TCP performance study over wired and wireless networks

Zhang, Chunlei

2006

Zhang, C. (2006). TCP performance study over wired and wireless networks. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2573>

<https://doi.org/10.32657/10356/2573>

Nanyang Technological University

Downloaded on 26 Apr 2023 13:55:44 SGT

TCP Performance Study over Wired and Wireless Networks

Zhang Chunlei

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Master of Engineering

2006

TK
5105.585
Z63
2006

Acknowledgements

I would like to thank my supervisor, Dr Yap Ma Tit. I am very thankful to his support, guidance and patience. He has supervised me closely throughout this project according to the schedule, and has put in efforts in guiding me in all aspects.

I would like to thank Dr Fu Chengpeng, Franklin, for his guidance, advice and support in all aspects of the research work. His thorough advices help me clarify my doubts and solve various research problems encountered. He has taught me a lot about the many aspects of research. His inspiration will last far beyond the end of this project.

I would like to thank Dr Foh Chuan Heng for his review of my thesis and his innovative suggestions about my research work.

I also would like to thank my senior, Miss Guo Lijuan, and my group members, Zhang Ke, LV Qian, Yu Kuan and Zhou Bin. They gave me a lot of help. I am thankful to the technicians in the Digital Signal Processing Lab for their assistances in many aspects.

Finally, I would like to thank my family members; my mother Liu Guirong, my father Zhang Hanchen and my elder sister Zhang Chunli. In particular, without their encouragement and love, this thesis would not even be possible.

Table of Contents

Acknowledgements..... i

Table of Contents ii

List of Tables iv

List of Figures v

Summary vii

Chapter 1 1

Introduction..... 1

 1.1 A Bit of History 1

 1.2 Overview..... 2

 1.3 Motivations & Contributions 4

 1.4 Organization of Thesis..... 4

Chapter 2..... 5

Background..... 5

 2.1 Review of Transmission Control Protocol..... 5

 2.1.1 Slow Start and Congestion Avoidance 6

 2.1.2 Fast Retransmit and Fast Recovery..... 7

 2.2 Recovery Schemes 8

 2.2.1 NewReno..... 8

 2.2.2 SACK..... 9

 2.2.3 FACK..... 11

 2.3 TCP Veno Mechanism..... 13

 2.3.1 Veno Refined Multiplicative Decrease 14

 2.3.2 Veno Refined Additive Increase..... 15

Chapter 3..... 17

Performance Evaluation in Network Simulator..... 17

 3.1 Experimental Topology in Simulator..... 18

 3.2 Validation of TCP Veno 19

 3.2.1 Distinguishing between Types of Packet Loss 19

 3.2.2 Refined Additive Increase..... 22

 3.2.3 TCP Veno over wireless links..... 24

 3.3 Dynamics of TCP Veno and its Interaction with TCP Reno 25

 3.3.1 Fairness between Reno and Veno connections 30

 3.4 TCP Veno study with different recovery schemes 34

 3.4.1 SackVeno vs SackReno 35

 3.4.2 FackVeno vs FackReno 38

 3.4.3 NewVeno vs NewReno..... 40

 3.4.4 Compare Veno and Reno Variants between different recovery schemes.....43

 3.4.5 Impact of Normalized buffer 48

 3.4.6 Impact of propagation delay 49

 3.4.7 Fairness among multiple connections..... 50

3.5 Summary and Discussion.....	56
Chapter 4.....	57
Performance Evaluation in Testbed	57
4.1 Experimental Topology	57
4.1.1 Dummynet.....	58
4.2 Experiments Results.....	60
4.2.1 Single Connection	60
4.2.2 Multiple connections.....	66
Chapter 5	69
Implementations.....	69
5.1 TCP Veno Implementation in Network Simulator	70
5.1.1 State Estimation	70
5.1.2 Output Function	72
5.1.3 Recv Function	73
5.1.4 Window Adjustment	78
5.2 TCP Veno Modification in FreeBSD 4.3.....	81
5.2.1 Veno_send and Veno_receive functions.....	82
5.2.2 Calculation of Estimated Number of Packets in Bottleneck Buffer	87
5.2.3 Window Refinement	88
Chapter 6.....	93
Conclusions and Future Work	93
6.1 Summary	93
6.2 Contributions.....	94
6.3 Future work.....	95
6.3.1 Loss model development and mobility support.....	95
6.3.2 Veno over RED	95
Publications.....	97
References.....	98

List of Tables

Table 3.1: Performance Evaluation of Reno vs Veno, first connection is with
20Mbytes sent, second with 8Mbytes 34

Table 3.2 *FackVeno*, *SackVeno* and *NewVeno* congestion details in simulation..... 44

Table 4.1 Normalized Buffer Size 64

Table 4.2 Congestion loss vs Non-congestion loss of three Veno variants 66

List of Figures

Figure 2.1: Slow start and AIMD of TCP Congestion Control	7
Figure 2.2: The TCP SACK-permitted option.....	10
Figure 2.3: The TCP SACK option.....	10
Figure 2.4: FACK in non-recovery state.....	12
Figure 2.5: FACK in recovery state.....	12
Figure 3.1: Experimental Topology.....	19
Figure 3.2: Congestion window evolutions of TCP Reno when both random loss and congestion loss are present.....	20
Figure 3.3: Congestion window evolutions of TCP Veno when both random loss and congestion loss are present.....	21
Figure 3.4: TCP Reno behaviour over wired networks	23
Figure 3.5: The behaviour of TCP Veno over wired networks.....	23
Figure 3.6: TCP Veno and TCP Reno's throughput evaluation with varying random loss probability.....	25
Figure 3.7: (a) window evolution of two Reno connections; (b) window evolution of two Veno connections.....	27
Figure 3.8: TCP Veno running with TCP Reno connection.....	29
Figure 3.9: Window evolution of two Reno and Veno connections.....	30
Figure 3.10: Reno connection is started first, another Reno joins in at 20s	32
Figure 3.11: Veno connection is started first, Reno joins in at 20s.	32
Figure 3.12: Reno connection is started first, Veno joins in at 20s.	33
Figure 3.13: Throughput of TCP variants with different loss rates	35
Figure 3.14: Zoom figure of SackVeno sequence number and ACKs.	36
Figure 3.15: Zoom figure of SackReno.....	37
Figure 3.16: Zoom figure of FackVeno in recovery phase	38
Figure 3.17: Zoom figure of FackReno in recovery phase	40
Figure 3.18: Zoom figure of NewVeno in FF.....	41
Figure 3.19: Zoom figure of NewReno in FF.....	42
Figure 3.20: Sequence diagrams of three Veno variants	43
Figure 3.21: FackVeno congestion window evolution	45
Figure 3.22: SackVeno congestion window evolution.....	45
Figure 3.23: NewVeno congestion window evolution	46
Figure 3.24: Sequence diagrams of three Reno variants.....	47
Figure 3.25: Impact of Normalized Buffer	48
Figure 3.26: Impact of propagation delay in wireless environment	49
Figure 3.27: FackVeno and FackReno compete on the bottleneck.....	50
Figure 3.28: Two FackReno connections compete on the bottleneck.	51
Figure 3.29: Two FackVeno connections compete on the bottleneck.	51
Figure 3.30: (a) NewVeno and NewReno; (b) Two NewReno; (c) Two NewVeno...	53
Figure 3.31: (a) SackVeno and SackReno; (b) Two SackReno; (c) Two SackVeno.	55

Figure 4.1: Experiment Topoloy in Testbed 58

Figure 4.2: a) *NewVeno* congestion window evolution b) *NewReno* congestion
window evolution over wireless link 61

Figure 4.3: TCP Variants over different loss rate 63

Figure 4.4: Normalized buffer effect of TCP variants 64

Figure 4.5: Impart of propagation delay over TCP Variants 65

Figure 4.6: (a) sequence diagram of FackVeno with FackVeno (b) sequence
diagram of FackVeno and FackReno 67

Figure 4.7: (a) Sequence number of *FackReno* and *FackReno* with 1% random loss
rate (b) Sequence number of *FackVeno* and *FackReno* with 1% random loss
rate 68

Figure 5.1: *v_state* evolution of TCP Veno connection 71

Figure 5.2: *recv()* function flow chart 75

Figure 5.3: TCP Veno modifications overview 82

Figure 5.4: Send a segment in TCP Veno *output* function 84

Figure 5.5: ACK processing in TCP Veno input function 86

Summary

Internet suffer a few of severe “network collapses” due to lack of advanced congestion control mechanism in TCP. Until 1988 a modern TCP version, called TCP Tahoe, incorporates congestion control algorithms over wire line networks were developed by Van Jacobson in Lawrence Lab, Berkeley. Two years later, an enhanced version was born, referred to as TCP Reno. However, TCP Reno has performance problems when multiple packets are dropped from the same window of data. These problems result from the need to wait the packet retransmit timer expiration before continuing data flow. So there is a need to refine the fast recovery and fast retransmit scheme of TCP and this area is intensively studied by Sally Ford. And finally in 1996 she published TCP selective acknowledgment (SACK) and *NewReno* modification to TCP fast recovery scheme when SACK option is not available. Simulation and experiments shows that SACK can significantly improve TCP’s performance when burst losses are present. In the same year 1996, Matthew Mathis and Jamshid Madhdavi proposed TCP with Forward Acknowledgment (FACK), FACK is based on SACK and performs more precise control by making use of the forward-most data during recovery. While coming to the end of 1990s, wireless networks have been a hot area and have obtained huge deployment, diverse wireless links are widely deployed and becoming main access networks connected to Internet, however, TCP suffering come up again. Different from its last time suffering because of lack of congestion control mechanisms, this time its significant performance degradation is due to the lack of capability to distinguish random loss caused by bit error from congestion loss. Up to 2001, a new TCP version called TCP Veno is proposed, which is designed with advanced packet loss distinguishing scheme and a refined congestion control. Today TCP is most complicated protocol and its embedded advanced congestion control mechanism is formulating a foundation for 4th generation communication, in which wireless is an integral part.

In this thesis, we give an comprehensive study over the dynamic window evolutions of TCP Veno with the aid of network simulator. Since after 1996, researchers have intensively focused on solving TCP Reno’s suffering in wireless networks and less attention has been put on studying the performance of

fast recovery schemes in wireless environment. For this thesis, different recovery schemes interact with TCP Veno and TCP Reno are also examined in wireless environment by means of simulator and testbed. A few of novel and interesting experimental results are observed and analyzed.

Chapter 1

Introduction

1.1 A Bit of History

The first reference to TCP was in a 1973 review note, entitled “A Partial Specification of an Internet Transmission Protocol” by Dr. Vinton G. Cerf to a few colleagues. The note made a case for avoiding reassembly in network computers (routers). These were the early days of networking. The term *router* had not been coined back then. The concept of layering was not very clear. Most of the networking tasks were handled by Network Control Protocol (NCP).

In 1974, Dr. Vinton G. Cerf and Dr. Robert Kahn published a paper entitled “A Protocol for Packet Network Interconnection”. They provided a detailed discussion of several design choices in TCP [3]. The first official specification of TCP was written by Vint Cerf, Yogen Dalal, and Carl Sunshine [2]. The second version of TCP specification was written by Cerf in March 1977 [4]. Two months later, Jon Postel observed that Internetwork communication should be viewed as

having two components: the hop-by-hop relaying of messages and the end-to-end control of the conversation [5]. In January of 1978, Cerf and Postel split TCP into two protocols: TCP and IP [6]. IP specified routing packets while TCP handled packetization, error control, retransmissions, and reassembly. They argued that simplicity of IP would allow fast and inexpensive gateways to be built. Postel then wrote a new specification for IP and TCP. The specification was discussed in several IEN meetings and finally published in [7].

In February of 1980, the U.S. Department of Defense (DoD) adopted TCP/IP as the preferred protocol. DoD required that every site connected to Advanced Research Projects Agency (ARAP) net should switch to TCP/IP by 1983. In 1984, the network of networks so connected was later split into MILNET for military-related sites and the regular Internet for other sites. In the 1980s, there were many other networking protocol stacks that competed with TCP/IP. For example, Digital Equipment Corporation (DEC) net, Systems Network Architecture (SNA), AppleTalk, Xerox Network Systems (XNS). Most of these protocols were vendor-specific and proprietary. It was hoped that someday they would all be replaced by a single standard protocol designed by the International Standards Organization (ISO). The knowledge gained from TCP/IP development helped in the development of ISO's transport protocols, resulting in two varieties of network layer protocols (connectionless and connection-oriented). All this took too long and became too complicated. By this time, TCP/IP had been widely adopted because of its simplicity and the fact that it was an open protocol. Not only are all protocol specifications openly available, but the source code is available. It is much like UNIX or LINUX. In fact, the implementation of TCP/IP in Berkeley UNIX 4.2 BSD distribution in 1983 may be one reason for the wide adoption of TCP/IP.

1.2 Overview

The refinement on TCP and its performance studies never stop during the TCP evolution. One of the most researched aspects of TCP performance related problem is congestion and flow control.

Early TCP implementations only had a simple sliding window flow control

mechanism without any congestion control. After observing a series of congestion collapses in the 1980s, Jacobson [8] introduced several innovative congestion control mechanisms into TCP in 1988. This TCP version, known as TCP Tahoe, includes the slow-start, additive increase and multiplicative decrease (AIMD), and fast retransmit algorithms. Two years later, the fast recovery algorithm was added to Tahoe to form a new TCP version called TCP Reno. TCP Reno is currently the dominating TCP version deployed in the Internet.

Following Jacobson's pioneering work in 1988, considerable research has been carried out. Many researchers attempted to propose an "ideal" congestion control mechanism to combat TCP's inherent oscillation problem resulting from its additive increase and multiplicative decrease algorithms [8]. These works include CARD [9], Tri-S [10], Packet-Pair [11], TCP Vegas [12, 13, 14, 15], and CCA in TCP Santa Cruz [16]. However, from the practical deployment point of view, the performance of these congestion control mechanisms over the heterogeneous networks do not compare well with TCP Reno. They may even have certain negative impacts [17].

During the middle periods of 1990s, most researchers turned to refining the recovery algorithm of TCP Reno after packet loss is detected. New proposals included TCP *NewReno* [26], SACK [19, 20]. All these algorithms are refinement and enhancement to TCP Reno and not intended to replace TCP Reno.

With the advent of computing and communication technologies in the late 1990s, it is feasible to interconnect wired networks to wireless networks (e.g. Cellular IP network, Wireless LAN network). The traditional assumption that packet loss is an indicator of network congestion in wired network may be invalid in wireless networks. This is because bit errors could be the main reason behind packet loss. TCP Reno is equally sensitive to both network congestion and random loss. This sensitivity may lead to significant but unnecessary end-to-end throughput degradation. As a result, a new variant of end-to-end congestion control mechanism, called TCP Veno [24], was proposed. TCP Veno estimates the available bandwidth in order to discriminate the network state: congestive state or non-congestive state. It sets the slow start threshold (*ssthresh*) when packet loss is detected by three duplicate ACKs. In addition, TCP Veno refines the additive increase algorithm of TCP Reno. Previous experiments have shown that TCP Veno not only achieves significant throughput improvement over TCP Reno,

particularly in wireless networks, but also shows good compatibility and flexibility with TCP Reno.

1.3 Motivations & Contributions

TCP Veno is deployed with a congestion monitoring scheme. The scheme can differentiate the type of losses. TCP Veno has refined the AIMD from TCP Reno. In our opinion, the window dynamics of TCP Veno has not been studied extensively, especially related to interactions with TCP Reno connections. This thesis focuses on the detailed studies of window dynamics of TCP Veno by means of simulation. To facilitate the investigations, a TCP Veno module is implemented in network simulator (ns-2.26) from Lawrence Berkeley National Laboratory.

