

M. Sc. Engineering Thesis

# An Improved TCP Congestion Control Algorithm for Wireless Networks

by  
Ahmed Khurshid  
Student No. 100505023P



Submitted to

Department of Computer Science and Engineering  
in partial fulfillment of the requirements for the degree of  
Master of Science in Computer Science and Engineering

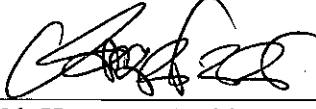

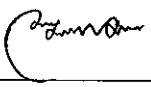
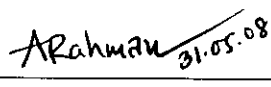
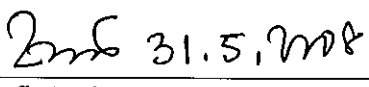


Department of Computer Science and Engineering (CSE)  
Bangladesh University of Engineering and Technology (BUET)  
Dhaka – 1000, Bangladesh

May 31, 2008

The thesis titled "An Improved TCP Congestion Control Algorithm for Wireless Networks", submitted by Ahmed Khurshid, Student No. 100505023P, Session October 2005, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on May 31, 2008.

## Board of Examiners

1.  31.05.2008  
Dr. Md. Humayun Kabir  
Associate Professor  
Department of CSE  
BUET, Dhaka – 1000  
Chairman  
(Supervisor)
2.   
Dr. Md. Saidur Rahman  
Professor & Head  
Department of CSE  
BUET, Dhaka – 1000  
Member  
(Ex-officio)
3.   
Dr. Md. Mostofa Akbar  
Associate Professor  
Department of CSE  
BUET, Dhaka – 1000  
Member
4.  31.05.08  
Dr. A. K. M. Ashikur Rahman  
Assistant Professor  
Department of CSE  
BUET, Dhaka – 1000  
Member
5.  31.5.08  
Dr. Hafiz Md. Hasan Babu  
Professor  
Department of CSE  
University of Dhaka, Dhaka - 1000  
Member  
(External)

## Candidate's Declaration

This is to certify that the work entitled “An Improved TCP Congestion Control Algorithm for Wireless Networks” is the outcome of the investigation carried out by me under the supervision of Dr. Md. Humayun Kabir in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka – 1000, Bangladesh. It is also declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

A. Khurshid. 31/5/2008

Ahmed Khurshid

Candidate

# Table of Contents

<b>Board of Examiners</b> .....	<b>i</b>
<b>Candidate's Declaration</b> .....	<b>ii</b>
<b>Acknowledgements</b> .....	<b>ix</b>
<b>Abstract</b> .....	<b>x</b>
<b>1 Introduction</b> .....	<b>1</b>
<b>2 TCP Basics</b> .....	<b>5</b>
2.1 TCP Segments .....	6
2.2 TCP Acknowledgments.....	6
2.3 TCP Receiver and Congestion Windows .....	6
2.4 TCP Congestion Control .....	7
2.4.1 Slow-Start .....	7
2.4.2 Congestion Avoidance.....	9
2.4.3 Fast Retransmission.....	10
2.4.4 Fast Recovery .....	11
2.5 Major TCP Variants.....	13
2.5.1 TCP Tahoe.....	13
2.5.2 TCP Reno .....	13
2.5.3 TCP NewReno.....	13
2.5.4 TCP Vegas.....	14
2.5.5 TCP Westwood.....	15
2.5.6 TCP SACK .....	15
2.5.7 TCP D-SACK.....	16
2.6 TCP in Wired Network.....	16
2.7 Characteristics of Wireless Networks Affecting TCP's Performance.....	17
2.7.1 Limited Bandwidth.....	18
2.7.2 Long Round-Trip Times.....	18

2.7.3	Random Losses.....	18
2.7.4	User Mobility.....	18
2.7.5	Power Consumption .....	19
2.7.6	Medium Access Control (MAC) Layer Activities .....	19
2.8	Performance of TCP Congestion Control Algorithm in Wireless Network	19
<b>3</b>	<b>State of the Art.....</b>	<b>23</b>
3.1	Link Layer Solutions .....	23
3.2	Base Station Dependent Approach .....	24
3.3	Using Redundant Error Correcting Segments .....	26
3.4	Router Assisted Approach .....	26
3.5	Split Connection Approaches .....	27
3.6	End-to-End Mechanisms .....	28
<b>4</b>	<b>Proposed New TCP Congestion Control Algorithm .....</b>	<b>30</b>
4.1	Reason for Using an End-to-End Scheme .....	31
4.2	Findings and Observations .....	31
4.2.1	General Observation – I.....	31
4.2.2	General Observation – II .....	33
4.3	Basic Structure of the Proposed Algorithm.....	34
4.4	Variables Used in the Algorithm.....	35
4.4.1	TC .....	36
4.4.2	DC.....	36
4.4.3	TDR .....	36
4.4.4	TDR_T .....	36
4.4.5	TDR_A .....	37
4.4.6	TI .....	37
4.4.7	TD.....	37
4.4.8	ITR.....	37
4.4.9	ITR_T .....	37
4.4.10	ITR_A.....	38
4.4.11	Latest_Timeout.....	38
4.4.12	cwnd.....	38
4.4.13	ssthresh .....	38
4.4.14	NR_cwnd .....	38
4.4.15	NR_ssthresh.....	39

4.4.16	K_cwnd.....	39
4.4.17	K_ssthresh .....	39
4.4.18	Counter_Refresh_Cycle.....	39
4.5	Actions Performed by K-Reno .....	40
4.5.1	Operations of Procedure "RECEIVE".....	40
4.5.2	Operations of Procedure "DUPACK_ACTION" .....	41
4.5.3	Operations of Procedure "TIMEOUT_ACTION" .....	42
4.5.4	Operations of Procedure "SLOWDOWN_ACTION" .....	43
4.6	Aggression and Fairness Issues .....	45
<b>5</b>	<b>Performance Evaluation of TCP K-Reno.....</b>	<b>48</b>
5.1	Modifications Performed in ns-2 Code .....	48
5.2	Simulation Setup.....	49
5.3	Simulating Bit Error in the Wireless Channel .....	51
5.4	Simulation Results and Analysis .....	52
5.5	Extensive Performance Comparison with TCP Westwood.....	56
<b>6</b>	<b>Conclusion .....</b>	<b>60</b>

## List of Figures

Figure 2.1 Slow-start in TCP .....	8
Figure 2.2 TCP Congestion Control Algorithm .....	11
Figure 2.3 Evolution of <i>cwnd</i> .....	12
Figure 2.4 Evolution of <i>cwnd</i> in a wired network .....	17
Figure 2.5 TCP in wired network (simulation setup) .....	17
Figure 2.6 Evolution of <i>cwnd</i> in a wired-cum-wireless network .....	21
Figure 2.7 TCP in wired-cum-wireless network (simulation setup) .....	21
Figure 5.1 Simulation setup of wired-cum-wireless network.....	49
Figure 5.2 Performance comparison of single TCP connection.....	53
Figure 5.3 Performance of single TCP connection (with no UDP traffic).....	55
Figure 5.4 Simulation setup for testing performance of TCP variants in a wired network .....	58

## List of Tables

Table 4.1 Effect of bit errors on timeouts and 3-dupacks (with no congestion) .....	32
Table 4.2 Effect of congestion on timeouts and 3-dupacks (Error Rate = 0.0%).....	33
Table 4.3 Variables and parameters used in the modified algorithm .....	35
Table 4.4 Values of some important parameters .....	36
Table 4.5 Effect of possible aggression of TCP K-Reno (Error Rate = 2.5%, $ITR_T = 0.045$ ).....	46
Table 4.6 Effect of possible aggression of TCP K-Reno (Error Rate = 2.5%, $TDR_T = 0.075$ ).....	46
Table 5.1 Performance comparison of single TCP connection .....	52
Table 5.2 Effect of UDP traffics on single TCP connection .....	54
Table 5.3 Performance comparison with two TCP connections .....	55
Table 5.4 Performance of TCP variants in Wired Network (with error).....	58

52



## List of Procedures

Procedure 4.1 RECEIVE .....	41
Procedure 4.2 DUPACK_ACTION .....	42
Procedure 4.3 TIMEOUT_ACTION .....	43
Procedure 4.4 SLOWDOWN_ACTION (EVENT_TYPE: Event) .....	44

## Acknowledgements

All praise to be ALLAH's. I thank ALLAH (SWT) for providing me the knowledge, capability and perseverance to successfully complete this thesis. Among His means, I express my heartily gratefulness and gratitude to my supervisor Dr. Md. Humayun Kabir, Associate Professor, Department of Computer Science and Engineering (CSE), Bangladesh University of Engineering and Technology (BUET). It is an honor and great opportunity for me to work with such a talented person and devoted researcher. His vast experience and in-depth knowledge in computer networks and related fields helped me enormously to figure out critical aspects of my research topic. His continuous support, scholarly guidance and enthusiastic attitude gave me the strength to complete my thesis without any hindrance.

I want to thank Dr. A. K. M. Ashikur Rahman, Assistant Professor, Department of CSE, BUET and one of the members of my M. Sc. Engg. Thesis Board of Examiners, for his invaluable support during my research work. Dr. Rahman's expertise in ns-2 simulator enabled me to design my simulation scenarios in realistic ways.

I also want to thank the other members of my M. Sc. Engg. Thesis Board of Examiners: Professor Dr. Md. Saidur Rahman, Professor Dr. Hafiz Md. Hasan Babu and Dr. Md. Mostofa Akbar for their valuable suggestions.

I express my gratitude towards my colleagues at Department of CSE, BUET for their caring support and encouragement that made my journey towards the completion of this thesis smooth and enjoyable.

Finally, I would like to express my ever gratefulness to my parents and other family members for their support, love and care towards me. Their presence makes me feel strong and confident in my academic and research works.

## Abstract

Transmission Control Protocol (TCP) congestion control algorithm deals with reducing the network load during congestion for achieving better throughput. It reduces the transmission window size after a retransmission timeout or reception of consecutive three duplicate acknowledgements (3-dupacks). It works well for the wired networks where most of the timeouts and 3-dupacks are the result of congestion. But we like to argue that in the wireless networks, where random segment loss due to bit errors is a dominant concern, the arrival of duplicate acknowledgements or even the retransmission timeouts do not necessarily denote congestion. In those cases, throttling transmission rate is not necessary. As the basic TCP congestion control algorithm cannot distinguish between congestion event and bit error event, it fails to perform well in wireless networks. In this thesis, we propose some modifications to the basic TCP congestion control algorithm so that its performance is enhanced in wireless networks. In particular, our algorithm refines the multiplicative decrease algorithm of TCP NewReno. We are using some statistical counters to track the frequencies of the occurrences of timeouts and 3-dupacks. Different ratios of these counter values are then used to differentiate a congestion event from a non-congestion event. We are also tracking the time difference between two consecutive timeouts to figure out whether timeouts are caused by network congestion or random bit errors. We tested our proposed algorithm using the Network Simulator version 2 (ns-2) and found that it shows better performance than any other TCP variants in wired-cum-wireless networks. Moreover, our algorithm is end-to-end in nature and modifies only TCP sender's algorithm.



# 1 Introduction

Advancements in wireless technology and ever-increasing need for all-time connectivity have made wireless networks a significant part of modern world. Wireless communication technology is playing an important role in access networks as evidenced by the widespread adoption of wireless local area networks (WLANs), wireless metropolitan area networks (WiMAX – the Worldwide Interoperability for Microwave Access), and cellular networks. Although wireless networks are quite different when compared to their wired counterpart, popular protocols and applications designed for and implemented in wired networks have found their way in wireless networks too. Most of the heavily used applications in both wired and wireless networks rely on the TCP/IP protocol suite.

