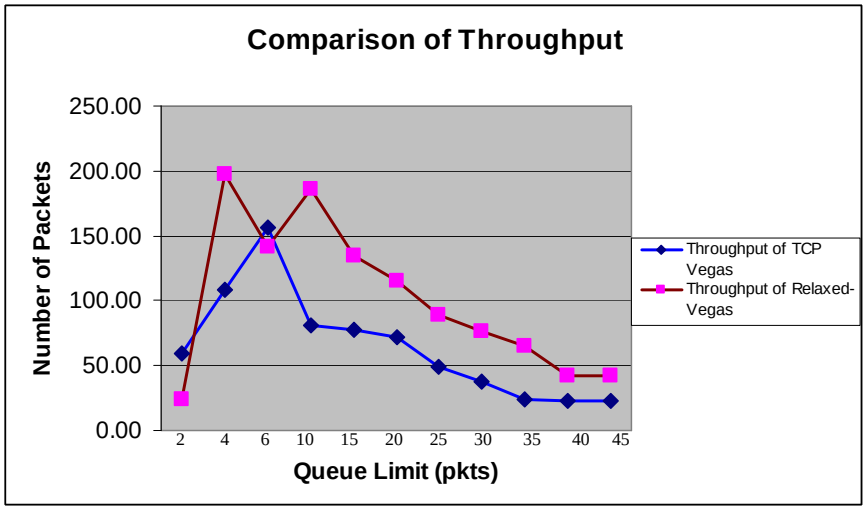


Figure 4.14 Throughput comparison of TCP Vegas and Relaxed Vegas

Table 4.16: Comparison of Drops of TCP Vegas and Relaxed-Vegas

Comparison of Dropped Packets	
TCP Vegas	**Relaxed-Vegas**
Vegas(2,2,1)	**45.18**
Vegas(6,6,5)	**30.55**

Chapter 5

Analysis and Discussion of Results

As the simulations categorized in the above, the discussion of the results is also done in 3 parts.

- Discussion of Results while the two TCP types working independently
- Discussion of Results while the two TCP types working simultaneously
- Discussion of Results of Relaxed-Vegas

5.1 Discussion of Results while TCP-Reno and TCP-Vegas working Independently

In this part simulations are done to observe the characteristics of the two TCP types while working independently and it is evaluated using the performance metrics stated above. The following are the observations:

Observation 1: From Table-4.1 it is obvious that when TCP Vegas running alone the number of received packets is greater than the Reno's. Also it is observed that many packets from Reno are dropped but there is no drop of packets from Vegas. Figure 4.1 also assures the same.

Observation 2: In Figure 4.2, it is clear that TCP Reno's window highly fluctuates. It is because of TCP Reno is trying to increase its window size till packet loss is occurred, and it tries to fill the buffer to the maximum. But TCP Vegas maintains its window size in a constant way. And this shows that window management of TCP Vegas is more stable than TCP Reno.

Observation 3: Figure 4.3 shows that as congestion decreases TCP Reno try to send more packets and hence reaches to the queue limit quickly. Once it reaches the limit, it slows down its window and at the same time decreases its number of queued packets. This case is different when using TCP Vegas by no means that Vegas reaches the maximum queue limit, most of the time Vegas keeps very less packet in a queue because Vegas's main target is to leave more space free in the queue as mach as possible.

Observation 4: The records in Table 4.2 and Figure 4.4 show that the throughput of TCP Vegas is more than the throughput of TCP Reno.

From section 4.2.6, the number of ACKs of TCP Vegas is more than TCP Reno's. The reason behind is that TCP Vegas transfers more packets and to calculate the difference of the current and previous values of RTT also it needs to exchange more information and then maintains its window size due to that information. So its overhead cost is more than the Reno's.

5.2 Discussion of Results while TCP-Reno and TCP-Vegas working Simultaneously

In this part simulations are done to observe the characteristics of the two TCP types while working simultaneously and it is evaluated using the performance metrics stated above. The following are the observations:

Observation 1: As can be seen from Table 4.3 and Figure 4.6, when the queue limit increases, TCP Reno gets more advantages to send more packets but TCP Vegas still suffers to less space in the queue.