Our research is also motivated by the fact that multiple or bursty loss in wireless and mobile networks can cause serious performance degradation in TCP. Several recovery schemes, *NewReno*, SACK, FACK, have been proposed to solve this problem. This thesis also presents a detailed study of TCP Veno over different recovery schemes. The study is carried out in simulation and testbed.

1.4 Organization of Thesis

Chapter 2 provides some background knowledge of TCP. The emphasis is on the congestion control algorithms and recovery schemes. Extensive experiments to evaluate the performance of TCP Veno, by means of network simulation, are described in chapter 3. Performance evaluation experiments by means of testbed are presented in chapter 4. The results are analyzed and compared against those obtained by simulation. In chapter 5, the detailed implementation of TCP Veno, in both simulator and FreeBSD 4.3 kernel, are presented. Finally, chapter 6 summarizes the main contributions of this thesis and proposes several future research directions.

Chapter 2

Background

2.1 Review of Transmission Control Protocol

A best-effort network, such as the Internet, does not have the notion of admission control or resource reservation in order to control the imposed network load. To ensure network stability, network end-points sharing a best-effort network need to respond to congestion by implementing congestion control. Without congestion control, a network may be driven into congestion collapse; network load stays extremely high but throughput is reduced to close to zero. The purpose of an end-to-end congestion control algorithm is to reduce the rate at which a sender injects data into the network when congestion is detected, and to increase the rate when the network bandwidth is not fully utilized. Over the past decade, congestion control algorithms of TCP Reno have become the pseudo standard.

TCP Reno, a widely deployed end-to-end congestion control mechanism in the Internet, employs slow start, congestion avoidance, fast retransmit and fast recovery algorithm to adjust the sending window size. These algorithms will be reviewed in the following subsections.

2.1.1 Slow Start and Congestion Avoidance

When a connection is first established, a sender does not have much information about the network traffic condition. TCP uses a technique called slow start to probe the available bandwidth of the network. The sender sets congestion window (*cwnd*) to one segment (or packet), begins to transmit one segment, and waits for its acknowledgement (ACK). Each time an ACK is received, *cwnd* is increased by one. When the first ACK is received, the sender increases *cwnd* to two and transmits the next two segments. *cwnd* would be increased in an exponential manner during the slow start phase. The slow start phase ends when a segment loss is detected or when the congestion window reaches the slow-start threshold (*ssthresh*). When the slow start threshold is exceeded, the sender is in the congestion avoidance phase. The sender increases the congestion window roughly by one segment per RTT. *cwnd* grows in a linear manner. In TCP, a segment loss is taken as a sign of congestion. A loss is detected: 1) by duplicate ACKs; or 2) by timeout. When a segment loss is detected, the slow start threshold is set to half of the current *cwnd*. TCP enters the fast recovery phase. After the loss is recovered, the *cwnd* is set to *ssthresh*. The additive increase algorithm is restarted. However, if the loss is detected by timeout, which means that the corresponding retransmission timer for the loss packet expires. Because of the lack of ACK to bring TCP out of recovery phase, *cwnd* is set to one. The timeout action normally means that the network is in serious congestion. The congestion window is set to one segment. The sender proceeds with the slow start algorithm.

Figure 2.1: Slow start and AIMD of TCP Congestion Control

Figure 2.1 shows a typical behaviour of the congestion window of a TCP connection. The mechanism is commonly called additive increase and multiplicative decrease (AIMD).

2.1.2 Fast Retransmit and Fast Recovery

The fast retransmit algorithm is simple to understand. A packet may arrive out of order. Datagram networks, such as the Internet, may re-order or duplicate packets. TCP systems generate duplicate acknowledgments (DUPACKs) for every out-of-order segment received. A threshold of three DUPACKs was chosen as a compromise between the danger of spurious loss detection and a timely loss recovery. Upon the reception of three DUPACKs, the fast retransmit algorithm is triggered (*ssthresh* is set to half of the current *cwnd*). The “fast recovery” algorithm governs the transmission of data until a non-duplicate ACK arrives. Usually the fast retransmit and fast recovery algorithms are implemented together as following:

- Retransmit the lost segment and set *cwnd* to *ssthresh* plus three. This artificially inflates the congestion window by the number of segments (three) that have left the network.
- For each additional duplicated ACK received, increase *cwnd* by one. This artificially inflates the congestion window. Each duplicated ACK means one

segment has left the network.

- Transmit a segment if allowed by the new *cwnd* and the receiver's advertised window.

When an ACK arrives that acknowledges new data, this is the end of fast recovery phase. The AI (additive increase) algorithm takes over starting from the latest *ssthresh*.

2.2 Recovery Schemes

TCP uses a cumulative acknowledgment scheme. Received segments that are not at the left edge of the receive window are not acknowledged. Multiple packet losses from a window of data can cause catastrophic effect on TCP throughput. The default Reno recovery scheme will cut window by half for each of the loss. This results in a small *cwnd*. As a consequence, TCP would lose its ACK-based clock. It has to wait for packet retransmit timer to expire. The overall throughput is significantly reduced. To overcome the problem, Selective Acknowledgment (SACK) is proposed to correct this behaviour in the face of multiple dropped segments. *NewReno* is recommended when the SACK option is absent. Forward Acknowledgment (FACK), based on SACK option, is later proposed. SACK option attains more precise control over the data flow in the network during data recovery phase.

2.2.1 NewReno

The *NewReno* [18, 26] recovery scheme is a small but significant modification of Reno's FF (fast retransmit and fast recovery schemes). It aims to eliminate the performance degradation caused by multiple losses in a single congestion window. For example, *cwnd* is 16 but segments 2, 5, 7, 13 are lost in transmission. FF is hardly able to recover. It has to wait until timeout. In this case, the acknowledgements of *retransmitted* segments are very important information for the sender. They carry new information rather than the duplicate ACKs. In our example, ACKs for retransmitted segment 5 means retransmitted segments 1 to 4 are successively received. ACK for retransmitted segment 7 actually acknowledges segments 1 to 6. Such ACKs acknowledge some but not all of the

segments transmitted before the Fast Retransmit. They are known as “partial acknowledgements”. Based on the account of partial acknowledgements, *NewReno* do the following work:

- 1) When the third duplicate ACK is received and the sender is not already in the fast recovery procedure, set *ssthresh* to $cwnd/2$, and record the highest sequence number transmitted in the variable “recover”.
- 2) Retransmit the lost segment and set *cwnd* to *ssthresh* plus $3 \times MSS$ (Maximum Segment Size). This artificially inflates the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.
- 3) For each additional duplicate ACK received, increase *cwnd* by MSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
- 4) Transmit a segment if allowed by the new value of *cwnd* and the receiver’s advertised window.
- 5) When an ACK arrives that acknowledges new data, then i) if this ACK acknowledges all of the data up to and including “recover”, set *cwnd* to *ssthresh* (the value of step 1) and exit the recovery procedure. Otherwise, ii) this is a partial ACK. Immediately retransmit the first unacknowledged segment, deflate the *cwnd* by the amount of new data acknowledged, and send new segments if permitted by the new value of *cwnd*. Do not exit the recovery procedure.

2.2.2 SACK

SACK (Selective ACK) [19,20] is another solution to deal with the multiple-loss problem in FF. By making use of the SACK option in the TCP header, the receiver could inform the sender which segments are lost. The sender can retransmit and recover even in one single RTT.

When a TCP connection is first established between a sender and a

receiver, the sender uses SYN, which contains the “SACK-permitted” option, to indicate that the SACK option is enabled. Figure 2.2 shows the TCP SACK-permitted option format.

Kind = 4	Length = 2
----------	------------

Figure 2.2: The TCP SACK-permitted option.

If the receiver accepts this option, SACK scheme is enabled. Whenever a segment is lost in transmission, resulting in the receiver receiving non-continuous data blocks, the receiver sends duplicate ACKs, containing SACK option, to inform the sender about the lost segments. Figure 2.3 shows the SACK option format. Note that the SACK option does not change the meaning of the Acknowledge Number field.

	Kind =5	Length
Left Edge of 1 st Block		
Right Edge of 1 st Block		
.		
.		
.		
Left Edge of nth Block		
Right Edge of nth Block		

Figure 2.3: The TCP SACK option.

SACK option contains a list of blocks of contiguous sequence space occupied by data that have been received and queued within the receiver’s window. Each contiguous block is defined by two 32-bit unsigned integers in network byte order. “Left edge of block” is the first sequence number of this block. “Right edge of block” is the sequence number immediately following the last

sequence number of this block. A SACK option that specifies n blocks will have a length of $8*n+2$ bytes. The 40 bytes available for TCP option can specify a maximum of 4 blocks. In addition, if the Timestamp option is used, a maximum of 3 blocks will be allowed.

When the sender enters recovery procedure by three DUPACKs, SACK scheme requires that the sender keeps a “pipe” variable” to estimate the amount of outstanding segments. The initial value of *pipe* is ($cwnd - 3$). The DUPACK segment is then transmitted. The congestion window is halved to $cwnd/2$. On receiving a DUPACK resulting *pipe* minus one and receiving a partial ACK resulting *pipe* minus two. Partial ACK only represents one segment having left the network. However, *pipe* was incremented when a retransmitted segment is sent out, but was never decremented for the segment assumed to have been lost. When the partial ACK arrives, it implies that two packets have left the network. They are the original packet (assumed to have been lost) and the retransmitted packet. New or retransmitted segments can be sent only if *pipe* is less than $cwnd$. Transmitting one segment will result in *pipe* plus one. The sender could intelligently retransmit the lost segments as soon as possible.

2.2.3 FACK

The FACK (Forward ACK) [29] is also based on the TCP SACK option. It performs more precise control by making use of the forward-most data during recovery. The word “forward-most” refers to the correctly received data with the highest sequence number. As FACK keeps an accurate estimate of the amount of outstanding segments in recovery, it seems that FACK is more efficient in face of multiple losses.

FACK uses variables *snd.fack* to represent the forward-most segment, *snd.una* to represent the first unacknowledged segment, *snd.nxt* to represent the next new segment to be sent, and the *retran_data* to represent the number of retransmitted segment in the network. In the non-lost and non-recovery state, the *snd.fack* must be equal to *snd.una*, and less than *snd.nxt*. Every time, *snd.fack* is updated by acknowledgement number in the TCP header from receiver. Figure 2.4 shows an example, where the sender records that forward-most segment is 10 and

the next segment to sent is 14. Segments 11 to 13 have not been acknowledged.

Figure 2.4: FACK in non-recovery state.

In the recovery state, *snd.ack* must be less than *snd.fack*. The SACK option from receiver reports non-contiguous blocks. In this case, *snd.ack* conforms to ACK number in the TCP header. *snd.fack* is updated by the SACK option. Figure 2.5 shows the example as above except that segments 4 and 8 are lost. We can see that the receiver returns an ACK for segment 4. It reports two non-contiguous blocks. *snd.fack* is set to 11 but *snd.una* becomes 4. In addition, *snd.nxt* is still 14.

Figure 2.5: FACK in recovery state.

FACK defines *awnd* to be the sender's estimate of the actual quantity of data outstanding in the network.

$$awnd = snd.nxt - snd.fack + retran_data$$

Note that *retran_data* reflects the amount of the retransmitted segment in

the network. Each time a segment is retransmitted, the *retran_data* is increased by one segment size. When a retransmitted segment is determined to have left the network, the *retran_data* is decreased by one segment size.

As long as *awnd* is less than *cwnd*, the sender can inject additional segments into the network. Normally, the sender could choose to send new data or retransmit lost segments reported in the SACK option. If new data is sent, FACK advances *snd.next*. If a lost segment is retransmitted, FACK increases *retran_data*. Correspondingly, ACK from receiver will either decrease *retran_data* or increase *snd.fack*. This guarantees FACK's robustness.

2.3 TCP Veno Mechanism

TCP Reno has enjoyed much success to-date. However, it is increasingly being stretched by the emergence of heterogeneity in the Internet. The assumption that packet loss is an indicator of network congestion may not apply to heterogeneous network. Packet loss may be induced by noise, link error or reasons other than network congestion. Such packet loss is known as random loss.

To consider the random loss impact on TCP performance, a new end-to-end congestion control mechanism, suitable for networks with heterogeneous properties, called TCP Veno[24] is proposed in 2001. It only requires modification to the TCP sender. TCP Veno is therefore interoperable with existing TCP. Deployment can be achieved by having only one end of the connection employs TCP Veno. The algorithm differs from the conventional TCP in two ways:

- 1) It dynamically adjusts the slow start threshold (*ssthresh*) according to the perceived state of a connection – congestive or non-congestive, as opposed to using a fixed drop-factor (e.g. $\frac{1}{2}$ in Reno, $\frac{3}{4}$ in Vegas) when packet loss is encountered.
- 2) It uses a refined linear increase algorithm to adjust the congestion window evolution for congestion avoidance.

Both network testbed experiments and live Internet measurements results show that in a typical wireless access network with 1% random packet loss rate, throughput improvement of up to 80% can be achieved. Moreover, Veno can achieve significant throughput improvements without adversely affecting other concurrent TCP Reno connections.

TCP Veno only refines the AIMD mechanism of Reno. All other parts of Reno, including initial slow start, fast retransmit, fast recovery, computation of the retransmission timeout and back-off algorithm remain intact. Veno AI and MD will be introduced in the following section.

2.3.1 Veno Refined Multiplicative Decrease

Veno makes use of a mechanism similar to that in Vegas [25] to estimate the state of the connection. Nonetheless, such a scheme is used to deduce what kind of packet loss – congestion loss or random loss – is most likely to have occurred, rather than to pursue the prevention of packet loss as in Vegas.

Recall that, in Vegas, the sender measures the *Expected* and *Actual* rates as:

$$\begin{aligned} \textit{Expected} &= \textit{cwnd}/\textit{BaseRTT} \\ \textit{Actual} &= \textit{cwnd}/\textit{RTT} \end{aligned}$$

Where *cwnd* is the current TCP window size, *BaseRTT* is the minimum of measured round-trip time, and *RTT* is the smoothed round-trip time measured. The difference of the rate is:

$$\textit{Diff} = \textit{Expected} - \textit{Actual}$$

When $\textit{RTT} > \textit{BaseRTT}$, there is bottleneck link where the packets of the connection accumulate. Let the backlog at the queue be denoted by *N*. We have:

$$\textit{RTT} = \textit{BaseRTT} + N/\textit{Actual}$$

That is, we attribute the extra delay to the bottleneck link in the second term of the right side above. Rearranging, we have:

$$N = Actual * (RTT - BaseRTT) = Diff * BaseRTT$$

Veno uses the measurement of N as an indication of whether the connection is in a congestive state when packet loss occurs. Specifically, when the sender receives 3 duplicate ACKs for segment sequence i , it knows that the i th segment is lost. Then, the sender do the following steps to multiplicatively decrease the $cwnd$, retransmit, and recover:

- 1) If ($N < \beta$) */*random loss due to bit error is most likely to have occurred*/*
 $ssthresh = cwnd * (4/5);$
 Else $ssthresh = cwnd/2;$ */*congestive loss is most likely to have occurred*/*
- 2) Each time another dup ACK arrives, increase $cwnd$ by one segment.
- 3) When the next ACK acknowledging new data arrives, set $cwnd$ to $ssthresh$ (value in step 1)

In other words, if the connection is not in the congestive state (i.e. if $N < \beta$), Veno assumes that the loss is random. Accordingly, it reduces $ssthresh$. Hence the $cwnd$, by a smaller amount with factor of 4/5. Otherwise, (i.e. $N \geq \beta$), Veno sets $ssthresh$ as one-half of current $cwnd$ and hence halves the $cwnd$, as in Reno.