Transmission Control Protocol (TCP) [1] is the principal transport protocol used in the Internet. TCP ensures a reliable, ordered, connection oriented, byte streamed full duplex communication over an unreliable medium. It was originally designed to provide reliable data delivery over conventional (wired) networks for a limited range of transmission rates and propagation delays. It performs both flow control and congestion control. The purpose of flow control is not to overwhelm the receiver of a TCP connection. During TCP communication, the TCP receiver constantly informs the TCP sender about its current buffer capacity and the TCP sender tunes its transmission rate accordingly. On the other hand, congestion control is a network wide issue. Its purpose is to control the transmission rate so that the sender does not transmit in excess of the capacity of the network. One of the strengths of TCP lies in its congestion control mechanism proposed in the cornerstone work by Van Jacobson [2]. Generally, the congestion information is not advertised by the congested nodes. The sending entity adjusts a congestion window based on successful transmissions

and timeouts and uses the congestion window as the maximum limit for transmission. Setting the congestion window too small might result in under utilization of network resources. On the other hand, a large congestion window may over feed the network that might result in dropping of segments at the congested nodes.

The congestion control algorithm used in the TCP/IP protocol suite [2], [3] is a sliding window mechanism that uses segment loss to detect congestion. The TCP sender probes the network state by gradually increasing the window of segments that are outstanding in the network until the network becomes congested and drops segments. Initially, the increase is exponential and this phase is called "*Slow-start*". This phase is intended to quickly grab the available bandwidth. When the window size reaches a slow-start threshold (called *ssthresh*), TCP enters into the second phase called "*Congestion Avoidance*", where the increase becomes linear. This is done to make the TCP sender less aggressive in probing for the available bandwidth. Clearly, it is desirable to set the threshold to a value that approximates the connection's "*fair share*". The optimal value for the slow-start threshold is the one that corresponds to the number of segments in flight in a pipe when TCP transmission rate is equal to the available bandwidth [4], i.e. when its transmission window is equal to the available bandwidth-delay product.

The current strategy taken by TCP in controlling network congestion is not adequate to perform well in wireless networks. When a loss is detected either through duplicate acknowledgements, or through the expiration of the retransmission timer, the connection backs off by shrinking its congestion window. If the loss is indicated by the three duplicate acknowledgement event, TCP Reno, one of the variants of the original TCP algorithm, attempts to perform a "*fast recovery*" by retransmitting the lost segment and halving the congestion window. If the loss detected through a retransmission timeout, the congestion window is reset to 1. In either case, when the congestion window is reset, TCP's window-based probing needs several round-trip times to restore its value to the near-capacity. This problem is exacerbated when random or sporadic losses occur. Random losses are losses not caused by congestion at the bottleneck link. They are common in the wireless channels. In this case, a burst of lost segments is erroneously interpreted by a TCP source as an indication of congestion, and dealt with by shrinking the sender's window. Such action, clearly,

does not alleviate the random loss condition and it merely results in reduced throughput. The larger the bandwidth-delay product, the larger the performance degradation caused by such action.

For this reason, the congestion control strategy employed by TCP works fine in wired networks, where most of the timeouts and delivery of misordered segments are caused by network congestion. However, in wireless networks, a good percentage of timeouts and reception of out of order segments happen due to the bit error rather than congestion. In those cases, throttling transmission rate does not help as this action results in under utilization of network bandwidth without any improvement in network activities. As the basic TCP congestion control algorithm cannot distinguish between congestion and bit error timeouts, it fails to give good performance in wireless networks. This thesis proposes a new TCP congestion control algorithm in order to get better performance in wireless networks. We have also designed our algorithm to perform well in wired and wired-cum-wireless networks. We have designed our algorithm with the following objectives in our mind:

- Develop a new TCP congestion control algorithm (we call it **TCP K-Reno**) in order to get better performance in both wired and wireless networks.
- Modify only the sending host software keeping the internal network devices and protocols unchanged.
- Develop an algorithm that will be able to differentiate between congestion and non-congestion losses and react accordingly.
- Keep TCP less aggressive during network overload.
- Utilize most of the available bandwidth in the network.
- Ensure a good throughput for connections that incorporate at least one wireless link (characterized by much longer round-trip time).
- Perform all the complex functionalities at the end hosts so that the network (i.e. routers) can be kept simple and is not filled with extra responsibilities to avoid affecting its packet forwarding speed and efficiency.

The rest of the thesis is organized as follows. Chapter 2 explains the basic TCP operations and its congestion control strategy in detail. Related research works done

by other researchers are discussed in Chapter 3. Our proposed new TCP congestion control algorithm is presented in Chapter 4. Detailed performance analysis of the proposed algorithm with the help of Network Simulator version 2 (ns-2) [5] is presented in Chapter 5. Chapter 6 concludes the thesis with some pointers for future study.

## 2 TCP Basics

TCP is a predominant transport protocol used in public and private IP (Internet Protocol) networks. IP, being a connectionless protocol, has no provision for detecting damaged, lost, duplicated or misordered data. That is why applications that require a reliable data transfer service use TCP or similar transport layer protocols (such as SCTP [6]) to establish virtual connections across an unpredictable and unreliable network. Without TCP, application developers would have to build reliability into each application.

The fundamental characteristics of TCP include the following:

- TCP is a connection-oriented service. A connection is established before data is being transmitted. Parameters that control the data transmission are exchanged between the sender and the receiver when the handshaking for the connection is done.
- TCP provides a reliable delivery service. While the data stream is transmitted, the receiving host sends back acknowledgements (ACKs) confirming that the data has been received in correct order and without errors. TCP source maintains a record of the segment that it has sent and waits for the ACK before sending the next set of segments. TCP source also starts a timer when it sends a segment and retransmits the segment if the timer expires before the ACK is received from the receiver.
- TCP source always attempts to fill the “*pipe*” between the sending and receiving hosts while adapting its transmission rate to avoid potential congestion in the network. TCP source continually monitors and modifies its



transmission rate so that the rate at which it injects segments into the network is just below the point at which segment loss starts to occur.

- All TCP connections are full duplex. This means that a TCP connection supports simultaneous transfer of data in both directions.

## 2.1 TCP Segments

The basic unit of transfer between two hosts in a TCP connection is called a *segment*. A segment consists of a TCP header and its associated data. Since each TCP segment is transmitted in an IP datagram and because IP datagrams can be reordered as they cross the network, TCP segments can arrive at TCP destination in a different order than that was originally followed by the TCP source to transmit them. TCP segments can also be corrupted, dropped, or duplicated along the way.

A TCP source assigns a *sequence number* to each byte in the stream that it transmits to the destination. A TCP header carries a 32-bit sequence number that is used to identify the TCP segment. The sequence number field in the TCP header is set to the sequence number that the source has assigned to the first byte of the transmitted stream. TCP destination keeps track of received segments and identifies the out of order segments using the sequence number present in the segment header.

## 2.2 TCP Acknowledgments

TCP uses acknowledgements (ACKs) to support the reliable transmission of data. When TCP source transmits segments, it expects TCP destination to acknowledge the segments when they are received. The ACK number used by a TCP destination is the number of the next byte in the stream that the destination expects to receive from the source. For example, if a TCP destination ACKs 1001, it informs a TCP source that it has successfully received all bytes up to and including byte 1000 and expecting the next segment with the first byte sequence number 1001.

## 2.3 TCP Receiver and Congestion Windows

As mentioned earlier, TCP uses a sliding window mechanism to control its rate of transmission. It is both receiver and network friendly. The actual transmission rate of TCP depends on two windows – the current receiver window that is advertised by the

receiver (called *rwnd*) and the current congestion window (called *cwnd*) that depends on sender's perceived bandwidth of the network. A TCP sender sets its transmission rate according to the minimum of the two windows to avoid both the receiver and the network overflow. The receiver always advertises the receiver window using the *window size* field in the TCP header. However, TCP has to estimate the available network bandwidth with the help of acknowledgements received from the receiver. TCP maintains a retransmission timer for every segment it transmits. If the timer times out before the arrival of the acknowledgement against a particular segment, TCP considers the segment has been lost and retransmits the lost segment. Moreover, TCP assumes that the segment has been lost due to buffer overflow in an intermediate router, i.e. congestion in the network, and throttles its transmission rate by reducing its congestion window with a hope that the reduced transmission rate will ease the congestion from the network.

## 2.4 TCP Congestion Control

TCP congestion control prevents a source from exceeding network capacity by adapting its transmission rate to avoid congestion in the network. This section sheds more light on the basic congestion control mechanisms used in TCP by discussing the followings in detail.

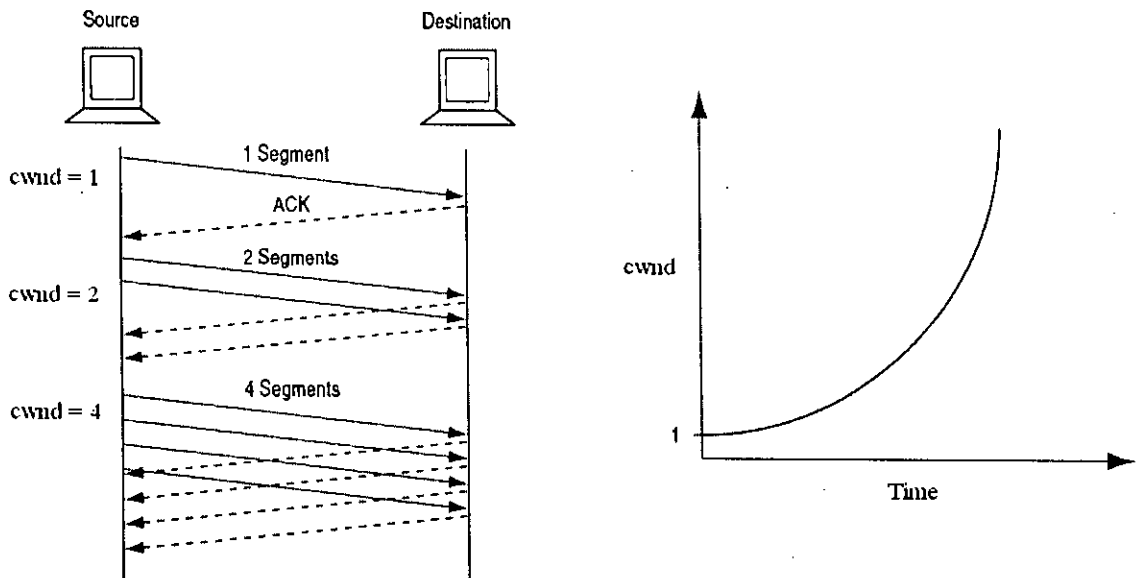
- Slow-start
- Congestion avoidance
- Fast retransmission
- Fast recovery

### 2.4.1 Slow-Start

When a TCP connection is first established, the TCP source remains a bit conservative and starts transmitting only a few segments, waits for the ACKs against those segments, and then gradually increases its transmission rate upon successive reception of acknowledgements. This allows the TCP source to probe the network gently to determine the amount of bandwidth that is available for the connection. This slow-start mechanism is used in the following cases.

- At the beginning of each new TCP connection.
- When an existing TCP connection is restarted after a long idle period.
- When an existing TCP connection is restarted after the retransmission timer expires.

As a result, the slow-start mechanism keeps TCP away from over feeding the network with too many segments when a new TCP connection is established or a congestion event is assumed on a running TCP connection.



**Figure 2.1 Slow-start in TCP**

Figure 2.1 illustrates the operation of TCP slow-start mechanism. In the slow-start mechanism the sender maintains a congestion window (*cwnd*) which represents its estimation of the traffic that the network can absorb without getting congested. When a TCP connection is first established, *cwnd* is initialized to the size of a single segment (called MSS – Maximum Segment Size) advertised by the host at the other end of the connection. A TCP source always transmits the data equal to the minimum of its *cwnd* (representing the congestion control administered by the sender) and the destination's advertised window (representing the flow control governed by the receiver).

TCP source initiates the slow-start by transmitting one segment and waiting for its ACK. When the ACK is received, the source increases *cwnd* from one to two, and two segments are sent (provided that the size of the receiver window is more than or equal to two segments). When these two segments are acknowledged, the source increases *cwnd* from two to four, and four segments are sent (again provided that the size of the receiver window is more than or equal to four segments). It continues to transmit according to the congestion window and doubles the congestion window until a threshold value (64 KB at the beginning) is reached. This threshold is called slow-start threshold (*ssthresh*). Whenever *cwnd* reaches this threshold value TCP exits from the slow-start phase and enters into the congestion avoidance phase.