Observation 2: From Table 4.4 and Figure 4.7 it is observed that many packets of TCP Reno are dropped in all simulations. It is observed also many packets of TCP

Vegas are dropped. But TCP Vegas maintains its window size and control the drops and finally reduced to zero. Even though it controls at the end, this value is very huge comparing to Vegas's behaviour so it should be improved using the new solution.

Observation 3: One of the main and good qualities of Vegas is it achieves higher throughput than all the previous TCP types including Reno. But as can be seen from Table 4.5 and Figure 4.8, while Vegas is working with Reno it throughput becomes the reverse. From the table it is seen that throughput of Reno is 173.10 but Vegas's throughput is 64.65. Therefore this problem should be also improved in the new solution.

Observation 4: As the records of Table 4.6 and Figure 4.9 the number of ACK they exchanging between the source and the destination is increasing from both types of TCPs but here number of ACKs from Vegas is less than Reno's because since most of the space is occupied by Reno's packets Vegas sends and receives very less number of packets. So the influence is also clearly observed here.

5.3 Discussion of Results of Relaxed-Vegas

Observation 1: From Table 4.7 better results are achieved when queue size is 2,4,6, and 10. But when queue size increases (15, 20,..., 45), the amount of packets received from Vegas decreases. And from Table 4.8 when the value of $\alpha=\beta=3$ it is observed that most of the outputs are better than the previous, but with some exception (for example when the queue size = 6, the received value are less than the previous. Including Tables 4.9, 4.10 and 4.11, it observed that a better result is achieved when $\alpha=\beta=6$.

The summery of the outputs is displayed in Figure 4.10. And this graph also shows that the received of Vegas's packets when the thresholds $\alpha=\beta=6$, **G=1**, is highest from all the results with constant gamma (G=1). We also tried to make it clear by calculating

average values of the received packets in each simulation. And the results are found in the Table 4.12.

Observation 2: In this part all the simulation results assure that a change of gamma has a great effect in the outputs. As can be seen from Table 4.13 as gamma increases, the number of packets sent from the Vegas source also increases. From all the above values the better value is achieved when $\alpha=\beta=6$, $G=5$.

Note that many simulations also done with different values but similar results are achieved, (for example the output achieved with $\alpha=\beta=6$, $G=6$ is similar to the output achieved with $\alpha=\beta=6$, $G=5$). So we discarded the repeated values.

Observation 3: From Table 4.14 and Figure 4.12, it is observed that the change of the value of Gamma affects the throughput.

The last line of the same table shows the average results of the throughput gained for each category. From that we observed when the value of Gamma increases the throughput achieved also increases.

Observation 4: Figure 4.13 and Figure 4.14 Clearly shows the results of the received packets and the throughput of currently working TCP Vegas and the new proposed method Relaxed-Vegas. And it shows Relaxed-Vegas achieve more packets and higher throughput than TCP Vegas.

Observation 5: The dropped packets of Relaxed-Vegas in average for each category are displayed in Table 4.15. In this part it is observed that even though still some packets of Vegas are dropped but the number is less as can be seen from Table 4.16.

Chapter 6

Conclusions and Recommendations

6.1 Conclusions

Different researches assure that TCP Vegas improves the throughput of TCP Reno by 37 to 71% [5]. And the performance evaluation shows that Vegas achieves higher efficiency than Reno, and causes less packet retransmissions. But one of the main problems is incompatibility of TCP Vegas with TCP Reno. That is when TCP Vegas competes with TCP Reno there is unfair share of a bandwidth. This is because of their different congestion avoidance mechanisms. TCP Reno increase continuously its window size till the loss of the packets occurs but TCP Vegas has a conservative nature; it tries to use the queue size as small as possible.