It is worth pointing out that if the packet loss is not notified by duplicate ACKs, but rather the timeout, Veno will initiate the slow start algorithm again with $ssthresh$ set to $cwnd/2$ and $cwnd$ reset to one. This is not modified as that in Reno. Veno takes the timer expiration as an indication of severe congestion.

2.3.2 Veno Refined Additive Increase

Veno also refines the additive increase phase of Reno by forcing the TCP connection to stay longer at the operating region in which the bandwidth is fully utilized. It reduces self-induced congestion loss in Reno. The Veno AI is listed as follows:

If ($Diff * BaseRTT < \beta$) */*available bandwidth under-utilized*/*

```

    cwnd=cwnd+1/cwnd    /* when every new ACK received */
Else if ( $Diff * BaseRTT \geq \beta$ ) /*available bandwidth fully utilized*/
    cwnd = cwnd+1/cwnd    /* when ever other new ACK received */

```

The first part is the same as the algorithm in Reno. In the second part, the backlog packets (N) in the buffer exceed β , and Veno increases the window size by one for every two round-trip times. One may ask, “Why more conservative window-increase policy is used during the linear increase phase?” The reason can be specified as follows: firstly, referring to Vegas’ interpretation, when $Diff * BaseRTT = \beta$, it implies that the available bandwidth is utilized. At the same time, the network is trapped into a state in which congestion is impending (there are β packets at the bottleneck buffer). Therefore, Veno decreases window growth rate in order to leave more spaces for other joining connections. Even though there is no other traffic joining, such a refined additive increase algorithm defers the early onslaught of self-induced congestion loss. It forces a TCP connection to stay at the equilibrium region longer without under-utilizing the bottleneck link (because all the available bandwidths have been utilized at this time). Secondly, as Reno has been widely deployed in the current Internet, any modified TCP version must conform to the specifications defined in RFC 793, RFC 1122 and RFC 2914. Otherwise, it is difficult to be deployed into the current Internet. Veno continues to increase window when the imminent congestion is monitored but with a slower growth speed than Reno’s. Obviously it complies with Reno and satisfies the rule addressed in RFC standards.

Chapter 3

Performance Evaluation in Network Simulator

In this chapter, we present the performance evaluation of TCP Veno by means of simulation. A TCP Veno module is developed for the ns-2.26 network simulator. The TCP Veno's implementation includes various features, such as the measurement of packet's round trip time (RTT), the estimation of the number of packets accumulated in the bottleneck link buffer, and the refined congestion control algorithm. More details are illustrated in Chapter 5.

In order to make a detailed study on the dynamics of TCP Veno congestion window evolution, we choose to implement the TCP module in ns-2.26. The ns-2 simulator provides event-driven and monitoring environment for studying the congestion control algorithm employed by TCP Veno. In contrast, experiments may be quite difficult to be conducted in testbed by tools such as *trpt* to trace

incoming and outgoing segments in TCP kernel space which may result in failing to log some events because of system high utilization, priorities or other real-time constraints. In addition, attempting to synchronize the timing of each system clock is another serious issue when multiple TCP connections' performance are evaluated on different workstations. In ns-2, there is no need to worry about these issues, the ns-2 simulator provides a unique system clock. It is possible to monitor every event of interest within the simulator for subsequent analysis. Thus, ns-2 provides an ideal experimental environment for our research.

In Section 3.1 the network topology used for our experiments is firstly introduced and followed by the TCP Reno's validation test in Section 3.2. The dynamics of TCP Reno's window evolution are intensively studied in Section 3.3. Finally, the TCP Reno's performance with various recovery schemes in wireless environment are examined in Section 3.4.

3.1 Experimental Topology in Simulator

Figure 3.1 depicts the network topology used in our simulation. The network consists of two pairs of source and destination nodes. The two pairs of connections share the same bottleneck link which uses a drop-tail queue. The speed of each link, that connects the two nodes and routers, is 10Mbps with 1ms propagation delay. The bottleneck link connecting routers A and B has the throughput of μ bits/s and τ seconds of propagation delay. The queue size along this link is B packets for both forward and reverse directions. Typical data packet size is 1Kbytes by default unless otherwise specified. Typical ACK size is 40 Bytes.

Similar to [30], we define the capacity of the path to be the maximum number of packets (data packets and acknowledgement packets) outstanding along this path. These outstanding packets, data or ACKs, may spread along the transmission line or queue at the buffer of the bottleneck router. The capacity between source and destination is $B+2P$ where $P=\mu*(\tau+1\text{ms}+1\text{ms})/(8*\text{packet size}) \approx \mu*\tau/(8*\text{packet size})$. Besides, we assume that the source has infinite data to send and the receiver's advertised window is set sufficiently large enough.

Figure 3.1: Experimental Topology.

3.2 Validation of TCP Veno

3.2.1 Distinguishing between Types of Packet Loss

There exist two types of packet loss in heterogeneous networks. One is called congestion loss which occurs when the buffer of the bottleneck link is full. Another is called random loss which is caused by bit errors or other non-congestive reasons.

A good congestion control mechanism would increase its sending rate to make use of the available bandwidth when it detects the network is not fully utilized. On the other hand, it would try to reduce its sending rate when it senses that the network is congested and try to alleviate network congestion situation. Traditional congestion control algorithm, such as Reno, blindly cuts sending rate by half when loss occurs, without differentiating whether this loss is caused by congestion or random bit errors. This blind behaviour will result in network underutilized especially when random bit errors are commonly present as in wireless networks.

The link pipe size is always defined as bandwidth delay product ($BDP = 2P$) which is the product of the link bandwidth and the round-trip propagation delay. When the amount of outstanding packets in the network is equal to BDP, the link is fully utilized and almost no extra queuing delay is encountered by data packets. If the amount of outstanding data is smaller than BDP, there will be no accumulating packets in the bottleneck link buffer and the network is not fully

utilized. Increasing congestion window to send more data will make use of the available bandwidth and increase the throughput. However, when the amount of outstanding data segments reaches the amount of BDP packets, the queue on the bottleneck link starts to accumulate packets. Data packets will suffer from extra queuing delay. Although, the link is still fully utilized and does not have buffer overflowing yet, the network is trapped into a congestive state. TCP Reno attempts to find the congestion state of network by measuring this value using $Diff * BaseRTT$, which is regarded as the number of packets accumulated in the buffer of bottleneck link. When $Diff * BaseRTT < \beta$, Reno regards the network is under-fully utilized and currently in a non-congestive state with available bandwidth unused. Otherwise, if $Diff * BaseRTT \geq \beta$ Reno observes the network is in a congestive state and with available bandwidth fully utilized. β is set to 3 in TCP Reno for conservative estimation.

Figure 3.2: Congestion window evolutions of TCP Reno when both random loss and congestion loss are present.

Figure 3.3: Congestion window evolutions of TCP Veno when both random loss and congestion loss are present.

In this section of experiments, we artificially introduce three random losses with sequence numbers 1810, 1899 and 2340. The BDP value of simulated network link is 23 packets. In Figure 3.2, TCP Reno has four congestion losses C1, C2, C3 and C4 and three random losses R1, R2 and R3. We mark congestion loss as C# and random loss as R# respectively. Since TCP Reno does not differentiate the types of packet loss, the congestion window is always halved. Specifically, the congestion window reaches 6 after continuous window cuts for random loss R1 and R2.

In Figure 3.3, there are three congestion losses C1, C2, C3 and three random losses R1, R2, R3. R1 and R2 are treated as random loss which are correct decisions and the window is cut only 1/5 instead of halved because TCP Veno is in a non-congestive state. It is true that the network has not reached the BDP point yet. The window after deductions of random losses R1 and R2 is 15; this is much better than TCP Reno (6 after deductions). Referring to Figure 3.3, there is one random loss, R3, which is misinterpreted as congestion loss since at that time the TCP Veno has already entered into a congestive state. Therefore the window is

halved. However, when random loss R3 occurs, all the available bandwidth has been utilized. The network is approaching congestion point. In some sense, quick escape from the network congestion state is not a bad choice to avoid impending congestion loss.

3.2.2 Refined Additive Increase

Depending on the congestive state or non-congestive state TCP Veno is currently in, $cwnd$ can be increased by one segment for every RTT if the estimated number of buffered packets is less than 3, recalled that:

If $(Diff * BaseRTT < \beta)$ */*available bandwidth under-utilized*/*
 $cwnd = cwnd + 1/cwnd$ when every new ACK received

However, if more than 3 packets have been detected to be accumulated in the bottleneck buffer, TCP Veno will try to slow down the speed by increasing congestion window by 1 for every other RTT rather than every RTT .

Else if $(Diff * BaseRTT \geq \beta)$ */*available bandwidth fully utilized*/*
 $cwnd = cwnd + 1/cwnd$ when ever other new ACK received

By this refined additive increase mechanism, TCP Veno attempts to stay longer in link full utilization region. Here the full utilization region is referred to as the region in which all the available bandwidth has been fully utilized and there is no buffer overflow.

Figure 3.4: TCP Reno behaviour over wired networks

Figure 3.5: The behaviour of TCP Veno over wired networks

Figures 3.4 and 3.5 show the evolutions of both TCP Reno and TCP Veno in a non-lossy network environment. The line marked BDP separates the network into over-utilized and under-utilized regions. For this demonstration, the BDP value is 23 packets. The dotted line in Figure 3.5 is equal to $BDP + 3$. The threshold, β , is defined as 3 packets. It divides the connection states into congestive state and non-congestive state. In the congestive state, TCP Veno tries to slow down the *cwnd* increasing speed because it has already 3 packets accumulated in the bottleneck link buffer. An increase in *cwnd* will not likely to increase the throughput, since there is no more available bandwidth left. This refined AI algorithm may create two effects.

Firstly, the refined AI algorithm defers the onset of buffer overflow. It reduces the number of congestion losses. As we can observe from Figures 3.4 and 3.5 during the 20 to 40 seconds, there are in total about 8 AIMD cycles for TCP Reno connection, in which exactly 8 congestion losses occur. TCP Veno connection only has incurred 6 congestion losses. In a wired environment, the additive increase algorithm that actually dominates the most part of TCP window evolution, the refined AI algorithm, may help TCP Veno to gain more throughput and link utilization. During 20 to 40 seconds, TCP Reno increases its sequence number from 3150 to 6812 with throughput of 183.1 packet/s, while TCP Veno sends out a total of 3790 packets, starting from sequence 3264 to 6973, with a bit higher throughput of 185.5 packet/s.

Secondly, the refined additive increase algorithm introduces out-of-phase synchronization of TCP window evolution when TCP Veno interacts with TCP Reno connections. This will be studied in details in Section 3.3.

3.2.3 TCP Veno over wireless links

TCP Veno can distinguish between types of packet loss, and prevent unnecessary window deductions when the packet losses are caused by random bit errors other than network congestion. The following experiments make use of the network topology in Figure 3.1 with settings as: $\mu = 1.2\text{Mbps}$, $\tau = 60\text{ms}$ and $B = 16$ packet size for both forward and reverse directions.

Figure 3.6: TCP Veno and TCP Reno's throughput evaluation with varying random loss probability.

Figure 3.6 shows the throughput comparison of TCP Reno and TCP Veno under different packet loss rate (ranging from 10^{-5} to 10^{-1}) in wireless links. It is observed that the throughput of TCP Veno is always higher than that of TCP Reno. This figure clearly shows that TCP Veno has significant improvement over Reno especially when the random loss rate is 1%. With loss rate increased further to 10%, both TCP Veno and TCP Reno's throughput are quite similar. This is because under high loss rate, both TCP Veno and Reno experience frequent timeouts, and difficulty in maintaining self-clocking.

3.3 Dynamics of TCP Veno and its Interaction with TCP Reno

In this section of experiments, two TCP Reno or Veno connections are tested between the pairs of source and destination, as depicted in Figure 3.1. The following configuration is used: $B = 15$, $\mu = 1.6\text{Mbps}$ and $\tau = 50\text{ms}$. In [30,31] an epoch is defined as the time period during which an entire window's worth of packets have been acknowledged. Those special epochs in which packet losses

occur are referred to as congestion epochs. As explained in [30, 31], two Renos have in-phase synchronization paces. One connection drops its window by half while the other connection will follow the same window-penalty immediately. This is depicted in Figure 3.2(a). The two connections enter a congestion epoch whenever the total window size, the sum of the two connections' values for *cwnd*, reaches the capacity of the path. Each connection loses exactly one packet in each congestion epoch.

The in-phase synchronization can be explained as follows: It is known that during the additive increase phase, TCP Reno increases *cwnd* by one packet for each window of received ACKs. In other words, at the end of current epoch, the congestion window would increase by one and one more packet would be sent out during next epoch. This scenario continues until the link's capacity is reached. At the congestion epoch, when the extra packet reaches the router, there is no room to hold this packet since at this point of time the buffer of the router has been full. The extra packet will be dropped. The link capacity becomes full for all the connections at the same time. Every Reno connection would have increased *cwnd* by one packet at the congestion epoch. They would all have a single packet dropped. After that the *cwnd* will be halved for all connections.

Similarly, two Veno connections are simulated in the network. The same configuration is used. We might expect that there is no change in its behaviour over wired networks. However, as depicted in Figure 3.2(b), it can be seen that this intuitive expectation is invalid. Veno's window demonstrates out-of-phase synchronization, which is quite different from Reno's in-phase synchronization paces [30, 31]. With reference to the point, epoch 1 in Figure 3.2(b), when the link capacity is reached, one Veno connection loses one packet while the other connection does not incur any loss, and continues its window evolution. This out-of-phase synchronization is desirable since the connection which does not incur loss can still maintain a high congestion window and make use of the link's available bandwidth.

Figure 3.7: (a) window evolution of two Reno connections; (b) window evolution of two Veno connections.

The reason that resulted in such difference from Reno connections is due to the refined additive increase algorithm of TCP Veno. At epoch 1, both Veno connections have been trapped into a congestive state. They are taking more conservative increase than Reno. The window ($cwnd$) is increased by one packet for every two round-trip times. One connection sends one extra packet at the beginning of epoch 1, and the extra packet makes the buffer full. While, the other connection also increases its congestion window and one extra packet is sent out. Because the only one available packet space has been occupied by its counterpart,

the extra packet would get dropped. Although the buffer keeps full for two epochs, the connection would not increase its congestion window for the next epoch. So this connection escapes from congestion epoch 1. After that these two connections will continue to increase until the sum of two connections' window reaches the path capacity again, as seen in the congestion epoch 2 depicted in Figure 3.7(b). At this point of time, each of the two connections has a packet loss, which is the same as the two Reno's behaviour. The difference is that the connection with lower congestion window has less window reduction while the connection with the larger window takes severe window-halving penalty. The reason is that the former connection is evolving into a non-congestive state (with one packet increasing by one round trip time), and the latter one keeps its congestive state. While two connections are competing for network resource, the connection with more resource occupying, will suffer severe penalty. The connection with less resource occupying will have less severe window reduction. The weaker connection will have a chance to grab more resources when loss occurs. Meanwhile, the stronger connection will release some resources it occupied. The interesting observation is that their roles will be reversed in the next cycle. This kind of mechanism will definitely bring about fair competition for limited resources over the Internet. From the congestion epochs 1 and 2, this intertwined pattern repeats itself indefinitely. Only a few cyclical periods of window evolutions are shown in the figure.

Reno TCP has been widely deployed in practical networks. A Veno connection usually has to co-exist with Reno connections. In the following section, we study their co-existence. We experiment with one Reno and one Veno connection using the same setting. As addressed in [30], here we also notice that the packets from different sources are never intermixed together. Each connection sends out a whole window of packets in one burst and waits for the next epoch. The data packets within one window are sent in a batch manner. No spacing gaps are introduced between consecutive packets. The ACKs, which are returned for a window of data packets from the same source, always arrive at the sender with spacing equal to the transmission time of a data packet at the bottleneck link. Referring to the following Figure 3.8, we observe that TCP Veno is partially (or out-of-phase) synchronized with TCP Reno.