## 2.4.2 Congestion Avoidance

As we said before, the exponential growth of *cwnd* continues until *cwnd* reach the *ssthresh*. From this point, the sender increases *cwnd* linearly (by at most one segment per round-trip time), allowing it to slowly increase its transmission rate. This region of *cwnd*'s evolution is called congestion avoidance.

When a TCP source discovers that a segment has been dropped by the network, it sets the *ssthresh* equal to one-half of its current value of *cwnd*. The source reduces its transmission rate by restarting the slow-start mode and exponentially increases the *cwnd* value until the new *ssthresh* is reached. At this point, TCP source enters into the congestion avoidance mode again.

Slow-start with congestion avoidance forces TCP sender to reduce the value of its current *cwnd* each time it experiences a segment loss. If the segment loss continues for a period of time, the volume of traffic injected into the network by the TCP source decreases dramatically which allows routers to drain out their congested queues.

The TCP source can determine that a segment has been dropped by the network or got damaged in two ways:

- Through the reception of duplicate acknowledgements.
- Through the expiration of the retransmission timer.

The absence of a single segment in the middle of a transmission window causes the destination to immediately generate a duplicate acknowledgement. For example, when a destination receives all of the data in the stream up to byte 1000, it responds with an ACK of 1001 indicating that the next segment the destination expects to receive begins with byte 1001. If a segment is dropped by an intermediate router, the destination TCP continues to buffer the subsequent segments as they arrive, however, it continues to ACK 1001 since it has not received the expected segment. Generally, the receipt of duplicate ACKs means that the segment has been delivered out of order, however, the TCP source uses the receipt of three duplicate ACKs as an indication of segment loss.

The loss of the last segment in a transmission window does not generate a duplicate ACK. It rather causes the retransmission timer of the TCP source to time out due to the absence of subsequent out of order segment delivery. TCP retransmission timer supports adaptive retransmission by changing the timeout value as the sample round-trip times (RTTs) of the connection constantly changes with the network load. If the retransmission timer expires before the segment has been acknowledged, the source assumes that the segment was either lost or corrupted and retransmits the segment.

### **2.4.3 Fast Retransmission**

As discussed earlier, TCP assumes that a segment has been dropped when it receives duplicate ACKs although the reception of duplicate ACKs can also mean that the segment simply got out of order. Instead of responding immediately to a duplicate ACK by retransmitting the lost segment, the source TCP waits until it receives three duplicate ACKs (3-dupacks) in the fast retransmission strategy. After receiving three duplicate ACKs, the TCP source retransmits the missing segment, without waiting for the retransmission timer against that missing segment to expire. Fast retransmission enhances TCP performance in the following ways:

- Eliminates unnecessary segment retransmission, hence, the waste of network bandwidth if the segment simply becomes out of order and not dropped.
- Provides higher channel utilization and connection throughput.
- Does not force TCP to wait for the retransmission timer to expire before resending a potentially lost segment.

Fast retransmission strategy, however, goes back to the slow-start phase and reduces its *ssthresh* value to the half of its current *cwnd* value.

The flowchart in Figure 2.2 summarizes the operation of TCP congestion control involving slow-start, congestion avoidance and fast retransmission.

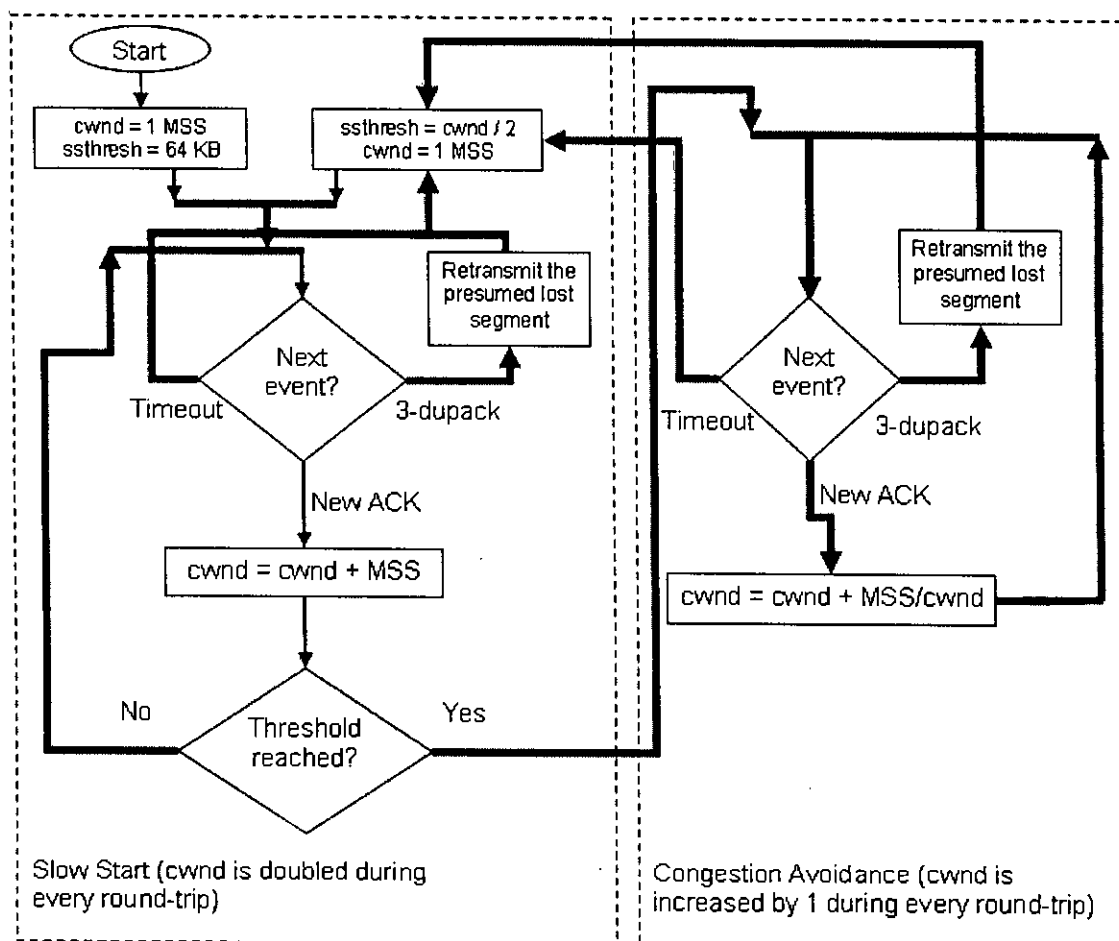


Figure 2.2 TCP Congestion Control Algorithm

### 2.4.4 Fast Recovery

When the TCP source receives duplicate ACKs it means that the data is still flowing towards the destination, which allows the destination to generate duplicate ACKs. In this case, a TCP source, while using the fast recovery strategy, does not suddenly reduce the flow of data by returning to the slow-start phase. Instead, the TCP source sets *cwnd* to the half of its current value plus 3 MSS after responding to the receipt of three duplicate ACKs by retransmitting the lost segment. It artificially "inflates" the congestion window by three, which is equal to the number of segments those have already left the network and has been buffered by the receiver. It also sets *ssthresh* to

the half of the previous *cwnd* value and enters into the congestion avoidance mode. For each additional duplicate ACK received, *cwnd* is incremented by 1 MSS. When the next ACK arrives that acknowledges the new data, *cwnd* is set to *ssthresh*. This strategy provides better overall throughput for the TCP connection when segments simply get reordered in the network or a single segment is lost from a flight of segments.

Fast recovery prevents the TCP connection pipe from getting completely empty after the fast retransmission of a single lost segment. This enhances TCP performance by eliminating the need to return to the slow-start mode and filling the TCP connection pipe slowly after a single segment loss. However, while the fast recovery strategy improves TCP performance when a single segment is dropped from a window of data stream, it cannot do the same when multiple segments are dropped.

Figure 2.3 illustrates the evolution of *cwnd* in a typical TCP connection using the fast recovery strategy.

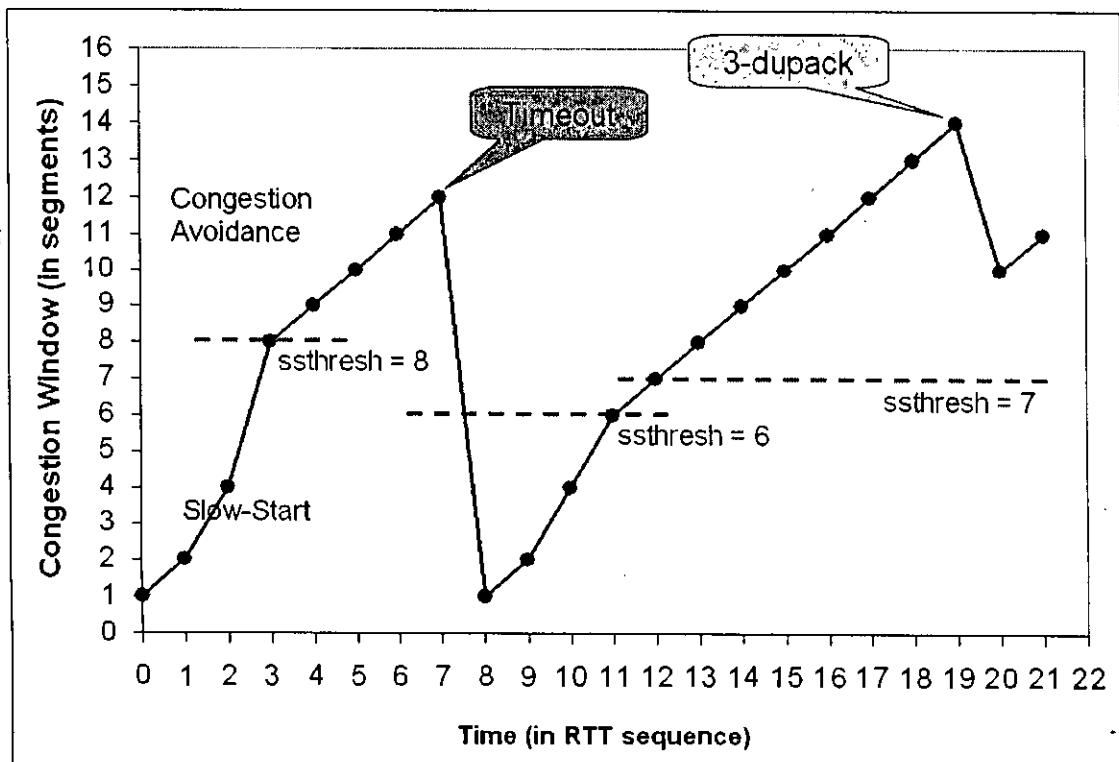


Figure 2.3 Evolution of *cwnd*

## 2.5 Major TCP Variants

Over the years, researchers have proposed and implemented a good number of variants of original TCP algorithm to make TCP performing well in all possible types of networks. The following subsections described some important variants of TCP along with their working strategy and limitations.

### 2.5.1 TCP Tahoe

Three key algorithms discussed earlier: slow-start, congestion avoidance and fast retransmit, were originally proposed by Van Jacobson [2] and implemented in TCP Tahoe under “4.3 BSD Tahoe TCP” module in 1988. TCP Tahoe shows good performance in those networks where most of the segment losses are due to congestion. However, it shows poor performance when segments are delivered out of order or when segments get damaged due to non-congestion related events (e.g. random bit errors). We have observed this poor performance of TCP Tahoe using simulation. Performance of TCP Tahoe in wired-cum-wireless networks has been presented in Chapter 5.

### 2.5.2 TCP Reno

TCP Reno first implemented in “4.3 BSD Reno TCP” in 1990, supports all of the Van Jacobson’s algorithms and extends TCP by introducing the fast recovery algorithm. By supporting fast recovery, TCP Reno overcomes the throughput performance limitations of TCP Tahoe that occur when a single segment is lost or misordered. But it cannot handle multiple segment losses in a single window efficiently. This poor performance of TCP Reno has been observed by us using simulation. Performance of TCP Reno in wired-cum-wireless networks have been presented in Chapter 5.