This thesis work explored the current TCP Vegas congestion control mechanisms and its incompatibility problems with other TCP (TCP Reno). And using different simulations we observed that:

- From all the simulation results in part-1 we observe that when congestion happen TCP Reno is going to suffer much than Vegas because many of its packets are getting dropped. But TCP Vegas still continue to utilize the link to the maximum as none of its packets are allowed to drop. This implies that Vegas has a more sophisticated bandwidth estimation scheme and tries not to override the available bandwidth. In general from the above observations we can conclude that TCP Vegas in isolation performs better in overall network utilization, stability, throughput, and packet loss.
- From the simulation results of part-2 it observed that when TCP Vegas is working with other TCP Reno connections, its performance decreases and gets

penalized due to the aggressive nature of TCP Reno. And Vegas do not receive a fair share of bandwidth due to its conservative congestion avoidance mechanism.

- From the simulation results of Part-3 we studied the effect of the changing of the standard threshold values currently used in TCP Vegas (α=1,β=3, G=1). And we found that α=β=6, are best values to be used. Also we didn't found studied papers which considers the effect of the change of Gamma, but in this thesis work it is seen that Gamma has a great influence in the performance of TCP Vegas and we found the best value is G=5. And using the new proposed method (Relaxed-Vegas) better results are achieved in:
    - Number of Received Packets
    - Throughput and
    - Less dropped packets

Average	TCP Vegas	Relaxed-Vegas	Difference	% Increase
Received Packets	**1939.45**	**3042.45**	**1103.00**	**56.87**
Throughput	**64.65**	**101.42**	**36.77**	**56.87**
Dropped Packets	**45.18**	**38.42**	**6.76**	**17.59 (% Decrease)**

Table 6.1: Comparison of TCP Vegas and Relaxed-Vegas

There fore, we conclude that making Vegas aggressive to some extent (Relaxed-Vegas) can solve the Incompatibility problem and improves the performance while TCP Vegas working with other TCP Reno Connections.

6.2 Recommendations

- For future works this method can be implemented in high speed processors with large number of data.
- And we recommend for future works to implement this solution in a practical Network.
- Also we recommend to use the new values (**α=β=6** and **G=5),** as a standard threshold values of TCP Vegas.

Appendix A: Awk Scripts for Metrics Calculations

A.1 Received of Reno

```
BEGIN {a=0; no_pkts=0;}
      {
      event = $1 ;
      to_node = $7 ;
      fid = $17 ;     # 2= reno fid,  3= vegas fid
      pkttype = $9 ;
      pktsize = $11 ;
      }
{
      if($1 == "r" && to_node == 3)
      { if(fid == 2 && pkttype == "tcp")
      {no_pkts ++;
       a = a+ pktsize;}
      }
}

END {print "\nNumber of RecvPkts = " no_pkts "\nOutput = " a }
```

A.2 Received of Vegas

```
BEGIN {a=0; no_pkts=0;}
      {
      event = $1 ;
      to_node = $7 ;
      fid = $17 ;
      pkttype = $9 ;
      pktsize = $11 ;
      }
{
      if($1 == "r" && to_node == 3)
      { if(fid == 3 && pkttype == "tcp")
            {no_pkts ++ ;   a = a+ pktsize;}
      }
}


END {print "\nNumber of RecvPkts = " no_pkts  "\nOutput = " a }
```