Figure 3.8: TCP Veno running with TCP Reno connection.

A Reno connection always increases its window size by one. In a non-congestive state, Veno behaves the same way as Reno. In a congestive state, Veno increases its window size by 1 every other round trip time. Let us consider the epoch that the path is just one packet less than full capacity ($C - 1$). The Veno connection in congestive state happens to have increased its *cwnd* by one for this epoch. It sends the whole window of packets first (the extra packet from Veno makes the path full). The Reno connection then starts to send out its window of packets. The path is full and Reno increases its *cwnd* by one. There will be one extra packet from Reno dropped. However, when the window of ACKs from the Veno connection returns, it will begin the next epoch. Since it has increased *cwnd* during the previous epoch, it will not increase its *cwnd* for this time. The window of packets will still keep the path full but without packet loss. As seen in the congestion epoch 3, the Veno connection does not incur any loss while Reno does. When the Reno connection detects a loss, it will reduce its *cwnd* by half. In the following epochs, the path's capacity will not be full any more. Thus, Veno will not incur any loss (although its window would increase for this epoch) until in the future when the path capacity is full again. See the congestion epoch 4 in Figure 3.8. At epoch 4, both Reno and Veno suffer a packet loss due to their simultaneous window increase. Both windows are cut into halves. Nonetheless, in the following window evolution phase (from 47.5s to 53s), Reno will gradually exceed Veno's window and grab more bandwidth resource because of its aggressiveness. At time 53, it will repeat the case of "epoch 3". This *intertwined* pattern repeats itself

indefinitely. Over the long run, fairness between Veno and Reno can be maintained. Among these two competing connections, Veno has suffered less packet congestion losses (9 times in Figure 3.8), while Reno has more (13 times). This reduced congestion number helps to increase bandwidth utilization. It is proven in the following experiments done in Section 3.3.1.

Figure 3.9: Window evolution of two Reno and Veno connections.

A more interesting dynamic window evolution scenario with two Veno and two Reno connections is shown in Figure 3.9. The experiment is done with settings of $B = 44$, $\mu = 1.8\text{Mbps}$ and $\tau = 50\text{ms}$. The events on “epoch 5” and “epoch 6” are similar to that of “epoch 4” in Figure 3.8. At “epoch 5”, only Veno 1 does not incur loss. At “epoch 6”, both Veno 1 and Veno 2 connections do not incur loss. Spreading of a series of synchronization congestion epochs in which all Reno and Veno connections lose exactly one packet during each epoch. The Reno connections gradually grab more bandwidth because of its aggressive window policy in Additive Increase phase. After that the *intertwined* pattern is started (at 274s) to repeat with veno1 and veno2 interchanging their roles. We notice that window evolution becomes more complicated with increasing number of connections, especially when external packets are introduced (e.g. background traffic is presented). Sometimes Reno connection does not incur any loss while Veno does, which is also an out-of-phase synchronization scenario.

3.3.1 Fairness between Reno and Veno connections

In this subsection, we investigate the fairness issue of TCP Reno and Veno over wired networks.

It can be observed that two Reno connections always have synchronized in-phase paces. When a Veno connection co-exists with a Reno connection, there are some occasions that Veno does not incur packet loss in a specific epoch. To some extent, this behaviour will avoid some congestion loss, and may improve the utilization of network link. We design our experiments as follows: either the Reno or the Veno connection starts first to transmit a 20 Mbyte file. The other connection -- Reno or Veno -- joins in at time 20s and begins to transfer a 8 Mbyte file. The setting of the network topology is: transmission rate of 2Mbps, a round-trip delay of 100ms and the bottleneck link buffer size of 15. We plot the congestion window as well as the sequence number against time in Figures 3.10, 3.11 and 3.12 respectively. From these three different scenarios, we observe that later coming connection would not be biased by previous running connection. They will share the bandwidth evenly. This can also be seen in each diagram of sequence number vs time, where the slope of two connections are almost the same.

Figure 3.10: Reno connection is started first, another Reno joins in at 20s

Figure 3.11: Veno connection is started first, Reno joins in at 20s.

Figure 3.12: Reno connection is started first, Veno joins in at 20s.

In Table 3.1, we calculate the throughput using the file size plus the total retransmitted bytes divided by time in seconds. From this table, we obtain three results; 1) as for longer transmission, Veno achieves slightly higher throughput regardless of whatever combination is employed; 2) as for short transmission, Veno also achieves slightly higher throughput; 3) referring to row 3 of this table, we know that this slight improvement is brought about by less congestion losses in Veno as compared to Reno. These results conform to our above analysis. At the congestion epochs that Reno always incurs packet loss while Veno may not. This kind of out-of-phase synchronization leads to the efficient utilization of network link.

	Reno/Reno	Reno/Veno	Veno/Reno
Time Transmitted	124.66/73.52	124.28/71.47	123.89/73.73
Retransmitted packets	105/56	103/45	90/54
Throughput (Kbytes/s)	161.27/109.51	161.76/112.56	162.4/109.85

Table 3.1: Performance Evaluation of Reno vs Veno, first connection is with 20Mbytes sent, second with 8Mbytes

3.4 TCP Veno study with different recovery schemes

In this section we are going to investigate the TCP Veno’s interactions with different recovery schemes. The experimental network topology used is the same shown in Figure 3.1. The link between router A and router B is the bottleneck with bandwidth of 1.2 Mbps, one-way propagation delay of 60 ms, and buffer size of 16 for both directions. Note that all the links in the experimental network are symmetrical.

We run each of the six TCP variants (*NewVeno*, *SackVeno*, *FackVeno*, *NewReno*, *SackReno*, and *FackReno*) on the source for a period of 200 seconds under different loss rates, ranging from 10^{-5} to 10^{-1} . We calculate the throughput of each connection. Combining the six diagrams, the result is shown in Figure 3.13, where Veno variants are drawn in solid lines while Reno variants in dotted lines.

Figure 3.13: Throughput of TCP variants with different loss rates

We can see that along the loss rate ranges, all Reno connections' throughput gained are close to each other while the Veno connections' throughput are separate (especially at random 1% loss rate). The three lines representing Veno variants are all above the lines representing Reno variants. This observation is quite novel and representative. It says that the different fast recovery schemes have no effect on Reno connections but they can distinguish Veno connections. Most importantly, no matter what recovery scheme is employed, Veno always outperforms Reno. The improvement is remarkable at 1% loss rate.

3.4.1 SackVeno vs SackReno

In order to point out where the performance improvement is gained from, we trace the detailed behaviour of the senders. Here, we illustrate the comparison results of *SackVeno* versus *SackReno*. Figure 3.14 is the zoomed-in figure of the *SackVeno* sender. Figure 3.15 is the zoomed-in figure of the *SackReno* sender. In these figures, the vertical axis is the sequence number. The horizontal axis is time. If the sender sends out a segment with sequence number Y at time X , we plot a point at position (X, Y) using “.” sign. If the sender receives an ACK, we also plot it with “+” sign in the figure.

First, let's investigate the behaviour of *SackVeno* sender. We pick the

particular time when the sender's *cwnd* is 16 and continuously introduce four artificial losses: segments 3590, 3592, 3594, 3596. After a short period, three DUPACKs are returned. The sender is triggered into a recovery phase, which is represented in the figure using solid rectangular.

Figure 3.14: Zoom figure of SackVeno sequence number and ACKs.

At the beginning of the recovery phase, the *SackVeno* sender is in non-congestive state thus judges the segment loss as a random loss and this is what we expected. Therefore, the sender cuts only 1/5 of the *cwnd*, which results in the *cwnd* becoming 12 and retransmits the first lost segment 3590 immediately. The pipe size is 14 ($16-3+1$) now. After that, the sender waits for three more DUPACKs, deflating the pipe again to 11, to retransmit the second lost segment 3592; the pipe is then increased again to 12. After that, two more DUPACKs trigger the sender to send the third and fourth lost segments 3594 and 3596 respectively.

After all of the lost segments are retransmitted, in the recovery phase the sender will attempt to send some new segments if there are enough ACKs returned. We can see the following two DUPACKs trigger the sender to send two new segments, and thereafter, comes three non-dup ACKs which are partial ACKs.

On receipt of each partial ACK, the sender decreases the pipe size by two, and thus can transmit two new segments. Therefore, after the ACK for segment 3604 takes the sender out of recovery phase, the sender has totally transmitted 8 new segments.

Next, let's investigate the behavior of *SackReno* in the recovery phase. Likewise, we choose a particular time when the sender's *cwnd* is 16 and continuously introduce 4 artificial segment losses: 3642, 3644, 3646, and 3648. After a short while, three DUPACKs return and trigger the sender entering recovery phase, which is represented in the following figure in the solid rectangular.

Figure 3.15: Zoom figure of *SackReno*.

At the beginning of the recovery phase, the sender blindly cuts half of the *cwnd*, resulting in *cwnd* becoming 8 and pipe size being 13, although it is caused by a random loss and congestion has not happened yet. Then the sender immediately retransmits the first lost segment 3642 and pipe is increased to 14. After that, the sender must wait for seven DUPACKs to decrease pipe to 7 and then to retransmit the second lost segment 3644. Then, the following DUPACKs and partial ACKs will allow the sender to retransmit the third and fourth lost

segments and five new segments.

Now we define two useful notations: X and Y . X denotes the number of DUPACKs the sender must wait for after the first retransmission and before the retransmission for the second lost packet. Y denotes the number of new segments transmitted in the recovery phase. Therefore, X is 3 for *SackVeno*, and 7 for *SackReno*; Y is 8 for *SackVeno*, and only 5 for *SackReno*. By intuition, if the X is small and Y is large, the sender in the recovery phase can wait for a short time to complete retransmission and send more new data, and hence achieve higher throughput.

3.4.2 FackVeno vs FackReno

Figure 3.16: Zoom figure of *FackVeno* in recovery phase

Now let's have a look at the recovery phase of *FackVeno* connection in Figure 3.16. *FackVeno* keeps a precise estimate of the amount of outstanding segments in the network using *acwd*, which is calculated by $snd.next - sand.fack + retran_data$.

Packets with sequence numbers 2851, 2853, 2855 and 2857 have been artificially dropped. All of these four losses are in the same window of data segments with a congestion window size of 16. Now *FackVeno* is in a non-congestive state. When the second DUPACK acknowledging packet 2850 arrives, *FackVeno* would have *snd.nxt* equal to 2867, *snd.una* equal to 2851 and *snd.fack* equal to 2855. The condition $snd.fack - snd.una$ greater than 3 is satisfied and the FF is triggered. Otherwise we have to wait for the third DUPACK to trigger FF. The congestion window is then cut by 1/5 and becomes 12. Immediately *FackVeno* retransmits packet 2851 and sets *retran_data* is equal to 1. When the fourth DUPACK indicating that segment 2856 has arrived at receiver, *snd.fack* is then forward to 2857. Because *awnd* is 11 now ($2867 - 2857 + 1$) which allows one more packet to be sent out. And *FackVeno* chooses the missing segment in the sack holes instead of new packets and increases *retran_data* to 2. The fifth DUPACK arrived indicates segment 2858 has arrived at the receiver side and now *snd.fack* is equal to 2859 and *awnd* is 10 ($2867 - 2859 + 2$) and two more packets can be sent out. Again *FackVeno* chooses to send lost segments. After that, one more DUPACK will forward *snd.fack* by one and a new packet can be sent. Although partial acknowledgement for lost segments does not forward *snd.fack*, it decreases *retran_data* by one and more new packets can be transmitted. In the fast recovery scheme, a total of 12 new packets has been sent out. So in *FackVeno* X is 1 and Y is equal to 12, which is even more efficient than *SackVeno* where X is 3 and Y is 8.

Now let's look at what happens in the recovery phase of *FackReno* in Figure 3.17. The burst loss of packets with sequence numbers 2783, 2785, 2787 and 2789 occurs when the current congestion window is 16. After the congestion window deduction the *cwnd* becomes 8 and the *snd.nxt* is 2800. When the second DUPACK returns which indicates packet 2786 has been successfully received and now *snd.fack* is equal to 2787 and *snd.una* is 2783, the *FackReno* will immediately retransmit the first loss packet 2783 since the condition for FF is satisfied ($snd.fack - snd.una > 3$). After that the *retran_data* is set to 1, it has to wait for 6 more packets when the DUPACK corresponding to reception of packet 2893 arrives, *FackReno* can send one more packet since *awnd* would be 7 ($2800 - 2894 + 1$). After that DUPACKs and partial ACKs will trigger transmission of lost segments and 7 new more packets.

Figure 3.17: Zoom figure of *FackReno* in recovery phase

3.4.3 NewVeno vs NewReno

Figure 3.18 is the zoom figure of *NewVeno* sender. We choose a particular time when the sender *cwnd* is 16 and continuously introduce four losses: 3651, 3653, 3655, and 3657. On receipt of three DUPACKs, the sender enters recovery phase. At the beginning of the recovery phase, the sender senses that the network is in a non-congestive state and cut the *cwnd* by 1/5, which results in *cwnd* becoming 12. This is the same as that in *SackVeno* and *FackVeno*. But the difference is what happens in the recovery phase. Unlike *SackVeno* and *FackVeno*, *NewVeno* uses four RTTs to retransmit the lost segments, each represented by a rectangular in Figure 3.18.

In the first RTT, the sender retransmits segment 3651, and set the usable window to 15 (*cwnd* + 3). At this moment, there are 18 unacknowledged outstanding segments in flight, so the sender must wait for 3 more DUPACKs to inflate usable window to 18. After that, the following DUPACKs trigger the sender to transmit 8 new segments. In the second RTT, the sender receives the partial ACK for segment 3653; it immediately retransmits segment 3653 and

deflates usable window to 12. Then, the sender receives 8 DUPACKs to inflate usable window to 20, but now there are 24 outstanding unacknowledged segments, so it cannot transmit any new data.

Figure 3.18: Zoom figure of *NewVeno* in FF.

In the third and fourth RTT, the sender receives the partial ACKs for 3655 and 3657 segments, and retransmits them respectively. After that, ACK for segment 3676 brings the sender out of recovery phase. On receipt of this ACK, the sender transmits a burst of 12 segments because the *cwnd* is 12 and the amount of outstanding segments suddenly becomes zero.

The details of *NewReno* recovery scheme is shown in Figure 3.18. The random loss occurs at sequence numbers 2725, 2727, 2729 and 2731, all within one congestion window of 16 segments.

When the third DUPACK arrived, the fast retransmit and fast recovery scheme is triggered. The *cwnd* is halved and the useable window is set to 11 ($cwnd + 3$). At this point in time, there are a total of 18 outstanding packets in the pipe ($snd.nxt - snd.una$). *snd.nxt* is 2743 and *snd.una* is 2725. *NewReno* must wait for 8 extra DUPACKs to enlarge useable window to 19 to allow to transmit new data segments. We notice that *NewReno* transmits four new packets by the last

four DUPACKs in the first RTT. During the second RTT, when the partial ACK for retransmit packet 2725 returns, the loss packet 2727 is immediately retransmitted and deflates the usable window to 8.

Figure 3.19: Zoom figure of *NewReno* in FF

After that the returning of the four DUPACKs corresponding to the new packets sent out in the first RTT inflates the usable window to 12. However, we now have total 22 outstanding packets, so no data would be transmitted. In the third and fourth RTT, the sender receives the partial ACK for 2729 and 2731 respectively. Eventually the ACK for 2746 brings the sender out of recover phase and a burst of 8 segments would be sent out since *cwnd* is 8 now and without any outstanding segments.