### 2.5.3 TCP NewReno

TCP NewReno [7] enhances TCP throughput when multiple segments are dropped from a single window using TCP Reno connections. When multiple segments are dropped from a single window, the TCP source enters into the fast retransmission phase after receiving 3-dupacks for the first lost segment. After retransmitting the first lost segment TCP NewReno enters into the fast recovery phase. While the retransmitted segment is in transit, the TCP source continues to receive duplicate



acknowledgements. These duplicate acknowledgements report the loss of that retransmitted segment and continue to do so until the retransmitted segment reaches the destination. The ACK that is generated in response to the successful reception of the retransmitted segment reports the second missing segment in the same transmission window. This ACK is referred to as a “*partial ACK*”. TCP Reno’s reaction to this event is to deflate the congestion window and exit from the fast recovery phase immediately as this ACK is not a duplicate acknowledgement of a previously received acknowledgement. This causes a TCP sender to enter into the fast retransmission and the fast recovery phases again after receiving three duplicate ACKs for the second lost segment and results in further reduction of *cwnd* and *ssthresh* values. TCP NewReno solves this problem by not exiting from the fast recovery phase when it receives a *partial ACK*. During the fast recovery phase when a TCP NewReno sender receives *partial ACKs*, the acknowledgement number present in the TCP header of the ACK segment informs the sender about the successive lost segments. TCP NewReno immediately retransmits the presumed lost segment after receiving the *partial ACK* and remains in the fast recovery phase. This strategy prevents reduction of the *cwnd* by entering into the fast retransmission phase multiple times. Thus TCP NewReno overcomes the throughput performance penalty when multiple segments are dropped from a single window. Although it is a superior protocol compared to TCP Tahoe and TCP Reno it fails to ensure a good throughput in wired-cum-wireless networks where most of the segments get damaged due to random bit errors.

#### 2.5.4 TCP Vegas

TCP Vegas [8] dynamically increases and decreases the transmission window size according to the observed RTT of previously sent segments. If the observed RTT becomes large, TCP Vegas assumes that the network is experiencing congestion, and it reduces the window size. Likewise, if the observed RTT becomes small, TCP Vegas concludes that the network is not experiencing congestion and it increases its window size for better utilization of the available bandwidth. Another modification introduced by TCP Vegas is that during the slow-start, the rate of *cwnd* increase is different than that of TCP Tahoe and TCP Reno. In TCP Vegas, *cwnd* is doubled with the receipt of every other ACK instead of every ACK.

### 2.5.5 TCP Westwood

TCP Westwood (TCPW) [9], [10] is a modified version of TCP Reno. TCPW enhances the window control and back off process. Here, a TCP sender performs an end-to-end estimate of the bandwidth available along the connection by measuring the rate of returning acknowledgements and the amount of bytes delivered to the receiver during a certain interval. Whenever a sender perceives a segment loss (i.e. a timeout occurs or 3-dupacks are received), the sender uses the bandwidth estimate to properly set the congestion window (*cwnd*) and the slow-start threshold (*ssthresh*). By backing off the *cwnd* and the *ssthresh* to the values those are based on the estimated bandwidth rather than simply halving the current values as Reno does, TCP Westwood avoids overly conservative reductions of *cwnd* and *ssthresh*; and thus it ensures a faster recovery. The benefits of TCPW include better throughput, goodput, and delay performance, as well as fairness even when competing connections differ in their end-to-end propagation times. TCPW is also effective in handling wireless loss. This is because TCPW uses the current estimated rate as the reference for resetting the congestion window. The current rate is only marginally affected by loss (as long as the loss is relatively small compared to the data rate). Although TCPW shows better performance than other TCP variants in contention free wireless networks (e.g. a dedicated channel between a VSAT and a satellite), it fails miserably when deployed in multi-access wireless networks where multiple wireless nodes share the same radio frequency using collision avoidance type of scheme. TCPW relies on a consistent flow of acknowledgements from the receiver in order to calculate a near-to-actual estimate of the available network bandwidth. Whenever the acknowledgement stream is disrupted, TCPW's estimation process gives wrong results and degrades the overall performance of a TCP connection. We have presented detail analysis of TCPW's performance in multi-access wireless networks in Chapter 5.

### 2.5.6 TCP SACK

TCP selective acknowledgement (SACK) [11] enhances the throughput performance of TCP Reno when multiple segments are dropped from a single window. When a TCP receiver observes that the arriving segments are not continuous (the segments are out of order), it responds to the TCP sender with ACKs that contain the SACK option. This option specifically tells the TCP sender which segments have been received by

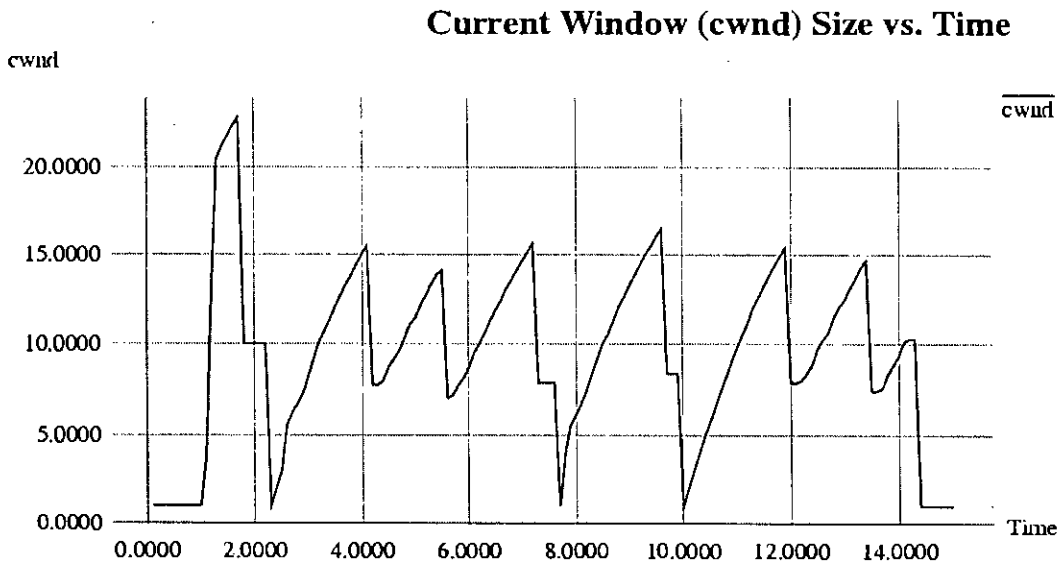
the destination and the sender retransmits only the missing segments. TCP SACK needs modification in both the sender and the receiver side protocol stack to incorporate the SACK option.

### 2.5.7 TCP D-SACK

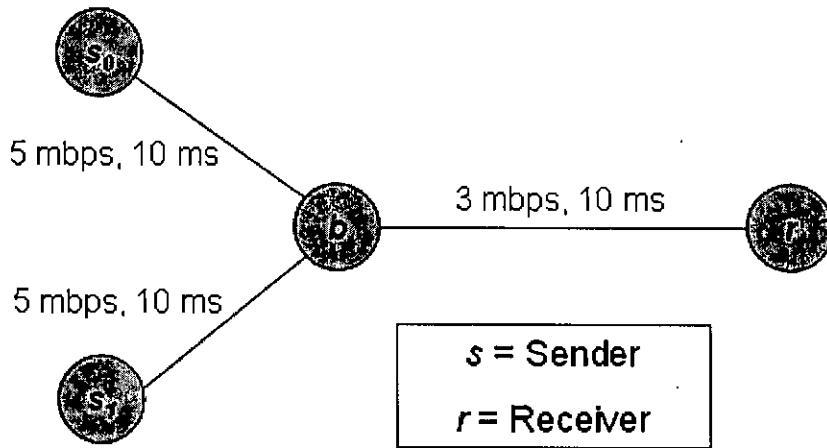
The duplicate-SACK (D-SACK) extension [12] allows a TCP receiver to use a SACK to report the receipt of duplicate segments. This extension allows the TCP sender to identify the segment received by the TCP receiver, including duplicate segments. If the TCP sender determines that the destination TCP received two copies of a segment and that the retransmission of the duplicate segment was unnecessary, the TCP sender can undo the halving of *cwnd*. The D-SACK extension overcomes the throughput performance penalty that results from halving the congestion window. However, this strategy requires modifications in both TCP sender and receiver protocol stack.

## 2.6 TCP in Wired Network

As mentioned earlier, TCP was originally designed for networks where loss is only due to congestion. Wired networks show this characteristic. In wired network any indication of segment loss can be considered as network congestion. For this reason, TCP's reaction to congestion, i.e., throttling transmission rate after timeout or reception of consecutive three duplicate acknowledgements works fine for wired networks. We have studied the evolution of congestion window (*cwnd*) of a typical TCP connection in a wired network using ns-2 simulator. The result obtained from an ns-2 simulation run by establishing a TCP connection between two nodes in a wired network is shown in Figure 2.4. Figure 2.5 shows the network configuration that was used to run the simulation. Here, the link between *b* and *r* acts as the bottleneck link. A TCP connection between *s<sub>0</sub>* and *r* is used. A CBR (Constant Bit Rate) traffic connection using UDP (User Datagram Protocol) between *s<sub>1</sub>* and *r* with 1000 bytes packet size and 0.75 mbps data rate is used to induce congestion in the network. The simulation was run for 15 seconds.



**Figure 2.4 Evolution of *cwnd* in a wired network**



**Figure 2.5 TCP in wired network (simulation setup)**

From Figure 2.4, it is evident that after throttling the transmission rate due to expiration of retransmission timer or reception of 3-dupack, TCP can regain a good throughput in a short time. From this simulation result we can conclude that TCP's performance in wired network is excellent.

## 2.7 Characteristics of Wireless Networks Affecting TCP's Performance

There are several characteristics that are unique to wireless environment that make it challenging to adapt TCP to work effectively. The characteristics of wireless medium

differ significantly than that of wired medium. The major factors affecting TCP's performance in wireless environment are described below [13].

### 2.7.1 Limited Bandwidth

Bit rates of 100 Mbps are common on wired LANs. Optical links provide data rate of the order of gigabits to terabits per second. However, the current wireless standard, for example the IEEE 802.11g standard for Wireless LAN, offers maximum raw bit rates of 54 Mbps. Thus available bandwidth is one of the major bottlenecks that degrade the throughput of TCP on wireless medium.

### 2.7.2 Long Round-Trip Times

In general, wireless media exhibits longer latency delays than wired media. The rate at which the TCP sender increases its congestion window is directly proportional to the rate at which it receives ACKs from the receiver. Due to longer round-trip times, the congestion window increases at a much lower rate with wireless links. This is imposing a limit on the throughput of TCP on wireless links.

### 2.7.3 Random Losses

The transmission losses on wireless medium, bit error rate (BER) of the order of  $10^{-3}$  to  $10^{-1}$ , are significantly higher than that on wired medium, BER of the order of  $10^{-8}$  to  $10^{-6}$ . These losses result in segment drops and hence the sender does not receive acknowledgements before the retransmission timer times out. This causes the sender to retransmit the segment, exponentially back off its retransmission timer and lowers its congestion window to one segment. Repeated errors result in low throughput. The loss of segments on wireless link, which in general is the last hop, results in end-to-end retransmission. This also causes traffic overload on the wired links.

### 2.7.4 User Mobility

In case of cellular networks when a user moves from one cell to another, all the necessary information has to be transferred from the previous base station to the new base station. This process is called *Handoff*, and depending on the technology used, there might be short duration of disconnection. TCP attributes delays and losses caused by these short periods of disconnection to congestion and triggers congestion control and avoidance mechanism. This again results in reduced throughput. In case

of ad hoc networks, mobile nodes can move randomly causing frequent topology changes. This causes segment losses and forces mobile hosts to initiate route discovery algorithms frequently. The overall result is significant throughput reduction.

### **2.7.5 Power Consumption**

The retransmission caused by frequent segment losses result in longer connection duration, hence, higher power consumption. Power consumption is a very important factor in case of battery operated devices like laptops, personal digital assistants (PDAs) and wireless phones. For this reason, it is better to keep the number of retransmissions and connection duration low in wireless networks.

### **2.7.6 Medium Access Control (MAC) Layer Activities**

In a typical wireless network (e.g. IEEE 802.11) all the nodes share the same radio frequency. In order to ensure collision avoidance, the MAC layer of a transmitting wireless node reserves the wireless channel for some time during which the neighboring nodes are strictly prohibited to transmit. This introduces waiting time and increases the communication delay among all other wireless nodes and eventually affects the overall TCP throughput in the wireless networks.