A.3 Dropped of Reno

```
BEGIN {
      nReceivedPkts = 0 ;
      nEnqueuedPkts = 0 ;
      nDequeuedPkts = 0 ;
      nDroppedPkts = 0 ;
      }
      {
```

```
      strEvent = $1 ;
      to_node = $7 ;
      pkttype = $9 ;
      fid = $17 ;
      pktsize = $11 ;

      if(strEvent == "r" && to_node == 3)
            {if(fid == 2 && pkttype == "tcp")
                  nReceivedPkts ++ ;}

      if(strEvent == "+" && to_node == 3)
            {if(fid == 2 && pkttype == "tcp")
                  nEnqueuedPkts ++ ; }

      if(strEvent == "-" && to_node == 3)
            {if(fid == 2 && pkttype == "tcp")
                  nDequeuedPkts ++ ; }

      if(strEvent == "d" && to_node == 3)
            {if(fid == 2 && pkttype == "tcp")
                  nDroppedPkts ++ ; }

      }

END {
      printf("\n Enqueued Pakets  =  %d\n Dequeued Pakets  =  %d\n
Received Packets =  %d\n  Dropped Pakets   =  %d\n",nEnqueuedPkts,
nDequeuedPkts, nReceivedPkts, nDroppedPkts);
      printf("Ratio Drop:Enqu  =  %2.4f\n",
nDroppedPkts/nEnqueuedPkts);
      printf("Throughput(pkts/s) = %6.4f \n",  nReceivedPkts/30);
    }
```

A.4 Dropped of Vegas

```
BEGIN {
      nReceivedPkts = 0 ;
      nEnqueuedPkts = 0 ;
      nDequeuedPkts = 0 ;
      nDroppedPkts = 0 ;
      }
      {
      strEvent = $1 ;
      to_node = $7 ;
      pkttype = $9 ;
      fid = $17 ;
      pktsize = $11 ;

      if(strEvent == "r" && to_node == 3)
            {if(fid == 3 && pkttype == "tcp")
                  nReceivedPkts ++ ;}

      if(strEvent == "+" && to_node == 3)
```

```
            {if(fid == 3 && pkttype == "tcp")
                  nEnqueuedPkts ++ ; }

      if(strEvent == "-" && to_node == 3)
            {if(fid == 3 && pkttype == "tcp")
                  nDequeuedPkts ++ ; }

      if(strEvent == "d" && to_node == 3)
            {if(fid == 3 && pkttype == "tcp")
                  nDroppedPkts ++ ; }

      }

END {
      printf("\n Enqueued Pakets  =  %d\n Dequeued Pakets  =  %d\n
Received Packets =  %d\n  Dropped Pakets   =  %d\n",nEnqueuedPkts,
nDequeuedPkts, nReceivedPkts, nDroppedPkts);
      printf("Ratio Drop:Enqu  =  %2.4f\n",
nDroppedPkts/nEnqueuedPkts);
      printf("Throughput(pkts/s) = %6.4f \n",  nReceivedPkts/ 30);

    }
```

A.5 Overhead (Ack)

```
BEGIN {
      nReceivedPkts = 0 ;
      nEnqueuedPkts = 0 ;
      nDequeuedPkts = 0 ;
      nDroppedPkts = 0 ;
      }
      {
      strEvent = $1 ;
      to_node = $7 ;
      pkttype = $9 ;
      fid = $17 ;
      pktsize = $11 ;


      {if(fid == 2 && pkttype == "ack")
            ack_reno ++ ; }

      {if(fid == 3 && pkttype == "ack")
            ack_vegas ++ ;}

      }

END {
      printf("\n Reno ACK  =  %d\n\n Vegas ACK =  %d\n\n" , ack_reno,
ack_vegas);
    }
```