3.4.4 Compare Veno and Reno Variants between different recovery schemes

From the zoomed-in figures of *FackVeno*, *FackReno*, *SackVeno*, *SackReno*, *NewVeno* and *NewReno*, we can clearly see that with the same recovery scheme deployed, TCP Veno variants can achieve higher performance compared with their Reno counterpart in terms of time to wait to retransmit the second lost packet and number of new packets sent during recovery phase due to its less congestion window deduction based on the congestion monitoring scheme. Besides when connection comes out of recovery phase, the higher congestion windows with TCP Veno variants would result in more throughput achievement because they help to get a higher start point in AI phase. And that explains why Veno variants' throughput are all above Reno's.

Further, we plot three Veno variants in Figure 3.20. And it is the same experiments done in Figure 3.13 with 1% random loss rate.

Figure 3.20: Sequence diagrams of three Veno variants

From this figure, the most distinguishing fact we observed is that different recovery schemes separate the performance of Veno TCP variants. We notice that

SackVeno and *FackVeno* are closer in terms of throughput gained (the slope of sequence diagram), and are separated with *NewVeno* connection. Now lets look at the detailed performance statistics, presented in the following table.

	Timeout	FF	non-congestion loss	congestion loss	no. of retrans pkt
FackV	3	200	150	50	269
SackV	2	186	150	36	250
NewV	7	172	136	36	821

Table 3.2 *FackVeno*, *SackVeno* and *NewVeno* congestion details in simulation

According to Table 3.2, we can observe that *FackVeno* has the most number of “congestion loss” (actually they are random loss, since with 1% loss rate, there are rarely cases that the congestion window can reach $BDP+B$ value). This means *FackVeno* has most opportunities to stay above the $BDP + 3$ value and incurs random losses. However, *NewVeno*, *SackVeno* have less congestion losses and are expected to have a smaller average congestion window compared to *FackVeno*. Another observation from this table is that *NewVeno* incurs more timeouts than its SACK and FACK counterparts. The timeout in wireless environment can be caused by two scenarios. Firstly, when three DUPACKs are detected, the connection triggers fast retransmit algorithm to retransmit the lost packet. There is a probability that the retransmitted packet is also corrupted and got dropped. For this situation the connection is determined to incur timeout. Secondly, when new packets sent in the previous recovery phase have incurred loss, the connection would result in a small congestion window. Any loss of packets or returning ACKs, makes the DUPACKs number less than three which would never trigger FF to retransmit the loss packet and in this situation , there is still a need of timeout to break the deadlock and bring the connection back to slow start phase. And that explains why *FackVeno* has achieved the most throughput

while *NewVeno* is the lowest. Further let's observe the detail congestion window evolution of this three Veno variants in the following figures.

Figure 3.21: *FackVeno* congestion window evolution

Figure 3.22: *SackVeno* congestion window evolution

Figure 3.23: *NewVeno* congestion window evolution

The BDP value is 18 packets. Observing from these figures, *FackVeno*, *SackVeno*, and *NewVeno* all have their average congestion window size oscillating around the BDP value over the entire period. The X and Y values we defined earlier can be used to explain why *FackVeno*, *SackVeno* and *NewVeno* perform differently. However, this is only true when some conditions are satisfied. The condition is that the TCP connection must have the ability to maintain a high congestion window even in presence of random loss. As we know, the *FackVeno* has the ability to trigger loss packet retransmission even without the need to wait for the third DUPACK. It can recover from random loss faster and maintain the congestion window at a higher level. That is why it achieves the highest throughput. However, when the loss rate increases to 0.1, *FackVeno* shows no difference compared to *SackVeno* and *NewVeno*, since none of them has the ability to maintain a higher congestion window (around BDP value). This also explains the scenario shown in Figure 3.24 below.

Figure 3.24: Sequence diagrams of three Reno variants

Figure 3.24 shows the sequence diagram of TCP Reno connections with different recovery schemes deployed. These three Reno variants do not differentiate between congestion loss and random loss. They have a severe window deduction whenever a loss is encountered. All of them do not have the ability to maintain the congestion window level around network equilibrium point (around BDP value). Despite their different recovery schemes deployed there is no much difference between their throughputs.

So our conclusion is that although different recovery schemes are not the key factor to affect the performance of TCP connections in wireless environment, it can create differences when the TCP connection has the ability to differentiate congestion loss and random loss and maintain the congestion window around BDP value even in presence of high random bit errors.

3.4.5 Impact of Normalized buffer

Figure 3.25: Impact of Normalized Buffer

The normalized buffer size is the ratio of the bottleneck buffer size to BDP value. As defined in [36], in most of the cases we use the normalized buffer size to differentiate between LAN and WAN. In this experiment, the impact of different value of normalized buffer size is investigated. The network configuration is the same as in Figure 3.1 with $\mu = 1.2\text{Mbit/s}$ and one way propagation delay $\tau = 60\text{ms}$ and the varying normalized buffer size from 0.1 to 3. The experiment is conducted under 1% random loss rate. We run each of them independently and plot the combined results in Figure 3.25. It is obvious that all Veno variants gain higher throughput than their Reno counterpart, as we expected in the presence of random losses. Varying from LAN to WAN with corresponding normalized buffer greater than one to less than one, there is no much changes in Veno variants TCP, except in extreme cases (less than 0.1). So we can conclude that different values of normalized buffer size do not significantly impact the throughput of Veno and Reno TCP variants. *FackVeno* still gains the highest throughput among all the circumstances. This is due to its efficient recovery schemes, especially in wireless environments.

3.4.6 Impact of propagation delay

Figure 3.26: Impact of propagation delay in wireless environment

The experiment in this section focuses on flexibility of various TCP variants over the impact of propagation delay. In our test, we set round trip delays to 60ms, 120ms, 180ms, 240ms and 300ms, with corresponding bottleneck buffer sizes of 7, 14, 20, 27, and 33 respectively. The normalized buffer for all experimental cases would be around 0.73. All other experimental parameters are configured as same as before.

As seen from Figure 3.26, the throughput drops roughly in linear manner for all Reno and Veno variants. As we have expected, all TCP Veno variants' throughputs are higher than all TCP Renos'. All TCP Reno variants stick together and their throughputs are mostly determined by their serious window cut behaviours and relatively more timeouts. While TCP Veno variants' throughput have some differences with the same AIMD and the same congestion monitoring schemes deployed. The recovery schemes in this situation play a role in influencing the performance. One main purpose of SACK design is to improve the efficiency of TCP's recovery scheme in slow links where the propagation delay is considerably long. Normal TCP only can recovery only one packet loss in each round trip time and SACK options can facilitate the process in recovery phase by

recovering multiple loss in one RTT time. As we would expect, *SackVeno* and *FackVeno* have the best performances with long propagation delay.

3.4.7 Fairness among multiple connections

Experiments with multiple connections are conducted in network topology, depicted in Figure 3.1. The source node 1 (src1) wants to send infinite data to destination 1 (dst1), and the source node 2 (src2) wants to send infinite data to destination 2 (dst2). Therefore, two TCP connections are competing on the bottleneck. Other parameters are the same as experiments on single connection with $\mu = 1.2\text{Mbps}$, $\tau = 60\text{ms}$ and 16 for bottleneck link buffer size. And 1% random loss is artificially introduced into the bottleneck link.

We first run *FackVeno* on source 1 and *FackReno* on source 2. Figure 3.27 shows the sequence number diagram of the two connections within 200 seconds. *FackVeno* exhibits remarkable higher throughput than co-existing *FackReno*.

Figure 3.27: *FackVeno* and *FackReno* compete on the bottleneck.

One may ask, “whether the throughput improvement is gained by grabbing bandwidth from the co-existing connection?” To answer this question, we run two *FackRenos* on sources 1 and 2 respectively. Figure 3.28 shows the sequence number evolution of two *FackReno* connections. We can see that the *FackReno* connection in Figure 3.27 has the same throughput as that in Figure 3.28, where *FackVeno* is absent. It proves that the Veno connection does not achieve its

superior throughput by aggressively grabbing bandwidth away from Reno connection, but rather it makes better use of the available bandwidth.

Figure 3.28: Two *FackReno* connections compete on the bottleneck.

When Reno connections run on lossy link, some available bandwidth is left unused, but Reno is unaware of this. In other words, Reno wastes some of the available bandwidth. If we use Veno to replace the existing Reno implementations, we can achieve higher total throughput. To verify this, we run two *FackVeno* on source 1 and 2 respectively, and plot the sequence number diagram of two connections in Figure 3.29.

Figure 3.29: Two *FackVeno* connections compete on the bottleneck.

The first observation is the total throughput are higher compared to Figure 3.28. The total throughput improvement can be viewed as the sum of individual throughput improvement gained by each Veno connection over Reno. The second observation is fairness. Two lines representing *FackVeno* connections twist with each other, indicating the fair share of the bottleneck.

We further extend the experiment to other recovery schemes. The results are shows in Figure 3.30 and 3.31, where we can see very similar cases as what we observe in FACK. Compare Figure 3.30 (c) and (b), two *NewVeno* connections share the link in a fair manner and achieve better total throughput than two *NewReno* connections. Compare Figure 3.31(c) and (b), the case is the same for two *SackVeno* connections.

(a)

Figure 3.30: (a) *NewVeno* and *NewReno*; (b) Two *NewReno*; (c) Two *NewVeno*.

(a)

(b)

(c)

Figure 3.31: (a) *SackVeno* and *SackReno*; (b) Two *SackReno*; (c) Two *SackVeno*.

Now we compare Figure 3.30(c) and Figure 3.31(c), two *SackVeno* versus two *NewVeno*. Clearly, the two *SackVeno* have higher total throughput. This improvement is gained in the recovery procedures by the Selective ACK scheme of each *SackVeno* connections. With the same AIMD of TCP Veno, recovery scheme again becomes the factor that determines the TCP performance.

Then we compare Figure 3.30(b) and Figure 3.31(b), two *SackReno* versus two *NewReno*. Very interesting, two *SackReno* connections have the same total throughput as that of two *NewReno* connections. It verifies our conclusion that, in the wireless environment, recovery schemes become useless due to Reno's blind window-cut behavior that heavily degenerates the TCP performance, however, they create performance difference for Veno variants because TCP Veno and maintain a higher congestion window even in presence of high random bit errors.

3.5 Summary and Discussion

The TCP Veno's dynamics are intensively studied when the TCP Veno module is developed in the simulator. A desired out of phase synchronization on window evolution of TCP Veno and Reno has been observed, this could lead to better link utilization and later experiment results have proved this. With typical settings single TCP Veno and single TCP Reno connection can co-exist very well

In simulator, the performance of TCP Veno with different recovery schemes over wireless links has been examined. There are three main observations: 1) No matter what recovery schemes is employed, Veno outperforms Reno. 2) Veno over Fack, which is referred as *FackVeno* has the best performance. 3) In the wireless environment, recovery schemes become useless for Reno, due to Reno's blind window-cut behavior heavily degenerates the TCP performance, however, they create performance differences among Veno Variants since TCP Veno can maintain a high congestion window even in presence of random bit errors.

Chapter 4

Performance Evaluation in Testbed

In order to conduct performance evaluation of TCP VenO over a testbed, several versions of TCP VenO have been implemented in FreeBSD 4.3 kernel. They are VenO, *NewVenO*, *SackVenO* and *FackVenO*. In this chapter, we are going to look at TCP VenO performance in *Dummynet* testbed. We would verify and compare the evaluation results obtained from testbed with those obtained in network simulator.

4.1 Experimental Topology

Figure 4.1 shows our experimental network topology. All experiments reported in this chapter are conducted using this network topology. Src1 ... SrcN are the TCP senders, including TCP VenO or TCP Reno of FreeBSD 4.3, and Dst1...DstN are the TCP receivers by using Linux Fedora.

Figure 4.1: Experiment Topology in Testbed

The links are labeled with their bandwidth and delay. Both sender and receiver's buffer size are set to 64 Kbytes to make the advertised window large enough. The packet size used in all experiments is 1024 bytes by default unless otherwise specified. A drop-tail router is set up by using *Dummynet* embedded in FreeBSD 4.3. We can use IPFW [34, 35] command to configure forward / backward buffer size B_f / B_r , forward / backward bandwidth BW_f / BWr , forward / backward propagation delay D_f / D_r , and forward / backward packet drop rate L_f / L_r .

4.1.1 Dummynet

Dummynet [35] is a flexible tool with minimal modifications to an existing protocol stack. It allows experiments to be run on a standalone system. *Dummynet* works by intercepting communications of the protocol layer under test. It simulates the effects of finite queues, bandwidth limitations, communication delays and multi-path effects. It runs in a fully operational system. *Dummynet* facilitates the use of real traffic generators and protocol implementations, while solving the problem of simulating unusual situations. More importantly, *Dummynet* is targeted to TCP. Normally

Dummynet specifies a path's properly by defining pipes. The objects of pipe can be configured separately by *ipfw* commands of FreeBSD. By using these commands, different limitations (e.g. delay, bandwidth and loss rate) can be applied to different traffics traversing over *Dummynet* routers.

In the experiments, we set up a full-duplex communication path. The path acts as a bottleneck when traffic flows from *Src* to *Dst*. The path is set up with the following list of *ipfw* commands.

```
ipfw flush
ipfw add pipe 1 ip from Src_ip to Dst_ip out
ipfw add pipe 2 ip from Dst_ip to Src_ip in
ipfw pipe 1 config bw  $BW_f$  queue  $B_f$  plr  $L_f$  delay  $D_f$ 
ipfw pipe 2 config bw  $BW_r$  queue  $B_r$  plr  $L_r$  delay  $D_r$ 
```

“ipfw flush” clears all the previous settings. A forward channel pipe 1 that allows IP packet flowing from the sender with IP address *Src_ip* to the receiver with IP address *Dst_ip* is added. In order to allow full duplex communication, another channel pipe 2 that allows traffic flowing from the receiver with IP address *Dst_ip* to the sender with IP address *Src_ip* is also added. The in and out parameters specify the direction of the pipe filter. Finally, both forward and backward channels are configured. The related parameters, such as bandwidth (BW_f and BW_r), buffer size (B_f and B_r), propagation delay (D_f and D_r) and random packet loss rate (L_f and L_r) are set.

For instance, we want to set the path with bandwidth of 1Mbps, buffer size of 12 packets, and one way propagation delay of 60ms with random packet loss rate of 1%. The corresponding *ipfw* commands look like this:

```
ipfw pipe 1 config bw 1Mbit/s queue 12 plr 0.01 delay 60ms
ipfw pipe 2 config bw 1Mbit/s queue 12 plr 0.01 delay 60ms
```

4.2 Experiments Results

4.2.1 Single Connection

In this section, we conduct all testbed experiments using our own TCP test suite tools which makes use of *trpt* program to do kernel tracing. The congestion window evolution of TCP Veno is investigated and TCP Reno and TCP Veno variants with different recovery schemes are tested with different loss rates. Again, the impact of normalized buffer and the impact of propagation delay are also studied and compared with the results obtained in network simulator.

4.2.1.1 Congestion Window Evolution

We set the bandwidth BW_f and BW_r to 1Mbit/s, propagation delay D_f and D_r to 60ms and the queue buffer size to 16 packets irrelevant to the packet size.