## **2.8 Performance of TCP Congestion Control**

### **Algorithm in Wireless Network**

In wireless networks, it is evident that the packet losses may occur not only for the congestion in the network but also for:

- Transmission errors in wireless links due to fading, shadowing, jamming etc.
- Handoff between cells due to user mobility.
- Temporary disconnection between transceivers.

In wired connections, where the bit error rate (BER) is negligible and handoff or temporary disconnection is almost non-existent, TCP's congestion control strategy is sufficient as most of the segment losses are due to congestion at the bottleneck nodes.

If a packet is lost due to some non-congestion related events but TCP's congestion control policy is activated that will certainly reduce the throughput of the connection. When a packet is lost due to bit error, handoff or temporary disconnection there is no benefit to reduce the transmission rate. In case of transient network errors the future packets may not suffer the loss. So categorization of packet losses is very important for efficient performance of TCP over wireless networks. Packet losses due to bit errors, mobility and hard handoff need to be handled differently than that of due to network congestion. In networks with large bandwidth-delay product, reducing the congestion window inappropriately will lower the performance of the connection severely as it will take much longer for the acknowledgements to arrive and the congestion window to increase.

We have analyzed the evolution of congestion window (*cwnd*) of a typical TCP connection in a mixed environment using ns-2 simulator. The result obtained from an ns-2 simulation run by establishing a TCP connection between two nodes in a mixed network is presented in Figure 2.6. Figure 2.7 shows the network configuration that was used to run the simulation. In Figure 2.7  $s_0$ ,  $s_1$  and  $b$  belongs to the wired domain and  $r$  belongs to the wireless domain.  $BS$  is the base station that acts as the gateway between the wired and the wireless domain. The link between  $b$  and  $BS$  acts as the bottleneck link. A TCP connection between  $s_0$  and  $r$  is used. A CBR (Constant Bit Rate) traffic connection using UDP (User Datagram Protocol) between  $s_1$  and  $r$  with 1000 bytes packet size and 0.75 mbps data rate is used to induce congestion in the network. The simulation was run for 15 seconds.

## Current Window (cwnd) Size vs. Time

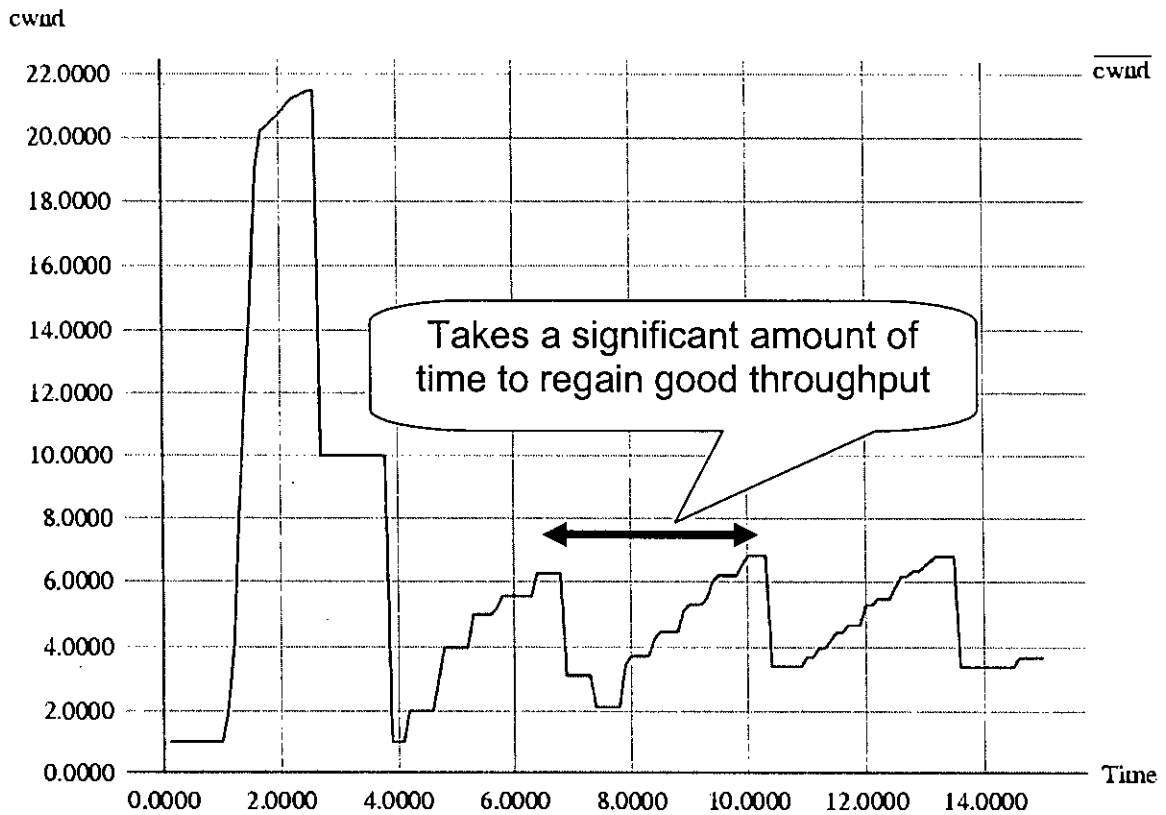


Figure 2.6 Evolution of *cwnd* in a wired-cum-wireless network

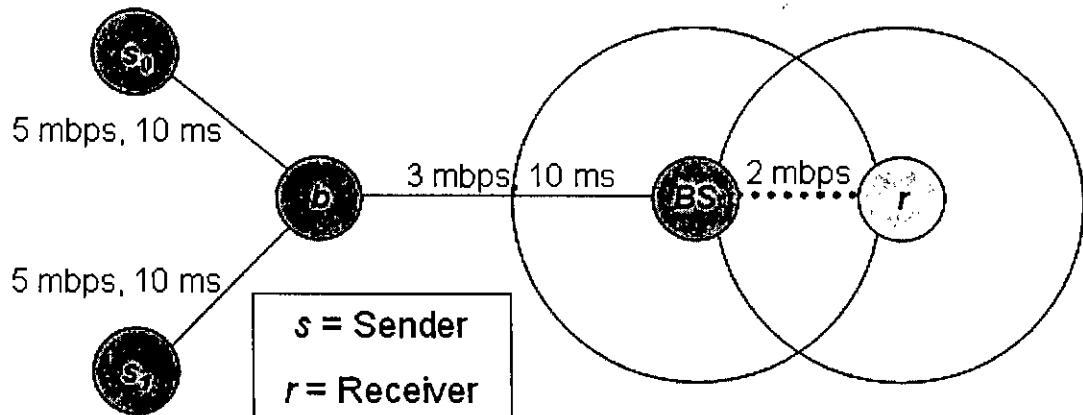


Figure 2.7 TCP in wired-cum-wireless network (simulation setup)

From Figure 2.6, it is quite evident that in the wireless networks whenever TCP throttles its transmission rate after an indication of packet loss, it takes a significant amount of time to regain the previous throughput. If the packet was lost due to bit error then this reduction in transmission rate does not bring any good. This situation



gets worse when we have a network with large bandwidth-delay product. In this type of networks, higher delay results in high round-trip time. Hence acknowledgements of a TCP connection arrive at the TCP source at a much slower rate. If TCP throttles its transmission rate due to a non-congestion related event, it will not only take significant amount of time to regain a good throughput but also will keep a significant amount of bandwidth unutilized. For this reason, differentiating between congestion and non-congestion related losses are very important if we want TCP to show better performance in a heterogeneous network environment.

The next chapter discusses some state of the art research works focused on improving performance of TCP in heterogeneous networks.

## 3 State of the Art

Effective error and congestion control for heterogeneous (wired and wireless) networks has been an active area of research recently. Research works in [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26] and [27] have studied and analyzed congestion control for wireless networks. The major focus of the most of the aforementioned works is to nullify the adverse effects of TCP's basic congestion control mechanism in wireless networks. In the following subsections different strategies proposed for improving TCP's performance in heterogeneous networks are discussed.

### 3.1 Link Layer Solutions

The link layer (LL) protocol runs on top of the physical layer and has immediate knowledge of the transmission medium and dropped frames. At the same time, the LL protocol has more control over the physical layer protocol. Hence, alleviating the wireless medium inefficiencies at the LL provides the transport layer protocol with a dependable communication channel, similar to a wired one. This way, the transmission media heterogeneity introduced in the network remains transparent to the existing software and hardware infrastructure, and does not necessitate any changes to current TCP implementations.

Asymmetric Reliable Mobile Access in Link Layer (AIRMAIL) [14] is a popular link layer protocol designed for indoor and outdoor wireless networks. It provides a reliable link layer by using local retransmissions and forward error correction (FEC) at the physical layer. The protocol is asymmetric to reduce the processing load at the mobile host. The asymmetry is needed in the design because the mobile terminals have limited power and smaller processing capability than the base station. The

asymmetric design places the bulk of the intelligence in the base station, allows the mobile terminal to combine several ACKs into a single ACK to conserve power, requires the base station to send periodic status messages, and forces the acknowledgement from the mobile terminal to be event driven. The side-effect of the asymmetric design is that no error correction can be done until the ACKs arrive which can cause TCP to time out if the error rate is high.

There are many adverse interactions between TCP and link layer for which the end-to-end performance of TCP does not improve when this type of link layer solutions are used.

1. Timer interactions – Independent timers at both layers could trigger at the same time, leading to redundant retransmissions at both layers and degraded performance. As a result the transport sender is not shielded from the problems of the wireless link.
2. Fast retransmission interactions – This arises when a link layer protocol achieves reliability by local retransmissions, but does not preserve the in-order sequential delivery of the TCP segments to the receiver. In this case, although the local recovery occurs the receipts of later segments causes duplicate ACKs from the receiver, which leads to redundant retransmissions, sender window reductions, and throughput reduction.
3. Large RTT variations – The retransmission at the link layer results in long latencies and variable RTTs at the TCP sender.

## 3.2 Base Station Dependent Approach

In [16] Balakrishnan *et al.* proposed the design and implementation of a simple protocol called “Snoop” for the scenario where a fixed host is communicating with a mobile host with the help of a base station. Here, the TCP implementation at the fixed host does not need any modification. The network layer code at the base station is changed to implement the Snoop protocol. No transport layer code runs at the base station. The packets sent from the fixed host are buffered at the base station before delivering them to the mobile host. When the Snoop agent residing at the base station receives a duplicate acknowledgement against a lost packet at the mobile host, it retransmits the missing packet locally to the mobile host and conceals the packet loss

events from the sender and hence prevents it from reducing its congestion window to maintain a good throughput.

The Snoop protocol is not scalable. When a large number of concurrent TCP connections are active through a base station, the base station has to buffer all the segments from all TCP connections to retransmit, whenever necessary. The base station will exhaust the buffer space very quickly and will not be able to store new segments. Eventually, the base station will fail to conceal the losses. Moreover, packet loss handling will put extra load on the base station which will adversely affect its other performance.

As discussed earlier, one problem with TCP on wireless networks is that it cannot distinguish the exact reason of the segment loss. The delay characteristics shown when a wireless host moves to a different network is different from those that are shown when it moves from one cell to another cell in the same network. The data loss due to these two types of mobility is also different from the data loss due to congestion in the wired network. TCP is not able to distinguish these losses and interprets them as congestion and invokes the undesirable congestion control mechanisms. Mobile TCP [17] distinguishes the segment losses due to handoff and those due to interface switching. It lets the base station tell the sender whether the loss is due to handoff in the same network or it is due to interface switching. The sender then marks the segments and retransmits them once the mobile host has completed handoff. In case of interface switching, the wireless host may enter into a new network which may not have the same network characteristics as the previous one. For this reason, when the TCP sender knows about the interface switching it resets window size (*cwnd*), *ssthresh*, the estimation of round-trip time (RTT) and retransmission timeout (RTO) values, and begins slow-start. However, if the wireless host has moved to a cell in the same network then the values of *cwnd* and *ssthresh* are halved and the RTT value is kept the same. This algorithm performs well knowing the cause of the segment loss. Though it handles the handoffs well it does not consider the bit error characteristics of the wireless link.