Appendix B: Simulation TCL Script

```
# ======================================================================
# December 2007, Keria Wassie
# ======================================================================
#Create a simulator object
set ns [new Simulator]

#Open the a file for storing network animation
set nf [open simout.nam w]
$ns namtrace-all $nf

set file1   [open traceall.tr w]
$ns trace-all $file1

#open throughput measurement output files
set bw_file       [open bandwidth.tr w]
set wn_file       [open windowsze.tr w]
set qu_file       [open queuesze.tr w]
set dr_file       [open pktdropped.tr w]
set arriv_file    [open arrived.tr w]

#Define different colors for animation packet flows
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green

#Define a 'finish' procedure
proc finish {} {
      global ns nf file1 bw_file wn_file qu_file dr_file arriv_file
      $ns flush-trace

      #Close nam trace file
      close $nf

      #Close throughput measurement files
      close $file1
      close $bw_file
      close $wn_file
      close $qu_file
      close $dr_file
      close $arriv_file

      #Run network animation
        exec nam simout.nam &
}

proc record1 {} {
      global bw_file wn_file qu_file dr_file arriv_file qmon fmon
flowmon reno_tcp vegas_tcp

      #Get an instance of the simulator
      set ns [Simulator instance]

      #Set the time after which the procedure should be called again
      set time 0.04
```

```
      #Get the current time
      set now [$ns now]

      #How many bytes have been received by the traffic sinks?
      set pktarrived    [$flowmon set parrivals_]
      set pktdeparted   [$flowmon set pdepartures_]
      set pktdropped    [$flowmon set pdrops_]
      set flowclssfr    [$flowmon classifier]

      set reno_flow     [$flowclssfr lookup auto 0 0 2]
      set vegas_flow    [$flowclssfr lookup auto 0 0 3]

      #Calculate the throughput and write it to the files
      puts $wn_file "$now \t [$reno_tcp set cwnd_] \t [$vegas_tcp set
cwnd_]"
      puts $qu_file "$now \t [expr ($pktarrived - $pktdeparted -
$pktdropped) * (-1)]"
      puts $arriv_file "$now \t [expr ($pktarrived - 0)]"
      puts $dr_file "$now \t $pktdropped"

      #Reset the bytes_ values on the traffic sinks
      $flowmon set parrivals_ 0
      $flowmon set pdepartures_ 0
      $flowmon set pdrops_ 0

      #Re-schedule the procedure
      $ns at [expr $now+$time] "record1"
}

#Create nodes
#set n0   [$ns node]
set n0   [$ns node]
set n1   [$ns node]

set rtr1 [$ns node]
set n3 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $rtr1 4Mb 10ms DropTail
$ns duplex-link $n1 $rtr1 4Mb 10ms DropTail
$ns duplex-link $rtr1 $n3 1Mb 10ms DropTail

#limit queue size
$ns queue-limit $rtr1 $n3 2

#Flow Monitor
set linkrtr1n3 [$ns link $rtr1 $n3]

set flowmon [$ns makeflowmon Fid]
$ns attach-fmon $linkrtr1n3 $flowmon

#Monitor the queue for the link between Router1 and node 3
$ns duplex-link-op $rtr1 $n3 queuePos 0.5

#Setting toplogy layout
```

```
$ns duplex-link-op $n0 $rtr1 orient right-down
$ns duplex-link-op $n1 $rtr1 orient right-up
$ns duplex-link-op $rtr1 $n3 orient right

#TCP Reno traffic source
#Create a TCP agent and attach it to node n0
set reno_tcp [new Agent/TCP/Reno]
$ns attach-agent $n0 $reno_tcp
$reno_tcp set fid_ 2
$reno_tcp set class_ 2
$reno_tcp set window_ 40
$reno_tcp set packetSize_ 460

#Create a TCP sink agent and attach it to Router 2
set reno_sink [new Agent/TCPSink]
$ns attach-agent $n3 $reno_sink
#Connect both agents
$ns connect $reno_tcp $reno_sink

# create a Reno FTP source "application";
set reno_ftp [new Application/FTP]
$reno_ftp attach-agent $reno_tcp

#TCP Veags traffic source
#Create a TCP agent and attach it to node n1
set vegas_tcp [new Agent/TCP/Vegas]
$ns attach-agent $n1 $vegas_tcp
$vegas_tcp set fid_ 3
$vegas_tcp set class_ 3
$vegas_tcp set window_ 40
$vegas_tcp set packetSize_ 460

#Create a TCP sink agent and attach it to Router 2
set vegas_sink [new Agent/TCPSink]
$ns attach-agent $n3 $vegas_sink

#Connect both agents
$ns connect $vegas_tcp $vegas_sink

# create a Vegas FTP source "application";
set vegas_ftp [new Application/FTP]
$vegas_ftp attach-agent $vegas_tcp

#Schedule events for all the flows
$ns at 0.0 "record1"
$ns at 0.0 "$reno_ftp start"
$ns at 30.0 "$reno_ftp stop"
$ns at 0.0 "$vegas_ftp start"
$ns at 30.0 "$vegas_ftp stop"

#Call the finish procedure after 6 seconds of simulation time
$ns at 30.0 "finish"

#Run the simulation
$ns run
```