Figure 4.2: a) *NewVeno* congestion window evolution b) *NewReno* congestion window evolution over wireless link

We transfer a 6.4 Mbyte file from the sender to the receiver. Figure 4.2 shows the *NewVeno* and *NewReno*'s congestion window evolutions with random loss rate of 1%. As shown, the *NewVeno* connection finishes at 69 seconds with an average throughput of 92.7 Kbyte/s. *NewReno* finishes at 97.87 seconds with an average throughput of 65.3 Kbyte/s. The improvement of throughput can be explained by looking at the detailed congestion window evolution of each connection. In this experiment BDP is equal to 15, the *NewVeno*'s congestion window can fluctuate around the equilibrium point and make better use of link bandwidth. In contrast, the congestion window of *NewReno* is away from the link equilibrium for most periods. With an average window of 10, which is far below the equilibrium point (BDP = 15), it leaves the available bandwidth unused. The congestion monitoring scheme of *NewVeno* helps it to distinguish between the types of loss, preventing it from unnecessary severe window deduction. This can make *NewVeno* utilize available bandwidth even when random loss is present. In this experiment with 1% random loss rate, the throughput of *NewVeno* is 41% higher than its Reno counterpart.

However, in this experiment, during the transfer of the 6.4 Mbyte size file, only 3 partial acknowledgments are received during recovery phase as 64 FFs have been triggered in total by *NewVeno*. This means that although most of the losses are due to random bit errors since there are rarely cases when the

congestion window can reach the congestion point ($BDP + 16 = 31$), the possibility of multiple random losses in one window of data segments is quite small, which is roughly 4.68% (that is $3/64$). This scenario is basically related to the simple loss model deployed by the IPFW firewall filter. In order to distinguish TCP Veno variants over different recovery schemes, we have to increase the bandwidth of the bottleneck link and loss rate in order to introduce more losses in unit of time. Further implementation of complex distribution loss models should be considered, such as two state error loss model [33]. This is because 1% has already been considered as a very high loss rate in typical wireless environments.

4.2.1.2 TCP Veno over Different Recovery Schemes

We increase L_f and L_r to 2%, BW_f and BW_r to 2Mbit/s. Besides, we set D_f and D_r to 30ms and queue size to 12 packets for both forward and backward directions. We test the six TCP variants with different loss rates ranging from 10^{-5} to 10^{-1} . Throughput is plotted in Figure 4.3.

The experimental results, depicted in Figure 3.13, are quite similar compared with those we obtained in simulation. Except that we choose 2% loss rate instead of 1%. When the loss rate is 0.1% or lower, the throughputs of the six TCP variants are close to each other with a bit higher on Veno variants. This is due to its refined additive increase algorithm that helps to stay longer in the fully-utilized network region. The throughputs of Veno variants are separated at 2% random loss. This is due to the efficiency of different recovery schemes on the condition of TCP Veno can maintain a relative high congestion window oscillating around network equilibrium point (BDP) even when random loss is present. However, with 10% severe loss rate none of the connections can maintain the congestion window and all of them have suffered serious throughput degradation.

Figure 4.3: TCP Variants over different loss rate

Impact of Normalized Buffer

For LANs, because the round-trip delay of link is small, the BDP value would be much smaller than the buffer in the bottleneck link; thus, the normalized buffer as ratio of queue buffer size over BDP is greater than 1. However, in WAN, usually we have the larger round-trip delay and the normalized buffer is normally smaller than 1. The throughput evaluation over impact normalized buffer has been tested in simulator. In this experiment, we are going to evaluate these six TCP variants in testbed.

The throughput evolution with different normalized buffer size is shown in Figure 4.4. The network is configured with $BW_f = BW_r = 2\text{Mbps}$, $D_f = D_r = 30\text{ms}$ and the varying buffer size is showed in the following table.

Normalized Buffer	0.3	0.8	0.93	1.1	1.53	2
Buffer (packets)	5	12	14	16	23	30

Table 4.1 Normalized Buffer Size

Figure 4.4: Normalized buffer effect of TCP variants

Figure 4.4 illustrates the throughputs of TCP variants for different normalized buffer sizes in lossy network with 2% loss rate. In lossy networks, all Venos variants achieve higher throughput than corresponding Reno variants. There is no obvious difference among the throughputs of the three Reno variants. Some differences among Venos variants can be observed with different recovery schemes. When the normalized buffer is greater than 0.3, each of connection's throughput line stay horizontally. Thus we conclude that different normalized buffer sizes do not affect the throughputs of Venos and Reno variants.

Impact of Propagation Delay

We set BW_f and BW_r to 2Mbps, queues size of 12 packets for both forward and backward directions. We conduct the experiments for the six variants of TCP Reno and Veno employed with different recovery schemes with propagation delay ranging from 40ms to 100ms. The results are shown in Figure 4.5.

Figure 4.5: Impact of propagation delay over TCP Variants

The most significant observation is that all TCP Veno variants are above Reno TCPs in terms of throughput gain for the whole RTT ranges. All three TCP Reno variants' throughputs are close to each other at each RTT value although different recovery schemes are deployed. The three Veno variants are separated at 40ms, 60ms and 80 ms round trip times. There are not much difference at 100ms test cases. We list some of the details of each TCP variant in Table 4.2.

	40ms	60ms	80ms	100ms
FackVeno	29/84	10/100	6/92	1/100
SackVeno	26/70	9/95	3/91	2/106
NewVeno	30/84	9/90	4/94	1/109

Table 4.2 Congestion loss vs Non-congestion loss of three Veno variants

The table lists the number of congestion losses over the number of non-congestion losses differentiated by Veno’s congestion monitoring scheme. For 40ms, *FackVeno*, *SackVeno* and *NewVeno* incur 29, 26 and 30 congestion losses respectively. For most of times, the congestion window can stay above the BDP value. At 60ms, *FackVeno*, *SackVeno* and *NewVeno* have 10, 9 and 9 congestion losses respectively. This is a sign that the congestion window oscillates around BDP value. Recovery schemes create much difference between the three Veno variants.

At 100ms round trip time, the congestion window cut occurrences are 2, 1 and 2 for *FackVeno*, *SackVeno* and *NewVeno*. This is an indication that congestion window stays below BDP value for most of connection times. There will be not much difference on throughput despite different recovery schemes employed.

4.2.2 Multiple connections

4.2.2.1 Fairness and compatibility

With $BW_f = BW_r = 2.4\text{Mbit/s}$, $D_f = D_r = 40\text{ms}$ queue length of 12 packets, the Figure 4.6 shows the sequence diagram of two *FackVeno* running together in wired network work, as observed, the two connections can share the link in a fair manner so the fairness can be achieved among *FackVeno* connections when no random loss is introduced. Similar experiments also have been done with *SackVeno* and *NewVeno* variants, the fairness among multiple connections can be

maintained. Figure 4.6(b) shows the sequence diagram of one *FackReno* and one *FackVeno* connection compete together within the same link, and the result illustrates the two connection can co-exist well.

Figure 4.6: (a) sequence diagram of FackVeno with FackVeno (b) sequence diagram of FackVeno and FackReno

Figure 4.7: (a) Sequence number of *FackReno* and *FackReno* with 1% random loss rate (b) Sequence number of *FackVeno* and *FackReno* with 1% random loss rate

Figure 4.7(a) shows the sequence diagrams of two *FackReno* connections with 1% random loss. We replace one *FackReno* connection with *FackVeno* and redo the experiment with the same settings. The sequence diagram is plotted in Figure 4.7(b). As we can observe, the *FackVeno* connection's sequence is higher than *FackReno*. However, the *FackReno* sequence diagram still remains the same slope as shown in Figure 4.7(a). *FackVeno* does not bias against *FackReno* connections. The congestion monitoring scheme helps it to differentiate between random loss and congestion loss. This makes better use of available link bandwidth in lossy networks.

Chapter 5

Implementations

We have implemented TCP VenO in both network simulator (ns-2.26) and FreeBSD 4.3 kernel. The implementation of TCP VenO contains two important parts: refinement of the default additive increase algorithm and multiplicative decrease algorithm of TCP congestion control mechanisms; and addition of a network congestion monitoring scheme by estimating the number of buffer packets accumulated in bottleneck link buffer.

Ns-2 provides a clean and neat basic *TcpAgent* class as a mount point for the implementation of TCP VenO. Besides, *NewReno* and TCP with SACK and FACK options have been implemented as *NewRenoTcpAgent*, *Sack1TcpAgent* and *FackTcpAgent* in default TCP package respectively. The implementation of corresponding VenO agents has become easier by refining the existing *TcpAgent*'s

AIMD algorithm and adding more code for the network congestion monitoring scheme.

The kernel implementation of TCP Veno in FreeBSD 4.3 operating system is a tougher task. The default TCP suite, implemented in FreeBSD 4.3, is *NewReno* version. By introducing Veno modifications we get *NewVeno* implementation. However, TCP with SACK option has been realized after FreeBSD 5.2 (while, FACK has not been realized yet), so there is a need to add SACK and FACK options into FreeBSD 4.3 Veno kernel in order to realize *SackVeno* and *FackVeno*.

In the following section, we are going to introduce the details of our implementations in simulator as well as in FreeBSD 4.3 kernel with an emphasis on Veno modifications.

5.1 TCP Veno Implementation in Network Simulator

The TCP Veno module in network simulator is inherited from the basic TCP agent class which realizes the basic TCP operations. The *TcpVenoAgent* class definition is added into tcp.h header file. Its implementation has been realized in tcp-veno.cc file by modifying the default *timeout()*, *recv()*, *recv_newack_helper()* and *output()* functions.

5.1.1 State Estimation

5.1.1.1 Four Evolution Phase

One variable, *v_state*, is used to keep track of the evolution state of a TCP Veno connection. The connection can be in one of these states at any time: initial slow start phase (VENO_ISS), slow start phase excluding initial slow start phase (VENO_SS), additive increase phase (VENO_AI), fast retransmit, and fast recovery phase (VENO_FF). Figure 5.1 shows the transition diagram among these phases.

Figure 5.1: *v_state* evolution of TCP Veno connection

When a connection starts, its congestion window is initialized to one segment. Every new ACK increases the congestion window by one segment. TCP is in initial slow start phase. *v_state* is set to VENO_ISS. The congestion window will grow in an exponential manner. When the *cwnd* is bigger than *ssthresh*, TCP enters into the additive increase phase. *v_state* will be set to VENO_AI. During this phase, the congestion window will grow in a linear manner which is one segment increase for every round trip time.

If packet loss is identified by DUPACKs during the initial slow start phase, the fast retransmit and fast recovery phase takes over. The *ssthresh* is set to half of loss congestion window. *v_state* is set to VENO_FF. If a new ACK comes along, it will bring the TCP out of fast retransmit and fast recovery phase and back to additive increase phase. *v_state* would be set to VENO_AI. Otherwise, if lack of acknowledgments to trigger new packets sent out and no new ACKs return, the connection will be trapped in FF state. It would finally rely on timeout of the loss packet's retransmit timer to bring the connection out of FF into slow start phase and *v_state* is set to VENO_SS.

The difference between ISS and SS state is that, the *ssthresh* for slow start phase is more precisely estimated since the connection has just incurred TIMEOUT. It will grow the congestion window to *ssthresh* and start the congestion avoidance algorithm (AI phase). Since it has already learned from previous cycle (before TIMEOUT), it will obtain a better estimation of the *ssthresh* value of the network property. However, for initial slow start phase, because of the lack of previous knowledge about network property, it always sets the *ssthresh* to a very large value. It probes the network until loss occurs.

5.1.2 Output Function

The `output()` function is mainly used for packet transmission. It assigns various fields in the TCP header (sequence number, timestamp, reason for transmission), handles the 3-way handshake to open up a new TCP connection and negotiates the explicit congestion notification option if necessary. This function is also responsible for setting a retransmission timer for the packet being sent out if one is not pending.

It is also responsible for updating a variable, *maxseq*, to hold the maximum sent sequence number so far. The original TCP timestamp option enables round trip time tracing only for per window of data segments which is a loss tracing. Veno needs to keep track of fine round trip time (RTT) value for every packet sent, the additional computation cost comes with the compensation of better network congestion estimation. The implementation of TCP Veno output function not only inherits the functionalities from the base class *TcpAgent*'s *output* function but also creates two new arrays, *v_sendtime* and *v_transmits*, to hold the sending time and number of retransmissions for each outgoing packet. When the corresponding acknowledgment returns for each packet, fine RTT value can be calculated as the current time minus its sent time.

In the network simulator, the receiving ACKer (TCPSink) is set with non-delayed option by default which acknowledges every packet that has been received. Unlike in the real kernel implementation of TCP/IP, the segment is identified by a sequence of bytes. In simulator, each packet is just identified by a sequence number. No actual data has been transferred between the two ends. It is possible to record the sending time of each packet by using a single array

(*v_sendtime*) with enough size for circular use. In kernel implementation, it would be more complicated to record a segment's sending time. We use a link list of *vstamp* structure which is explained in more details later in kernel implementation part. The *v_transmits* array is necessary since a retransmitted packet cannot be used for fine RTT measurements.

```
double* v_sendtime_;    // each unacked pkt's sendtime is recorded.

int* v_transmits_;      // # of retx for an unacked pkt

Two dynamic queues are declared to hold every current outstanding packet's send time.

int index = seqno % v_maxwnd_;

v_sendtime_[index] = venotime(); // packets sending out time

++v_transmits_[index]; //number of times this packet has been
                        //transmitted.

int bytes = hdr_cmn::access(p)->size();

ndatabytes_ += bytes;

ndatapack_++;

send(p, 0);
```

Finally, the *tcp_output()* function calls *send()* which passes the packet to the IP layer for further processing. The output function in Veno is usually called by *send_much()* function. The *send_much()* function attempts to send as many packets as the current congestion window allowed in all evolution states.

5.1.3 Recv Function

Data packets are received by link layer processed by IP process, and then the *recv()* function of TCP will be notified. TCP Veno modifications are mainly implemented in *recv()* function. This function in turn calls *recv_newack_helper()* when receives new acknowledgement, which implements the slow start and refined additive increase (AI) algorithms, does per round trip time updating to calculate the estimated number of packets in bottleneck buffer, and updates

BaseRTT value to the minimum value since last time reset. Besides, *recv()* function also keeps track of number of DUPACKs has been received. When the threshold *numdupacks_* is reached, the fast retransmit and fast recovery algorithm will be triggered. The Figure 5.2 shows the detailed flow control of *recv()* function.