### 3.3 Using Redundant Error Correcting Segments

In [18] Subramanian *et al.* proposed an enhancement to TCP called LT-TCP (Loss Tolerant TCP) that performs well in extreme wireless environments. They showed that after certain bit error rate, where the original TCP fails miserably, LT-TCP shows good performance. Their work relies on ECN (Explicit Congestion Notification) from the network routers. LT-TCP uses FEC (Forward Error Correction) mechanism to recover segment errors and losses. Based on the current network condition, a certain amount of error correcting segments are pre-generated and kept in the transmission queue. Some of these are called PFEC (Proactive Forward Error Correction) segments and are sent along with the new data segments. The rest are called RFEC (Reactive Forward Error Correction) segments and are sent during retransmission of previously sent segments. LT-TCP also adaptively manages the maximum segment size to ensure a certain minimum number of FEC segments in the transmission window.

Using slots from the transmission window for additional error correcting segments results in reduced number of slots for the actual data. This reduces the throughput of a TCP connection. Moreover, generation of error correcting segments at the sender consumes processor time. Similarly, the use of error correcting segments at the receiving end in order to recover damaged segments also needs some extra processing activities.

### 3.4 Router Assisted Approach

In [19] Lien *et al.* proposed a router-assisted approach to solve the congestion control problems in TCP. This approach asks some help from the network. Basically, when congestion builds up in certain part of a network, the congested router informs the source about the congestion setting ECN (Explicit Congestion Notification) bit of the packets destined back to the source. Upon reception of packets with ECN bit set, the sender can reduce its transmission rate. This technique effectively distinguishes the segment loss due to congestion from the segment loss due to an error. It assumes that the routers present in the path are ECN enabled. However this approach gives some extra loads on the intermediate routers that deteriorates their packet switching performance. It is also hard to ensure that all the routers in the network will be ECN enabled.

### 3.5 Split Connection Approaches

The advocates of these schemes claim that since two completely different classes of sub networks (wired and wireless) are involved in wired-cum-wireless networks, the TCP connection could be split into two connections at the point where the two sub networks meet, i.e. at the base station. These approaches completely hide the wireless link from the sender by terminating the TCP connection at the base station. The base station keeps one TCP connection with the fixed host, while it uses another connection with a protocol specially designed for better performance over wireless links for the mobile host. The base station acknowledges the segments as soon as it receives them. An acknowledgement can arrive at the sender even before the corresponding segment has been received by the receiver. The base station forwards the segments and buffers a segment until it receives the acknowledgement from the mobile host. A base station transparently transfers state information to another base station during handoffs.

Indirect-TCP (I-TCP) [20], uses above TCP splitting approach with different flow control and congestion control mechanism on the wireless link and on the fixed network, allowing faster reaction to mobility and wireless link breaks. However, I-TCP has following drawbacks.

1. As there are two separate connections for every TCP session, every packet needs separate processing for each connection. So every packet suffers a certain amount of processing overhead while switching from one connection to another.
2. End-to-end semantic of TCP ACKs is violated in split connection.
3. Complex hardware and software are required at the base station.
4. The base station needs large buffers in case of heavy traffic. Moreover, extra loads are given on the base station that may affect its usual activities. This is not a scalable solution.
5. If there are frequent handoffs then the overhead related to the connection state transfer between the base stations may be large and may add delays.

## 3.6 End-to-End Mechanisms

End-to-end (E2E) mechanisms solve the wireless loss problem at the transport layers of the sender and receiver. All the TCP variants discussed in Chapter 2 fall into this category. Among them TCP Westwood has been specially developed for wireless networks and large bandwidth-delay networks. Some other end-to-end proposals for improving TCP performance in wireless networks are described in this section.

In [21] Tsaoussidis *et al.* proposed an end-to-end proposal called TCP Probing. In this scheme when a data segment is delayed or lost, the sender enters into a probe cycle instead of retransmitting and reducing the congestion window size. In a probe cycle only the probe segments are exchanged between the sender and receiver to monitor the network and the regular transmission is suspended. The probes are TCP segments with header option extensions and no payload. A probe cycle terminates when the sender can make two successive RTT measurements with the aid of receiver probes. In case of persistent error, TCP decreases its congestion window (*cwnd*) and *ssthresh*. For transient random error, the sender resumes transmission at the same window size that it was using before entering into the probe cycle. Although the probe segments are small, they increase the network load even when the network is highly congested.

The Eifel detection algorithm [22], [23] allows a TCP sender to detect a posteriori whether it has entered into the loss recovery phase unnecessarily. This algorithm tries to nullify the effects of spurious timeouts i.e. timeouts that would not have occurred had the sender waited “long enough”, and spurious fast retransmits that occur when segments simply get re-ordered in the network before reaching the receiver. It requires the TCP timestamp option, defined in RFC 1323 [28], be enabled for a connection. When the timestamp option is used, the TCP source writes the current value of a “timestamp clock” into the header of each outgoing segment. The TCP destination then echoes those timestamps in the corresponding ACKs according to the rules defined in [28]. The TCP source always stores the timestamp of the first retransmission irrespective of its reasons; whether it was triggered by an expiration of its retransmission timer or by the receipt of 3-dupack. Based on the timestamp on the first accepted ACK that arrives during the loss recovery phase it decides whether the loss recovery phase was entered into unnecessarily. It is a reactive solution. During a timeout or 3-dupack event, the transmission rate is throttled first. Based on the

timestamp value, it may restore the previous transmission rate, however, it reduces the connection throughput for a while. Moreover, adding TCP timestamp in every segment imposes certain amount of overhead in the communication.

From the above discussions we can see that the strategies proposed in the recent research works in order to improve TCP's performance in mixed networks are not completely flawless. There are occasions when some of those strategies fail to provide sufficient support to increase throughput of a TCP connection. So there exists some scope to look into this matter from a new perspective. In the following chapter, our proposed TCP congestion control algorithm is presented that explores a new dimension in solving the problem of TCP's degraded performance in wireless networks.



# 4 Proposed New TCP Congestion Control Algorithm

In order to control congestion TCP throttles its transmission rate so that the overloaded routers do not get flooded with the new packets and get some time to drain out their queues without dropping the packets. This action taken by TCP helps the network to ease the congestion if there is real congestion in the network. However, if packets are lost due to bit error then there is no gain in reducing the transmission rate. In this case, the sender should continue transmitting at the original rate and try to deliver as many packets as possible in the midst of random bit errors. Throttling transmission rate will not do any good in this scenario. The probability of bit errors in wireless channels is higher than the probability of bit error in wired media. The original TCP cannot distinguish between the packet loss due to congestion and the same due to random bit errors and activates congestion control in both cases. This is causing performance penalty to TCP connections in the presence of wireless link in a network. Hence, some new strategies should be introduced in TCP such that it can detect packet loss due to bit error and act accordingly.

In this thesis, we have developed some new strategies that can be easily integrated with current TCP variants so that the TCP sender can distinguish between segment loss events due to congestion and those due to random bit errors. Our strategies need to modify only the sender side of TCP entity.

## 4.1 Reason for Using an End-to-End Scheme

We have opted for an end-to-end (E2E) proposal for a good number of reasons. Firstly, E2E proposal always keeps the end-to-end semantic of TCP intact. There is no proxy or relay agent necessary in E2E schemes. Secondly, E2E schemes do not place any extra burden on the internal network routers. If routers are required to help TCP entities in determining the current network condition, then it will slow down the routers' routing functionalities and create the bottleneck in the network. It is more critical for the core routers in the network since they have to route a huge volume of packets every moment. Finally, E2E schemes do not need to take help from the link layer that keeps the layered architecture layered and transparent.

## 4.2 Findings and Observations

In order to help the sending entity to distinguish between segment loss event due to congestion and the same due to bit error, there must be some extra checking through which the sending TCP entity will be able to detect false congestion alarm. Unfortunately, the information that is typically available to the sending TCP stack is not enough to accurately determine false congestion alarm. We have examined the characteristics of TCP operations in both wired and wireless networks and found two possible ways through which the TCP sender will be able to detect false congestion alarm with good precision.

### 4.2.1 General Observation – I

When there is little or no congestion in the network, or no bit errors in the packets, the TCP sender will experience less timeouts. Due to reordering of segments in the network, there might be few 3-dupacks. There will be a continuous flow of acknowledgements when the network is in non-congestion state and the transmission rate of the sender will not be throttled that much due to TCP's fast retransmit and fast recovery strategy. The TCP sender will only experience the retransmission timer timeout event whenever the congestion window size reaches the bandwidth limit of that instant. However, these events will be less frequent during non-congestion state due to slow-start and congestion avoidance strategy used by TCP.

Let us examine what will happen when there is less congestion in the network but the probability of random bit error is high. If the packets flowing through the network experience bit error then some packets will be corrupted or completely damaged. The receiver will detect this while verifying the checksum present in the received packet/segment and will reject it. So the order of segment reception will be changed in the receiver. Since there is no congestion in the network at this time, the traffic flow in the network will not be changed. There will be a steady stream of segments in transit from the source to the destination. If the next segments in sequence do not suffer from bit error then they will be received successfully by the receiver but will be regarded as out of order segments. This will result in generation of duplicate acknowledgements by the receiver. If there were many segments in transit, a significant amount of duplicate acknowledgements will be generated by the receiver and received by the sender.

The continuous flow of acknowledgements from the receiver will prevent timeouts from being occurred at the sender. So the number of 3-dupacks experienced by the sender will be much higher than the number of timeouts. This phenomenon can be used during a timeout or a 3-dupack event to decide whether there is real congestion in the network, or the segment loss occurred due to random bit error.

So, we can keep a running count of the number of timeouts and the number of 3-dupacks experienced during an interval. Whenever the sender experiences a timeout or 3-dupack event, it will compute the ratio of the number of timeouts to the number of 3-dupacks. If the ratio is very small (in between 0.01 to 0.2), our observation shows that this event has been caused by a bit error event, not by the congestion. If the ratio is high (e.g. greater than 0.5) then the event is more likely the result of segment drops in intermediate routers due to congestion.

Tables 4.1 and 4.2 show some observed data obtained through our simulation. Details of the simulation setup have been explained later.

**Table 4.1 Effect of bit errors on timeouts and 3-dupacks (with no congestion)**

<b>Error Rate (%) with no congestion</b>	<b>Timeout Count (x)</b>	<b>3-dupack Count (y)</b>	<b>Ratio (x/y)</b>
1	6	127	0.05
5	49	476	0.1
10	195	651	0.3

**Table 4.2 Effect of congestion on timeouts and 3-dupacks (Error Rate = 0.0%)**

Background UDP Traffic Bit-rate (Mbps)	Timeout Count (x)	3-dupack Count (y)	Ratio (x/y)
0.75	61	19	3.21
1	12	4	3
1.25	11	3	3.67

When the possibility of congestion is less but the possibility of bit error is high, we can make TCP less conservative, i.e. no or little reduction in transmission rate. In 3-dupack-event case, the amount of throttling is not that high due to TCP's fast retransmission and fast recovery strategy. However, if we do not reduce the *cwnd* and *ssthresh* that much during a possible bit error event, then TCP can continue with a transmission rate close to the previous rate.

#### 4.2.2 General Observation – II

Network congestion typically occurs whenever internal routers are unable to forward incoming packets and their queues become full. In this case, due to unavailability of the space in the queue, routers have to drop subsequent packets. Multiple segments from the same congestion window of a TCP sender are typically dropped. The smooth flow of forward traffic towards the destination is lost which has an adverse effect on the reverse traffic that is carrying the acknowledgements. Due to insufficient supply of acknowledgement segments back to the sender, the retransmission timer at the TCP sender times out.

After a timeout, TCP enters into the slow-start phase to throttle its transmission rate in order to give the network some relief. Moreover, it doubles its retransmission timeout value. If the network congestion prevails then the timeout will happen successively. In this case, the time difference between two consecutive timeout events will be roughly equal to the timeout interval of the retransmission timer at that instant.