References:

[1] Bruce Barnett: "Awk - A Tutorial and Introduction,"
http://www.grymoire.com/Unix/Awk.html
Access date: 28 April, 2006

[2] Chuck Semeria, "Supporting Differentiated Service Classes: TCP Congestion Control Mechanisms," Juniper Networks, 2000.
Access date: 27, June 2006.

[3] G. Hasegawa, K. Kurata, and M. Murata, "Analysis and Improvement of Fairness between TCP Reno and Vegas for Deployment of TCP Vegas to the Internet," Proceedings of the IEEE International Conference on Network Protocols (ICNP), November 2000.

[4] Jae Chung and Mark Claypool, "NS by Example," Worcester Pllytechnic Institute, http://nile.wpi.edu/NS/,
Access date: 27, February 2006.

[5] Jeonghoon Mo, Richard J. La, Venkat Anantharam, and Jean Walrand, "Analysis and Comparison of TCP Reno and Vegas," University of California at Berkeley, July 13, 1998.

[6] Kenji Kurata, Go Hasegawa and Masayuki Murata, "Fairness Comparisons between TCP Reno and TCP Vegas for Future Deployment of TCP Vegas," 1999.

[7] Kenji Kurata, Go Hasegawa and Masayuki Murata, "Fairness Comparisons between TCP Reno and TCP Vegas for Future Deployment of TCP Vegas," July 2000.

[8] Lawrence S. Brakmo Sean W.O'Malley Larry L. Peterson: "TCP Vegas: New Techniques for Congestion Detection and Avoidance," in Proceedings of ACM SIGCOMM'94, pp. 24-35, October 1994.

[9] Lawrence S. Brakmo, Student Member, IEEE, and Larry L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet," IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, VOL. 13, No.8, pp.1465-1480, OCTOBER 1995.

Access date: 07, January, 2006.

[10] Leonardo Balliache, "Some TCP Related notes, 4.0. - TCP Vegas," http://www.opalsoft.net/qos/TCP-40.htm,
Access date: 15 July, 2006.

[11] M. Allman, V. Paxson and E. Blanton, "TCP Congestion Control," RFC 2581, June 2006.
Access date: 27, July 2006.

[12] Nauman Afzal: "Utilities of ns-2 network simulator,"
Access date: 28 April, 2006

[13] Richard J. La, Jean Walrand, and Venkat Anantharam, "Issues in TCP Vegas," http://citeseer.ist.psu.edu/la01issues.html,
Access date: May 15, 2006.

[14] Ruy de Oliveira and Paulo Roberto Guardieiro: "A Comparative Study of TCP Reno and TCP Vegas in a Differentiated Services Network,"
http://dragao.co.it.pt/conftele2001/proc/pap042.pdf
Access date: 01, January, 2007

[15] Van Jacobson and Michael J. Karels, "Congestion Avoidance and Control," November, 1988.
http://ee.lbl.gov/papers/congavoid.pdf,
Access date: May 26, 2006.

[16] W. Stevens, NOAO "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," RFC 2001, January 1997.
Access date: 20, May 2006.

[17] Yee-Ting, "TCP Tahoe, Reno, Vegas Tutorial" Yee's Homepage TCP/IP 2003,
Access date: 21, July 2006.

[18] The Network Simulator - ns-2, http://www.isi.edu/nsnam/ns,
Access date: 27, February 2006.

[19] Transmission Control Protocol (TCP)
http://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/tcp.html

[20] Transmission Control Protocol
http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[21] Understanding TCP/IP, Cisco Documentation: http://www.cisco.com/univercd/cc/td/doc/product/iaabu/centri4/user/scf4ap1.htm, Access date 10, June, 2006.