Figure 5.2: recv() function flow chart

5.1.3.1 Calculation of Estimated Number of Packets in Bottleneck Buffer

When the network pipe is filled by packets from different transmitting sources, the queue in the network router starts to build up. Network congestion starts to occur. In order to ascertain when network congestion occurs, a variable, *v_no_buffer_packet_*, is defined to represent the number of packets currently accumulated in the router. Some other related variables have also been defined in *tcp.h*. They are shown in the following, to be used in the estimation scheme.

```
int   v_cntRTT_;           // # of rtt measured within one rtt
double v_sumRTT_;         // sum of rtt measured within one rtt

double v_begtime_;        // time tagged pkt sent
int    v_begseq_;         // tagged pkt seqno

double v_baseRTT_;        // min of all rtt

double v_actual_;         // actual send rate (pkt/s; needed for tcp-rbp)
double v_predict_;        // predict send rate ( cwnd / base_RTT)
int    v_no_of_buffer_packet_;
```

One particular sign of a congested network is the rapid increase in route-trip times (RTTs) of data packets sent. We need to keep track of the fine RTT changes. The following code does it.

```
double sendTime = v_sendtime_[tcph->seqno()%v_maxwnd_];
int transmits = v_transmits_[tcph->seqno()% v_maxwnd_];
int range = tcph->seqno() - oldack;
for(int k=((oldack+1) %v_maxwnd_); \
    k<=(tcph->seqno()%v_maxwnd_) && range >0 ; \
    k=((++k) % v_maxwnd_), range--) {
    v_sendtime_[k] = -1.0;
    v_transmits_[k] = 0;
}
```



```

if((sendTime !=0.)&& (transmits==1)) {

    double rtt, n;
    rtt = currentTime - sendTime;
    if( v_state_ == VENO_AI) { // only do in VENO_AI phase
        v_sumRTT_ += rtt;
        ++v_cntRTT_;

        if (rtt>0){

            if(v_baseRTT_ <0 ){
                v_baseRTT_ = rtt;
            }
            if(rtt < v_baseRTT_)
                v_baseRTT_ = rtt;
        }
    }
}

```

By using the *sendTime* and *transmits* arrays described in Section 5.1.2, we can obtain fine RTT for each sent segment. We get a smooth RTT using $v_summRTT/v_cntRTT$, where $v_summRTT$ is the sum of every fine RTT. v_cntRTT is the total number of segments in the tagged window. Following this, the actual rate ($v_actual_$) is calculated by the number of sent packets during last tagged RTT ($t_seqno_ - v_begseq_$) divided by smooth RTT.

```

if(v_cntRTT_ > 0)
    rtt = v_sumRTT_ / v_cntRTT_;
else
    rtt = currentTime - v_begtime_;

rtt = v_sumRTT_ / v_cntRTT_;
int rttLen = t_seqno_ - v_begseq_;
if(rtt>0 && v_state_ != VENO_FF) {
    v_actual_ = double(rttLen)*100/rtt;
    v_predict_ = double(t_seqno_-last_ack_)*100/v_baseRTT_;
    ...
}

```

For the expected rate, we use the outstanding data segments ($t_seqno_snd_una$) divided by $v_baseRTT$, which is the minimum of the RTTs of all packets since last reset. After that we set the tagged sequence, v_begseq , to the current segment that is going to be sent out. Another cycle is started in order to collect fine RTT values until the whole tagged window of segments return. Specifically, it is when the ACK for the v_begseq segment returns. The v_sumRTT and v_cntRTT are reset to zero at the end of each cycle. The loop just continues. The values of v_actual and v_expect are calculated on a per RTT basis. We can easily use the formula $(v_expect - v_actual) * v_baseRTT$ to calculate the estimated number of accumulated packets ($v_no_of_buffer_packet$) in the bottleneck buffer.

The value of $v_no_of_buffer_packet$ is then used to estimate the state of network: congestive or non-congestive. The value of $v_no_of_buffer_packet$ represents the number of buffer packets contributed by this particular TCP Veno connection. It is not the value of total queued packets accumulated in the bottleneck buffer. It is worthwhile to emphasize that $v_no_of_buffer_packet$ is only calculated in VENO_AI state.

5.1.4 Window Adjustment

Based on the estimated number of accumulated packets in the bottleneck buffer presented in Section 5.1.4.1, we describe how to refine the additive increase algorithm. The implementation of a refined multiplicative decrease algorithm is reported in Section 5.1.4.2.

5.1.4.1 Refined Additive Increase

For typical TCP Reno connections, acknowledgements trigger new segments to be transmitted. Each acknowledgement increases the congestion window by $1/cwnd$. When all acknowledgements for the previous window of data segments return, which is exactly one RTT time period, the congestion window is increased roughly by 1. This would allow one more data segment to be sent out immediately,

following the last segment. For a Veno connection running in the VENO_AI phase, the congestion window may increase by 0 or 1 segment for a RTT time. It depends on the estimated network congestion state of the connection. Specifically, when the `v_no_of_buffer_packet_` is less than three segments, same as Reno's additive increase algorithm, the congestion window is increased by 1 per RTT time. On the other hand, when `v_no_of_buffer_packet_` is larger than or equal to three segments, which signifies that the network has been fully utilized. The increment of the congestion window may not result in an increase in throughput any more. So we slow down the increase in speed by every other RTT time. The source code reflects this change as follows.

```

delta=int((v_predict_-v_actual_)*v_baseRTT_+0.5);
if(delta>=v_beta_) {
    // increase the other time
    if(v_inc_flag_){
        v_incr_=1/cwnd_;
        v_inc_flag_=0;
    }else {
        v_incr_=0;
        v_inc_flag_=1;
    }
} else {
    v_incr_ = 1/cwnd_;
}
if (!ect_ || !hdr_flags::access(pkt)->ecnecho() || (old_ecn_ && ecn_burst_)) {
    cwnd_ +=v_incr_;
}

```

5.1.4.2 Refined Multiplicative Decrease

When three DUPACKs are received, the FF is triggered. For TCP Reno, the congestion window is always deducted by half. TCP Veno reduces the congestion window based on an estimation of network congestion state. If `v_no_of_buffer_packet_` is greater than or equal to three segments, we declare that

the connection is in the congestive state. The congestion window is halved. Otherwise the congestion window is reduced by 20%. The *v_state* variable is set to VENO_FF. This setting indicates that the connection is evolving into fast retransmit and fast recovery phase. When a new ACK is received, *v_state* is reset to VENO_AI.

```

int delta=v_no_of_buffer_packet_;

if((delta>= v_beta_ && v_state_==VENO_AI) || v_state_==VENO_ISS \
|| (v_state_==VENO_FF && delta>=v_beta_)){

    ssthresh_=(int)(win/2 +0.5);

    if (ssthresh_ <2)

        ssthresh_ =2;

    cwnd_ = ssthresh_ + dupacks_;

    v_state_ =VENO_FF;

}

else if((delta < v_beta_ && v_state_==VENO_AI) || v_state_==VENO_SS \
|| (v_state_==VENO_FF && delta <v_beta_)){

    ssthresh_ = (int)(win*4/5);

    if (ssthresh_ <2)

        ssthresh_ =2;

    cwnd_ = ssthresh_ + dupacks_;

    v_state_ =VENO_FF;

}

```

One point needed to be emphasized is that during VENO_SS phase, if a loss is encountered, the congestion window always gets cut by 20%. The loss is treated as random loss instead of congestion loss. The reason is that before VENO_SS state, there is a TIMEOUT which represents severe network congestion. The *ssthresh* is set to half of the loss window. Congestion window is reset to 1 to restart slow start operations. Since the network has just experienced

congestion and the Veno connection has been severely punished, there is a small possibility that Veno will encounter another network congestion shortly. More importantly, Veno does not estimate *v_no_of_buffer_packet_* in VENO_SS phase. Normally after fast recovery or TIMEOUT, the network pipe is cleared. All TCP connections would likely cut their congestion windows. It would be the most proper time to collect *v_baseRTT* values for later use in VENO_AI phase, rather than to consider the loss is caused by congestion or random noise.

When there is a transmission timer timeout, the *ssthresh* will be set to half the congestion window and the state variable, *v_state*, will be set to VENO_SS. Another slow start takes over. The following source code is extracted from *timeout()* function, residing in *tcp_veno.cc*.

```
if (tno == TCP_TIMER_RTX) {
    dupacks_ = 0;
    v_state_ = VENO_SS;
    recover_ = maxseq_;
    v_baseRTT_ = -1;
    v_inc_flag_ = 0;
    TcpAgent::timeout(tno);
    .....
```

5.2 TCP Veno Modification in FreeBSD 4.3

TCP Veno is implemented by adding a small amount of code into *tcp_input()* and *tcp_output()* functions. The *tcp_input* function is called by *ipintr* when a datagram is received with a protocol field of TCP. The *tcp_input* executes at the software interrupt level. The *tcp_output* function is called when TCP wants to send a segment.

Figure 5.3 gives an overview of Veno modification in FreeBSD 4.3. The respective additional modules are referred to as the *veno_receive* and *veno_send*. They are implemented in two extra files (*tcp_veno.c* and *tcp_veno.h*).

Figure 5.3: TCP Veno modifications overview

We first describe *veno_send* and *veno_receive* functions, which are embedded in *tcp_input()* and *tcp_output()*. We then introduce the calculation to estimate the number of buffer packets in the bottleneck link buffer in Section 5.2.2. Finally, we describe the refinement of the additive increase and multiple decrease algorithms in TCP Veno in Section 5.2.3.

5.2.1 Veno_send and Veno_receive functions

The mechanism used to calculate the fine round trip time for each segment in a congestion window is basically done in *veno_send()* and *veno_receive()* functions. Previous implementation in NetBSD calls them out and in filter respectively.

First of all, a timestamp structure, known as *vstamp*, is defined with sent segment length, sending time and how many times it has been transmitted. Basically it is a link list structure.

TCP Veno maintains a timestamp queue (*vp->v_queue* of *vstamp*)

structure for all outstanding segments, every time before the segment is sent out. *veno_send()* is called first to record each sent segment length and its sending time with millisecond resolution. When ACKs are received, *tcp_input()* would access the timestamp queue by *veno_receive()* function to extract the segment's *vs_sendtime*. By subtracting *currenttime* and *vs_sendtime*, we would get the round trip time for a particular data segment.

```
struct vstamp {
    int vs_len;           /* segment length */
    int vs_sendtime;
    int vs_transmits;     /* we don't calculate retransmit packet RTT */
    struct vstamp *vs_next;
};
```

In order to improve the kernel performance, we do not allocate memory when a new *vstamp* structure is needed every time. Instead, we create a memory pool of *vstamp* structure kept in timestamp queue (*q_array*) and let *q_free* first point to *q_array*. Whenever a new *vstamp* structure is needed, we just use the current *q_free* pointer without the need to allocate new memory space. *q_free* is updated to point to the next structure in the link list. When a *vstamp* is to be destroyed, we does not actually free the memory space by calling *free()* function or *m_freem()*. Instead we just assign dummy data to the various fields of the current *vstamp* structure. The new node is inserted into the front of the link list referenced by *vp->q_free*.

Before we really send a data segment by calling *ip_output()*, we call *veno_send(vp, off, len, currenttime)* function first to create a *vstamp* record in the *vp->v_queue*. The *vp->v_queue* stores all the outstanding data segments that have not been acknowledged yet. The parameter *off* is equal to *snd_nxt - snd_una*. It specifies the position where we should insert the record in the *vp->v_queue* link list. The *veno_send()* function is nothing more complex than an insertion of a record into a link list at a proper position. Of course, we have to account for the situation where there is a need to retransmit a packet. In this situation the *vs_transmits* will be increased by one.

Veno_Send()

Figure 5.4: Send a segment in TCP Veno *output* function

Veno_Receive()

The *veno_receive()* and so called in-filter function, which is called by *tcp_input()* function defined inside file *tcp_input.c* when a new ACK is received. The *veno_receive(struct venocb *vp, int len, int *sendtime, int *transmits)* function just retrieve the *sendtime* and *transmits* value (since they are defined as a pointer type) for the corresponding segments that has been successfully acknowledged. The *len* value this time would be equal to *th->th_ack - tp->snd_una*, the segment length that has been total acknowledged. Obviously by traversing the link list point by *vp->v_queue*, we not only need to get the *sendtime* and *transmits* information but also need to free the old space, again the *vstamp* record is not really destroyed as we have described previously. However, the corresponding data segments in the sender's socket buffer would be dropped.

By using *veno_send()* and *veno_receive()* functions, we can compute fine round trip time for each packet and which is going to be used to calculate the estimated number of packets in bottleneck buffer later in VENO_AI phase.

Figure 5.5: ACK processing in TCP Veno input function

5.2.2 Calculation of Estimated Number of Packets in Bottleneck Buffer

Unlike the way we define Veno related variables in network simulator using *TcpVenoAgent* class, in FreeBSD 4.3 kernel, we define the system variables used by Veno mechanism in Veno control block (*venoch*). Every TCP connection has its own TCP control block. A TCP control block identifies the source and destination addresses and port numbers. If the connection is a Veno connection, we would create a Veno control block (*venoch*) in the initialization phase and used as a structure member of TCP control block. The Veno control block (*venoch*) structure, residing in *tcp_veno.h*, is as follows.

```
struct venoch {
    int evo_state;      /*VENO_ISS, VENO_SS, VENO_FF, VENO_AI */
    int v_no_of_buffer_packet;
    int v_s_no_of_buffer_packet;
    int v_number_of_rtt;
    char v_flags;      /* TCP_TAHOE, TCP_RENO, TCP_VENO */
    int v_actual_rate;
    int v_pred_rate;

    /* Veno timer stuff */
    int v_rtt;
    int v_sa;
    int v_sd;
    int v_timeout;
    int v_worried;      /* While > 0, check for packet
                        * timeout */
    int v_time_cwin_chg; /* Time when congestion window
                        * was last decreased. Used to prevent
                        * decreasing cwin more than once
                        * for losses which occurred at
                        * same cwin level */
    int v_synsent;      /* Time when SYN was sent. Used to
                        * set v_rtt at beg of connection */

    /* Congestion detection (or more correctly avoidance ) variables.
     * The mechanism checks once per RTT. Current RTT began with sequence
     * v_cong_detect_begseq
     */
    int v_cong_detect_do; /* Whether to do it */
    tcp_seq v_cong_detect_begseq;
    int v_cong_detect_begtime; /* When RTT began */
    int v_cong_detect_predict_do; /* Whether to predict the
                                * available bw during
                                * slow-start */

    int v_cong_detect_last_sendtime;
    int v_cong_detect_baseRtt; /* Current base RTT, usually
                                * the minimum */
    int v_cong_detect_sumRTT; /* Used to get average */
    int v_cong_detect_cntRTT; /* RTT during a RTT interval */
};
```

```

.....
.....

/* Veno timestamp queue*/
struct timeval v_start;          /* starting time of the veno session */
struct vstamp *v_queue;         /* veno queue is a list of veno timestamps
                                of current outstanding segments */

#define POOLSIZE 4096
    struct vstamp *q_array;      /* veno queue is implemented as an
                                array of vstamps */
    struct vstamp *q_free;       /* list of un-used vstamps */
    int q_maxlen;               /* max len of the queue */
};

```

The variable *evo_state* is used to keep track of the state the TCP Veno connection is currently in. Once we get a sample of RTT of a packet by calling *veno_receive()* we cumulatively add *v_cong_detect_sumRTT* by the sampled RTT value and cumulatively add *v_cong_detect_cntRTT* by one and this continues until the sequence of received ACK is larger than tagged data sequence number (indicating this tagged RTT is over and next tagged data is starting to send). Similar to what we have done in simulator, we calculate *v_actual_rate* and *v_expect_rate* in order to get *v_no_of_buffer_packet_* value. After that, two variables *v_cong_detect_sumRTT*, *v_cong_detect_cntRTT* are initialized to zero and start another tagged RTT and the loop just continues. The value of *v_no_of_buffer_packet_* is used to estimate the state of network – congestive state or not congestive state.

5.2.3 Window Refinement

In this section we will show how to implement the refined additive increase algorithm and multiple decrease algorithm by using the estimated number of accumulated packets in the bottleneck buffer.