On the other hand, when the network is not congested, the TCP sender will only encounter timeout events whenever the *cwnd* crosses the current network capacity. That will be quickly resolved by the TCP sender by entering into the slow-start phase. Again, if there are random bit errors, then some segments will be damaged and will be rejected by the receiver. However, duplicate acknowledgements will be returned from

the receiver to the source for subsequent segments which are not lost. This will prevent the source from having timeout events and will also enable the source to solve potential loss of segments using fast retransmit and fast recovery. So, in non-congestion scenario the timeouts will be sparse and the time difference between two successive timeouts will be much greater than the retransmission timer's estimated timeout interval at that moment.

This observation can also be used during a timeout or 3-dupack event to decide whether the event is a result of real network congestion or due to random packet losses due to bit error. Let, two consecutive timeout occurs at time  $t_x$  and  $t_{x+1}$ , and the time difference between these two events is  $t_d$  (i.e.  $t_d = t_{x+1} - t_x$ ). Also let  $t_i$  denotes the current estimate of retransmission timer's timeout interval. During a timeout or 3-dupack event, the ratio of  $t_i$  to  $t_d$  will be computed. If the ratio is very small (in between 0.01 to 0.1), our observation shows that this event has been caused by a bit error event, not by the congestion. If the ratio is high (e.g. greater than 0.25) then the event is more likely the result of segment drops in intermediate routers due to congestion.

### 4.3 Basic Structure of the Proposed Algorithm

We propose our modification on top of the existing TCP NewReno algorithm. We call this new modified algorithm **TCP K-Reno**. However, our algorithm can be used with any TCP variants without affecting their usual activities. We have modified only TCP sender and left the receiver as it is. Our proposed algorithm adds some counters and decision blocks to the original TCP NewReno algorithm at the sender. One counter is added to count the number of timeout events and another counter is added to count 3-dupack events. A new variable is used that tracks the time difference between two consecutive timeouts. These values are analyzed when a timeout or 3-dupack event occurs and the sender tries to throttle its transmission rate. The activities of our algorithm can be summarized as follows:

- Based on the values of the newly introduced variables, if our algorithm detects a possible non-congestion event, it will prevent the sender from being too much conservative. For example, in this case the reduction of *cwnd* or *ssthresh* will be less compared to that of original TCP NewReno algorithm.

- However, if the new variables show a high probability of congestion in the network, then our algorithm lets the original TCP NewReno congestion control algorithm to take control and react in the usual way to throttle the transmission rate.

## 4.4 Variables Used in the Algorithm

Table 4.3 lists the variables and parameters used in our proposed algorithm to work along with existing TCP congestion control algorithm.

**Table 4.3 Variables and parameters used in the modified algorithm**

Variable/Parameter Name	Short Description
TC	Count of timeouts
DC	Count of 3-dupacks
TDR	TC : DC
TDR_T	Threshold of TDR
TDR_A	Aging factor of TDR
TI	Holds the current value of the retransmission timer's timeout interval
TD	Time difference between two successive timeouts
ITR	TI : TD
ITR_T	Threshold of ITR
ITR_A	Aging factor of ITR
Latest_Timeout	Holds the time of occurrence of the last timeout event
cwnd	Latest <i>cwnd</i>
ssthresh	Latest <i>ssthresh</i>
NR_cwnd	<i>cwnd</i> set by TCP NewReno
NR_ssthresh	<i>ssthresh</i> set by TCP NewReno
K_cwnd	<i>cwnd</i> set by our algorithm
K_ssthresh	<i>ssthresh</i> set by our algorithm
Counter_Refresh_Cycle	Used to reset TC and DC after a predetermined number of slowdown actions

Table 4.4 shows the typical values of some parameters.

**Table 4.4 Values of some important parameters**

Parameter	Value
TDR_T	0.05 - 0.2
TDR_A	0.125
ITR_T	0.025 - 0.4
ITR_A	0.125
Counter Refresh Cycle	30 - 50

Detailed descriptions of the newly introduced and old variables and parameters that are used in our algorithm are given in the following subsections.

#### 4.4.1 TC

This variable counts the number of times the retransmission timer has expired, i.e. it keeps track of the number of timeouts. Initially the value of this variable is set to zero. Every time the sender experiences a timeout event, we increase this variable by one. After a certain period this variable is reset to zero.

#### 4.4.2 DC

This variable is similar to the previous variable (*TC*) but counts the number of 3-dupack events experienced by the sending TCP entity. Initially the value of this variable is set to one. Every time the sending TCP entity receives the third duplicate acknowledgement, we increase this variable by one. After a certain period this variable is also reset to one. We initialize/reinitialize this variable by one so that our calculation of the ratio between timeout count and 3-dupack count does not suffer from divide-by-zero error.

#### 4.4.3 TDR

This variable holds the ratio between *TC* and *DC*. The value of this variable let us decide whether an indication of segment loss is due to congestion or any other non-congestion related issues.

#### 4.4.4 TDR\_T

This is a pre-specified threshold value against which our calculated *TDR* is compared. The range of values used by *TDR\_T* is given in Table 4.4. Typically *TDR\_T* is set to a value below 0.2. If *TDR* is below this threshold we can conclude that there is no congestion in the network as the number of timeouts is very low. But if *TDR* is above

$TDR\_T$  we will assume it as an indication of possible network congestion. Relying on this information we can modify the TCP congestion control strategy by being less conservative when segments are lost due to non-congestion related events such as random bit errors.

#### 4.4.5 $TDR\_A$

In order to keep a moving average of  $TDR$  values i.e. to incorporate past information with current ratio between  $TC$  and  $DC$ , we use an aging factor  $TDR\_A$ . The value we have used for  $TDR\_A$  is given in Table 4.4. During a timeout or 3-dupack event we update the value of  $TDR$  using the following formula.

$$TDR = (TDR\_A \times TDR) + (1 - TDR\_A) \times (TC / DC) \dots\dots\dots (4.1)$$

#### 4.4.6 $TI$

This variable holds the current value of the retransmission timer's timeout interval. The value of this variable varies according to the round-trip time (RTT) measurement performed by TCP. Also the value of this variable is changed according to TCP specification after every timeout event experienced by the TCP sender.

#### 4.4.7 $TD$

$TD$  represents the time difference between two latest timeout events. Initially  $TD$  is set to  $TI$ . We always keep track of the time when the last timeout event occurred. Whenever the retransmission timer expires at the TCP sender, we set  $TD$  with the time difference between the current time and the last timeout event time.

#### 4.4.8 $ITR$

Every time the TCP sender experiences a timeout or a 3-dupack event, we calculate the ratio of  $TI$  and  $TD$  and keep the value in  $ITR$ . If  $TD$  is much larger than  $TI$  the ratio will have a low value denoting less number of timeouts i.e. timeouts are sparse in nature and are occurring after large delay. This will help us to detect a possible non-congestion related event.

#### 4.4.9 $ITR\_T$

It is a pre-specified threshold value against which the value of  $ITR$  is checked. The range of values used by  $ITR\_T$  is shown in Table 4.4. If  $ITR$  is below  $ITR\_T$  we can



conclude that a timeout or 3-dupack event has occurred due to segment losses from non-congestion related issues. But if *ITR* is higher than *ITR<sub>T</sub>* we can use it as an indication of possible congestion in the network since the high values of *ITR* denotes frequent timeouts.

#### 4.4.10 ITR<sub>A</sub>

In order to keep a moving average of *ITR* values i.e. to incorporate past information with current ratio between *TI* and *TD* we use an aging factor *ITR<sub>A</sub>*. The value we have used for *ITR<sub>A</sub>* is given in Table 4.4. During a timeout or 3-dupack event we update the value of *ITR* using the following formula.

$$ITR = (ITR_A \times ITR) + (1 - ITR_A) \times (TI / TD) \dots\dots\dots (4.2)$$

#### 4.4.11 Latest<sub>Timeout</sub>

This variable records the time of occurrence of the last timeout event. This variable is updated with current time during every timeout event.

#### 4.4.12 cwnd

It is the variable used by the TCP sender to hold the size of its congestion window. The TCP sender transmits according to the minimum of *cwnd* and *rwnd* (the receiver window advertised by the receiving TCP entity). By increasing and decreasing the value of *cwnd* TCP sender controls its rate of transmission.

#### 4.4.13 ssthresh

This variable holds the threshold value that acts as the demarcation point between slow-start and congestion avoidance phases present in TCP congestion control algorithm. Whenever *cwnd* is below *ssthresh*, *cwnd* increases exponentially during every round-trip time. But after crossing *ssthresh*, *cwnd* reduces its rate of increment by incrementing linearly every round-trip time. In fact *ssthresh* denotes TCP's current estimate of available network bandwidth after reaching which TCP should become polite and continue increasing its transmission rate at a slow pace.

#### 4.4.14 NR<sub>cwnd</sub>

This variable holds the new *cwnd* value calculated by TCP NewReno algorithm. In the original algorithm this value is used to set the current *cwnd* after a timeout or 3-

dupack event. But in our modified algorithm, we do not immediately set *cwnd* to *NR\_cwnd*. The details of our modified approach are explained later.

#### 4.4.15 NR\_ssthresh

This variable holds the new *ssthresh* value calculated by TCP NewReno algorithm. In the original algorithm this value is used to set the current *ssthresh* after a timeout or 3-dupack event. But in our modified algorithm, we do not immediately set *ssthresh* to *NR\_ssthresh*. The details of our modified approach are explained later.

#### 4.4.16 K\_cwnd

This variable holds the new *cwnd* value calculated by our modified algorithm, i.e. TCP K-Reno. Based on the current network conditions as indicated by our decision variables (*TDR* and *ITR*) we set the value of *K\_cwnd* to *NR\_cwnd* or to a larger value.

#### 4.4.17 K\_ssthresh

This variable holds the new *ssthresh* value calculated by our modified algorithm, i.e. TCP K-Reno. Based on the current network conditions as indicated by our decision variables (*TDR* and *ITR*) we set the value of *K\_ssthresh* to *NR\_ssthresh* or to a larger value.

#### 4.4.18 Counter\_Refresh\_Cycle

After experiencing certain number of timeout and 3-dupack events we reset our counters (*TC* and *DC*) to their initial values (0 and 1 respectively). This is done to ensure that the data from the distant past cannot affect the current or the future decision making. If there is no congestion for a long time then the count of timeouts will be very low. However, due to reordering of segments and bit error in wireless networks, the count of 3-dupacks will have a moderately high value compared to timeout count. In this situation, if the network suddenly experiences heavy congestion then there will be some timeouts. However, the timeout count value will still be low, and will take a significant amount of time to reach close to 3-dupack count value. As a result the decision variable will continue to report that there is no congestion and the sender will remain aggressive. This will surely congest the network more. To remove this problem, we have opted the strategy to reset the counter values after certain

number of slowdown requests. Typical values of *Counter\_Refresh\_Cycle* are given in Table 4.4.

## 4.5 Actions Performed by K-Reno

Here we present the TCP K-Reno algorithm using pseudo code. We have divided the entire algorithm into four procedures. Each procedure is called to take appropriate action when a particular event occurs. These procedures are also part of the original TCP NewReno [7] specification and have been modified by us. They are:

1. **RECEIVE** – called when the TCP sender receives an acknowledgement segment.
2. **DUPACK\_ACTION** – called whenever the TCP sender experiences the third consecutive duplicate acknowledgement.
3. **TIMEOUT\_ACTION** – called whenever the retransmission timer at the TCP sender times out.
4. **SLOWDOWN\_ACTION** – called by “**DUPACK\_ACTION**” and “**TIMEOUT\_ACTION**” to throttle the transmission rate by reducing the values of *cwnd* and *ssthresh*.

Details of these procedures are given below. The following enumeration is used in these procedures to denote the type of event experienced by the TCP sender.

**ENUMERATION EVENT\_TYPE {DUPACK = 0, TIMEOUT = 1}**

### 4.5.1 Operations of Procedure “RECEIVE”

At TCP source procedure “RECEIVE” is invoked by the network layer whenever the source receives an acknowledgement segment from the TCP receiver. In this procedure the TCP source performs different actions based on the type of acknowledgement received. In fact, we have kept the original implementation of “RECEIVE” specified in TCP NewReno [7] intact. The pseudo code of this procedure is given in Procedure 4.1.