Refined additive increase algorithm

When Veno connection detects in congestive state, it would slow down the congestion window increasing speed by half. The source code of refined additive increase is as follows:

```

register u_int cw = tp->snd_cwnd;
register u_int incr = tp->t_maxseg;

if (cw > tp->snd_ssthresh){
    if (VP->v_no_of_buffer_packet < 3*10*tp->t_maxseg)
        incr = incr * incr / cw;
    else
        {
            if (VP->v_number_of_rtt >=2 ) {
                incr = incr*incr /cw;
                VP->v_number_of_rtt =1;
            }
            else
            {
                VP->v_number_of_rtt++;
                Incr =0;
            }
        }
}
tp->snd_cwnd = min (cw +incr, TCP_MAXWIN << tp->snd_scale);

```

Refined multiple decrease algorithm

The FF happens when the condition $if(++tp->t_dupacks == tcprexmtthresh \parallel ((SEQ_GT(tp->snd_fack, tcprexmtthresh * tp->t_maxseg + tp->snd_una)) \&\& SEQ_GT(tp->snd_una, tp->snd_last)))$ is true for FACK options. That is to say, FACK has the ability to trigger the FF even without the need to wait for three DUPACKs. For SACK and *NewReno*, the condition is still when three DUPACKs ($++tp->t_dupacks = tcprexmtthresh$) are received. In TCP Reno, window is always cut by half. TCP Veno reduces window based on the state of the connection. If it is in the congestion state which is equivalent to $v_no_of_buffer_packet$ is not less than three segments, and the congestion window is halved. Otherwise, window is reduced by a factor of 1/5, namely $win=(min(tp->snd_wnd, tp->snd_cwnd)*4/(5*tp->t_maxseg))$. The code fragment is shown below.

```

if (++tp->t_dupacks == tcprexmtthresh) {
    tcp_seq onxt = tp->snd_nxt;
    u_int win;

    if((VP->evo_state == VENO_AI && VP->v_no_of_buffer_packet >=
3*10*tp-
>t_maxseg) || VP->evo_state == VENO_ISS ||
(VP->evo_state == VENO_FF && VP->v_no_of_buffer_packet >= 3*10*tp-
>t_maxseg))
    {
        win = min(tp->snd_wnd, tp->snd_cwnd) / 2 / tp->t_maxseg;
    }

    else if((VP->evo_state == VENO_AI && VP->v_no_of_buffer_packet <
3*10*tp-
>t_maxseg) || VP->evo_state == VENO_SS ||
(VP->evo_state == VENO_FF && VP->v_no_of_buffer_packet < 3*10*tp-
>t_maxseg))
    {
        win = (min(tp->snd_wnd, tp->snd_cwnd) * 4 / (5*tp->t_maxseg));
    }

    if(ISVENO(tp)){
        VP->evo_state = VENO_FF;
        VP->v_cong_detect_baseRtt = -1;
        VP->v_number_of_rtt = 1;
        VP->v_s_no_of_buffer_packet = -10000;
    }

    #if defined(TCP_SACK)
        if (SEQ_LT (th->th_ack, tp->snd_last)) {
            /* false ff after timeout . do not cut */
            tp->t_dupacks = 0;
            goto drop ;
        }
        #else
        if (tcp_do_newreno && SEQ_LT(th->th_ack,
tp->snd_recover)) {
            /* False retransmit, should not
            * cut window
            */
            tp->snd_cwnd += tp->t_maxseg;
            tp->t_dupacks = 0;
            (void) tcp_output(tp);
            goto drop;
        }
        #endif
    if (win < 2)

```

```

        win = 2;

        tp->snd_ssthresh = win * tp->t_maxseg;

#ifdef TCP_SACK

        tp->snd_last = tp->snd_max;
#else
        tp->snd_recover = tp->snd_max;
#endif

#ifdef TCP_SACK
        if(!tp->sack_disable) {
            callout_stop(tp->tt_rexmt);
            tp->t_rtttime=0;
        }
#endif
#ifdef TCP_SACK && defined (TCP_FACK)
        tp->t_dupacks = tcprexmtthresh;
        void) tcp_output(tp);

        /***** during FR, snd_cwnd is held constant for FACK */
        tp->snd_cwnd = tp->snd_ssthresh;
#else

        /*      tcp_output() will send oldest SACK */
        (void) tcp_output(tp);

        tp->snd_cwnd = tp->snd_ssthresh + tp->t_maxseg*tp->t_dupacks;

#endif /*TCP_FACK*/
        goto drop;
    }

#endif /*TCP_SACK*/

/* original NewReno code in FreeBSD 4.3 */

    callout_stop(tp->tt_rexmt);
    tp->t_rtttime = 0;
    tp->snd_nxt = th->th_ack;
    tp->snd_cwnd = tp->t_maxseg;
    (void) tcp_output(tp);
    tp->snd_cwnd = tp->snd_ssthresh + tp->t_maxseg * tp->t_dupacks;

    tp->t_timer[TCPT_REXMT] = 0;
    tp->t_rtt = 0;
    tp->snd_nxt = ti->ti_ack;
    tp->snd_cwnd = tp->t_maxseg;

    ...
    ...
    ...
    ...
    ...
    goto drop;
}

```

The above code is a modification of the original FreeBSD 4.3 kernel. SACK and FACK options have been added. After window deduction is performed, SACK or *NewReno* (depends whether *TCP_SACK* is defined) would remember the max segment sequence sent so far. It enters into FF in *tp->snd_last* and *tp->snd_recovery* respectively for following checks against false FF happening when multiple losses occur in one congestion window of data. Unlike the original Reno or *NewReno* algorithm, which set *cwnd* to 1 and *snd_nxt* to *ti->ti_ack* to retransmit the loss packet, SACK and FACK TCP would directly call *tcp_output()* function, which will in turn call *tcp_sack_output()*, which decides the packet to be sent out.

There are not many changes in the timer timeout function. We just reset the *baseRTT* value and set the *evo_state* to *VENO_SS*. The following source code is extracted from *tcp_timers()* function, residing in *tcp_timer.c*.

```
case TCPT_REXMT:
...
u_int win = min(tp->snd_wnd, tp->snd_cwnd) / 2 / tp->t_maxseg;
if (win < 2)
win = 2;
tp->snd_cwnd = tp->t_maxseg;
/*Evolution of connection state enter to Slow Start state
regardless of previous state, */
if (ISVENO(tp)) {
VP->evo_state = VENO_SS;
VP->v_cong_detect_baseRtt = -1; VP->v_number_of_rtt = 1;
}
tp->snd_ssthresh = win * tp->t_maxseg;
tp->t_dupacks = 0;
}
(void) tcp_output(tp);
```


Chapter 6

Conclusions and Future Work

6.1 Summary

In this thesis, two alternative approaches simulation and testbed have been adopted. Due to ns-2 simulator's clean and event-driven monitoring environment, the experiments done in simulation are more on focus to explain the TCP's window dynamics in micro perspective. The detail reasoning on window's dynamics has been given for out-of-phase synchronization scenario when VenO and Reno co-exist. Furthermore, detail sequence diagrams in recovery phase have been given for VenO and Reno when different recovery schemes employed and explain why FackVenO can achieve the highest throughput among others.

The experiments done in testbed are more related to performance evaluation of VenO and Reno TCP with various recovery schemes. In addition the normalized

buffer and propagation delay impact on FackVeno SackVeno and NewVeno have been presented. The testbed experimental results are not only a supplementary to the simulation but also as a proof against the conclusion drawn in simulation. In summary we can conclude our main contribution as follows.

6.2 Contributions

1) Intensive studies on window evolutions of TCP Veno

This thesis has described the dynamic details of window evolutions in TCP Veno. By distinguishing between the types of packet loss, Veno algorithm prevents under-utilization of the network bandwidth caused by false detection of congestion. In addition, Veno also modifies the additive increase phase of congestion control algorithm to make it stay longer in the additive increase phase. This has resulted in interesting interactions of window dynamics between Veno and Reno.

Regarding the refined additive increase algorithm in TCP Veno, this thesis has demonstrated that the out-of-phase synchronization phenomena in network simulator can lead to better link utilization and avoid global synchronization problems.

For the refined multiplicative decrease algorithm in TCP Veno, this thesis has investigated its interactions with different recovery schemes to overcome multiple losses in wireless environments. Three significant results have been obtained; 1) no matter what recovery scheme is employed, Veno outperforms Reno. Different recovery schemes can differentiate the performances of Veno connections. Veno has the ability to retain high congestion window size even in the presence of random loss; 2) due to the efficiency of FACK recovery scheme, Veno over Fack, which is referred to as *FackVeno*, has the best performance; 3) in wireless environments, recovery schemes become useless for Reno. The blind window-cut behaviour of Reno degrades the TCP performance. It creates performance differences among Veno variants. TCP Veno can maintain a high congestion window even in the presence of random bit errors.

2) Implementation of Veno, *NewVeno*, *SackVeno* and *FackVeno* in network simulator and FreeBSD 4.3.

The implementation of TCP Veno module in network simulator facilitates the observation of TCP dynamic window evolution in details. Comprehensive performance evaluations under different network topologies and settings can be conducted. In addition to the implementation of *NewVeno*, we have added SACK and FACK options into the FreeBSD 4.3 kernel. The code realization facilitates further tests in realistic wireless Internet.

6.3 Future work

6.3.1 Loss model development and mobility support

We have pointed out in our testbed-based experiments, reported in chapter 4, that the loss model employed by IPFW firewall filter is not efficient to generate multiple or burst losses. More complex models are desired to be developed. A two-state loss model would be helpful to accurately simulate the loss condition in realistic wireless networks. With the availability of kernel sources code, experiments can be conducted through mobile devices, using transport service over real WLAN or WWAN networks. Further evaluations of TCP performance against long burst losses can be carried out. Long burst losses are most likely to occur when the mobile devices have a vertical handoff over different wireless access networks. TCP Veno over different recovery schemes also can be examined for long burst losses. Solutions can be proposed to provide Veno with this kind of mobility support.

6.3.2 Veno over RED

Our research work has only been concerned with routers with drop-tail policy. The research can be extended to cover the investigation of Veno over routers with Active Queuing Management (AQM), especially RED (Random Early Detection) which is equipped with ECN (Explicit Congestion Notification) support.

RED accepts all packets until the average queue length reaches the minimum threshold. After that it drops a packet according to a linear probability distribution function. When the average queue length reaches the maximum threshold, all packets are dropped with a probability of one. In comparison to the drop-tail policy, which is one of Passive Queuing Management schemes, RED can reduce the number of packets dropped in routers. RED provides a low delay interactive service. Lockout behaviour is avoided. RFC 2309 highly recommends AQM for the next generation Internet routers. It will be worthwhile to study the performance of Veno over AQM, such as RED. Over the past decade, many researchers have studied extensively the behaviour of Reno over AQM routers. Their experiences are very valuable to our research. Evaluations can be conducted in both simulator and testbed. It would be very interesting to study the dynamics of TCP Veno over RED routers, especially *NewVeno*, *SackVeno*, *FackVeno*.

Publications

Published

C. L. Zhang, M. T. Yap, C. P. Fu, C. H. Foh, K. K. Wong, C. T. Lau, M. K. Lai,
“Dynamics Comparison of TCP Reno and Venet”, *IEEE, Global
Telecommunications Conference (Globecom)*, 2004

References

- [1] V. Cerf and R. E. Kahn, "A Protocol for Packet Network Interconnection". IEEE, Trans on Comms, Vol Com-22, No 5 May 1974
- [2] V. Cerf, Y. Dalal, and C. Sunshine, "Specification of Internet Transmission Control Program", RFC675, IETF, December, 1974.
- [3] C. Borman, The Multi-Class Extension to Multi-link PPP. RFC 2686, IETF, September 1999.
- [4] V. Cerf, "TCP Version 2 Specification", USC/Information Sciences Institute, IEN-5, March 1997.
- [5] J. Postel, "Comments on Internet Protocols and TCP", IEN 2, IETF, August 1977.
- [6] V. Cerf, "TCP Specification 3", IEN21, IETF, January 1978.
- [7] J. Postel, DOD Standard Transmission Control Protocol. USC/Information Science Institute, IEN-129, RFC761, NTIS ADA082609, January 1980.
- [8] Van Jacobson. Congestion Avoidance and Control. SIGCOMM'88, ACM, Aug, 1988
- [9] R. Jain, "A Delayed-Based Approach fro Congestion Avoidance in Interconnected Heterogeneous Computer networks", ACM Computer Communication Review, Vol, 19, No.5, October 1989.
- [10] Z. Wang and J. Crowcrof. "A New Congestion Control Scheme: Slow start and Search (Tri-S)." ACM Computer Communication Review Jan.1991"
- [11] S. Keshav. "A Control-Theoretic Approach to Flow Control" SIGCOMM91. Sept. 1991
- [12] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," Proceedings of ACM SIGCOMM'94, no.4 pp.24-35, October 1994
- [13] Lin. D., Kung, H. T, "TCP fast recovery strategies: analysis and improvements", Proceedings of INFOCOM' 98, 29 Mar- 2 April 1998.

- [14] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan, "Evaluation with TCP Vegas: Emulation and Experiment," ACM SIGCOMM Computer Communications Review, vol.25, no.4, pp.185-195, August 1995.
- [15] H. Ohsaki, M. Murata, T. Ushio, and H. Miyahara, "A Control Theoretical Analysis of a Window-Based Flow Control Mechanism in TCP/IP Networks" 2000.
- [16] C. Parsa and J.J. Garcia-Luna-Aceves, "Improving TCP Congestion Control over Internets with Heterogeneous Transmission Media" 1999, Computer Engineering Dept, University of California.
- [17] U. Hengartner, J. Bolliger and Th. Gross. "TCP Vegas Revisited" INFOCOM 2000.
- [18] Sally Floyd, Kevin Fall. "Promoting the Use of End-to-End Congestion Control in the Internet" IEEE/ACM Transactions on Networking, August 1999.
- [19] Kevin Fall and Sally Floyd, "simulation-based Comparisons of Tahoe, Reno, and SACK TCP", Computer Communication Review, July 1996.
- [20] Mathis, M., Mahadavi, J., Floyd, S. and A. Romanow, "TCP Selective Acknowledgment Options", RFC2018, October 1996.
- [21] Network Simulator, version 2.26. <http://www.isi.edu/nsnam/ns>.
- [22] L. Rizzo, "Dummynet and Forward Error Correction", In Proc. Freenix 98, 1998. or http://www.iet.unipi.it/~luigi/ip_dummynet/
- [23] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on queue management and congestion avoidance in the Internet. RFC2309, IETF, April, 1998.
- [24] Cheng Peng Fu, "TCP Veno: End-to-end Congestion Control over Heterogeneous Networks", Ph.D dissertation, July, 2001.
- [25] Lawrence S. Brakmo and Larry L. Peterson, "TCP Vegas: End to End congestion avoidance on a global internet," IEEE journal an Selected Areas in Communication, vol.13, no8, pp.1465-1480, Oct, 1995.
- [26] Floyd, S. and T. Henderson, "The NewReno Modification to TCP's Fast

Recovery Algorithm”, RFC2582, April 1999.

- [27] Tcpcdump, <http://www.tcpcdump.org/>.
- [28] Floyd, S.1996. Issues of TCP with SACK. Technical report (Jan.), Information and Computing Sciences Division, Lawrence Berkeley Laboratory, Berkeley, CA.
- [29] M. Mathis, J. Mahdavi, “Forward Acknowledgment: Refining TCP Congestion Control,” Proceedings of SIGCOMM’96, August, Stanford, CA.
- [30] Scott Shenker, Lixia Zhang, David D.Clark, “Some Observations on the Dynamics of a Congestion Control Algorithm” ACM SIGCOMM Computer Communication Review, Volume 20, Issue 5, Oct, 1990.
- [31] Lixia Zhang, Scott Shenker, David D. Clark, “Observations on the Dynamics of Congestion Control Algorithm : The Effects of Two-way Traffic”. Proceedings of the conference on Communications architecture & protocols, Zurich, Switzerland Pages: 133-147, 1991.
- [32] Cheng Peng Fu, and Soung Chang Liew, “TCP Veno: TCP enhancement for wireless access networks,” IEEE Journal of Selected Areas in Communications, vol.21, no.2, February 2003.
- [33] Alhussein A. Abouzeid, Sumit Roy, Murat Azizoglu “Stochastic Modeling of TCP over lossy Links.” IEEE INFOCOM 2000.
- [34] L.Rizzo, “Dummynet and Forward Error Correction”, In Proc. Freenix 98,1998. or http://www.iet.unipi.it/~luigi/ip_dummynet/
- [35] L. Rizzo, “Dummynet: a simple approach to the evaluation of network protocols”, ACM Computer Communication Review 1997.
- [36] T. V. Lakshman, Upamanyu Madhow, “The performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss”, IEE/ACM Transactions on networking, Vol 5, No. 3, June 1997.