#### Procedure 4.1 RECEIVE

1. BEGIN
2. IF new acknowledgment is received THEN
  - 2.1. IF inside fast recovery phase THEN
    - 2.1.1. Deflate cwnd by setting  $cwnd = ssthresh$
    - 2.1.2. Exit from fast recovery
  - 2.2. ELSE
    - 2.2.1. Increase cwnd according to current phase (slow-start or congestion avoidance)
  - 2.3. END IF
3. ELSE IF partial acknowledgement is received THEN
  - 3.1. Remain inside fast recovery and retransmit the segment expected by the receiver
4. ELSE IF duplicate acknowledgement is received THEN
  - 4.1. IF inside fast recovery phase THEN
    - 4.1.1. Increase cwnd by 1 MSS
  - 4.2. ELSE
    - 4.2.1. IF 3-dupack THEN
      - 4.2.1.1. CALL DUPACK\_ACTION
    - 4.2.2. END IF
  - 4.3. END IF
5. END IF
6. END

#### 4.5.2 Operations of Procedure “DUPACK\_ACTION”

The procedure “DUPACK\_ACTION” is invoked whenever the TCP sender gets three consecutive duplicate acknowledgements (3-dupacks). In this procedure we are updating  $TDR$  and  $ITR$  using equations (4.1) and (4.2). We are keeping all the TCP NewReno actions in our “DUPACK\_ACTION” procedure. It invokes the procedure

“SLOWDOWN\_ACTION” and passes the *DUPACK* event as the parameter. The pseudo code of this procedure is given in Procedure 4.2.

#### Procedure 4.2 *DUPACK\_ACTION*

1. *BEGIN*
2.  $SET DC = DC + 1$
3.  $SET TDR = (TDR\_A \times TDR) + (1 - TDR\_A) \times (TC / DC)$
4.  $SET TD = Current\_Time - Latest\_Timeout$
5.  $SET ITR = (ITR\_A \times ITR) + (1 - ITR\_A) \times (TI / TD)$
6. *Reset the retransmission timer*
7. *Retransmit the presumed lost segment using fast retransmission*
8.  $CALL SLOWDOWN\_ACTION (DUPACK)$
9. *END*

#### 4.5.3 Operations of Procedure “TIMEOUT\_ACTION”

This procedure is invoked whenever the retransmission timer expires at the TCP sender. Like “*DUPACK\_ACTION*”, here, we are not changing the usual operations of TCP NewReno. We are updating the values of *TDR* and *ITR* using equations (4.1) and (4.2) as well. Finally, it invokes the procedure “*SLOWDOWN\_ACTION*” passing the *TIMEOUT* event as the parameter. One thing in this procedure needs special mention. Here we are not only incrementing the value of *TC* by 1 but also setting the value of *DC* to one-fourth of its current value. This is performed in order to make the algorithm being capable of detecting sudden congestion in the network after a long period of non-congestion state. If the network experiences sudden congestion after a long non-congestion period, there will be several timeouts, however, the *TC* value will still remain too low compared to *DC* value and K-Reno will mistakenly consider these timeouts due to bit errors. To solve this problem, we decrease the *DC* value after every timeout so that the *DC* value does not remain too high compared to the *TC* value. The pseudo code of this procedure is given in Procedure 4.3.

### Procedure 4.3 TIMEOUT\_ACTION

1. *BEGIN*
2. *SET TC = TC + 1*
3. *SET TDR = (TDR\_A × TDR) + (1 - TDR\_A) × (TC / DC)*
4. *SET DC = MAX(DC / 4, 1)*
5. *SET Previous\_Timeout = Latest\_Timeout*
6. *SET Latest\_Timeout = Current\_Time*
7. *SET TD = Latest\_Timeout - Previous\_Timeout*
8. *SET ITR = (ITR\_A × ITR) + (1 - ITR\_A) × (TI / TD)*
9. *Reset the retransmission timer*
10. *CALL SLOWDOWN\_ACTION (TIMEOUT)*
11. *Retransmit the segment*
12. *END*

#### 4.5.4 Operations of Procedure “SLOWDOWN\_ACTION”

The “SLOWDOWN\_ACTION” procedure is the place where the actual throttling actions of the TCP sender take place. It is invoked by both “DUPACK\_ACTION” and “TIMEOUT\_ACTION” procedures. If it is called by “DUPACK\_ACTION”, it retransmits the reported missing segment, updates *NR\_ssthresh* and *NR\_cwnd* and enters into the fast recovery phase. If it is called by “TIMEOUT\_ACTION”, it updates *NR\_ssthresh* and *NR\_cwnd* only. It enters neither into fast retransmission nor into fast recovery. *TDR* value is compared against a pre-specified threshold (*TDR\_T*) and *ITR* value is compared against a pre-specified threshold (*ITR\_T*) irrespective of the fact which procedure has called it. Based on these comparisons this procedure predicts whether the cause of the slowdown request is real congestion in the network or not. If non-congestion cause is predicted we do not reduce the *cwnd* and *ssthresh* that much. The pseudo code of this procedure is given in Procedure 4.4.

**Procedure 4.4 SLOWDOWN ACTION (EVENT\_TYPE: Event)**

1. BEGIN
2. IF Event = DUPACK THEN
  - 2.1. Enter into fast retransmission
  - 2.2. Enter into fast recovery
  - 2.3. SET NR\_ssthresh = cwnd / 2
  - 2.4. SET NR\_cwnd = NR\_ssthresh + 3
3. ELSE IF Event = TIMEOUT THEN
  - 3.1. SET NR\_ssthresh = cwnd / 2
  - 3.2. SET NR\_cwnd = 1
4. END IF
5. IF TDR < TDR\_T AND ITR < ITR\_T THEN
  - 5.1. SET K\_ssthresh =  $\frac{3}{4}$  cwnd
  - 5.2. SET K\_cwnd =  $\frac{1}{2}$  cwnd
6. ELSE IF TDR < TDR\_T OR ITR < ITR\_T THEN
  - 6.1. SET K\_ssthresh =  $\frac{2}{3}$  cwnd
  - 6.2. K\_cwnd = NR\_cwnd
7. ELSE
  - 7.1. K\_ssthresh = NR\_ssthresh
  - 7.2. K\_cwnd = NR\_cwnd
8. END IF
9. SET ssthresh = MAX(K\_ssthresh, NR\_ssthresh)
10. SET cwnd = MAX(K\_cwnd, NR\_cwnd)
11. END

There are three cases to consider in the “SLOWDOWN\_ACTION” procedure.

- If both  $TDR$  and  $ITR$  values are below their respective thresholds, we can conclude that the slowdown event has occurred because of a segment being lost due to bit error or any other non-congestion related event. In order to throttle the transmission rate less we set the variable  $K_{ssthresh}$  to three-fourth of the current  $cwnd$  and the variable  $K_{cwnd}$  to half of the current  $cwnd$ . Setting the  $K_{cwnd}$  to the half of the current  $cwnd$  will improve the performance of TCP in case of false alarms. Moreover setting  $K_{ssthresh}$  to three-fourth of the current  $cwnd$  value will widen the interval in which  $cwnd$  will be able to grow exponentially. This will let the sending TCP entity to have a good throughput and improve performance in case of non-congestion related timeouts or 3-dupacks.
- If either  $TDR$  or  $ITR$  value is below the respective threshold and the other is not, there is a possibility of real congestion in the network. However, the congestion is not that much severe since one of the decision variables is below the threshold. In order to be moderate aggressive we set  $K_{ssthresh}$  to two-thirds of the current  $cwnd$  and  $K_{cwnd}$  to  $NR_{cwnd}$ .
- Lastly, if both  $TDR$  and  $ITR$  values are above the respective thresholds, we conclude that the network is experiencing real congestion and we set  $K_{ssthresh}$  to  $NR_{ssthresh}$  and  $K_{cwnd}$  to  $NR_{cwnd}$  like original TCP NewReno.

Finally we take the maximum of  $K_{ssthresh}$  and  $NR_{ssthresh}$  and use this value to set the current  $ssthresh$  value. Similarly we take the maximum of  $K_{cwnd}$  and  $NR_{cwnd}$  and use this value to set the current  $cwnd$  value.

## 4.6 Aggression and Fairness Issues

While developing our algorithm we have tried to keep it friendly with other concurrent TCP connections. Our proposed algorithm ensures a good throughput in case of non-congestion related losses, however, it does not become aggressive during real congestion. In order to guarantee that this good behavior is shown in our algorithm, we have carefully chosen the values for  $TDR_T$  and  $ITR_T$ . If they were not properly chosen then our algorithm might have become too much aggressive and



caused serious performance penalty for itself and other concurrent TCP connections. Our experiment shows that our algorithm sometime becomes aggressive if we use higher values for  $TDR\_T$  (e.g.  $\geq 0.25$ ) and  $ITR\_T$  (e.g.  $\geq 0.5$ ). In this case, our algorithm will consider some timeout and 3-dupack events as being caused by non-congestion related matters even if there is real congestion in the network and will not throttle its transmission rate. This action will add more burden on the already congested network and will degrade the performance of all the TCP connections, sharing the same bottleneck link. Tables 4.5 and 4.6 show the simulation results of a single TCP connection with different  $TDR\_T$  and  $ITR\_T$  values in a wired-cum-wireless network. Here, the number of unique segments transmitted is used as the performance indicator.

**Table 4.5 Effect of possible aggression of TCP K-Reno (Error Rate = 2.5%,  $ITR\_T = 0.045$ )**

$TDR\_T$	NewReno ( $N$ )	K-Reno ( $K$ )	$K - N$
0.025	15207	15239	32
0.05		15240	33
0.075		15275	68
0.1		15223	16
0.2		15208	1
0.25		15142	-65

**Table 4.6 Effect of possible aggression of TCP K-Reno (Error Rate = 2.5%,  $TDR\_T = 0.075$ )**

$ITR\_T$	NewReno ( $N$ )	K-Reno ( $K$ )	$K - N$
0.01	15207	15254	47
0.025		15264	57
0.045		15275	68
0.075		15224	17
0.1		15209	2
0.2		15243	36
0.25		15218	11
0.4		15221	14
0.5		15209	2
0.6		15207	0

From Tables 4.5 and 4.6 it is clearly evident that TCP K-Reno performs better if  $TDR\_T$  value is set in the range 0.05 - 0.2 and  $ITR\_T$  value is set in the range 0.025 -

0.4. For this reason, we have used aforementioned ranges in our algorithm to ensure that we do not fall into the trap of false non-congestion alarm.

Above discussion concludes our congestion control algorithm, TCP K-Reno. In the next chapter we present the performance analysis of our algorithm.

# 5 Performance Evaluation of TCP

## K-Reno

As mentioned earlier, we have used ns-2 as the simulation platform to test the performance of our proposed congestion control algorithm and to compare the same with other major TCP variants. The following subsections will explain the changes made into ns-2 code, simulation setup, and the results obtained from different simulation runs in detail.

### 5.1 Modifications Performed in ns-2 Code

In order to test our proposed algorithm, we have used ns-2 version 2.31 as our simulation platform. ns-2 contains implementation of TCP NewReno and some other variants of TCP. The following files in ns-2 contain source codes of our concern.

- tcp.h
- tcp.cc
- tcp-newreno.cc
- tcp-sink.h
- tcp-sink.cc

Although we have not modified the receiving side TCP algorithm, we have added some code on the implementation of class *TcpSink* in ns-2 to simulate errors in the reverse link.

In the TCP NewReno code present in ns-2, three functions typically deal with the timeout and 3-dupack events. They are –

- void dupack\_action();
- void timeout(int tno);
- void slowdown(int how);

We have added our newly introduced variables in the files *tcp.h* and *tcp.cc* as these files contain the implementation of the class *TcpAgent*. *TcpAgent* class contains code for all the basic activities specified in TCP. It acts as the base class from which class *NewRenoTcpAgent* has been derived that contains the implementation of TCP NewReno. *NewRenoTcpAgent* inherits “slowdown()” function from class *TcpAgent* and overrides the “dupack\_action()” and “timeout()” functions. We have added our code in the aforementioned three functions of class *TcpAgent* and class *NewRenoTcpAgent* present in files *tcp.cc* and *tcp-newreno.cc* respectively.

## 5.2 Simulation Setup

We have used a mixed network in order to evaluate the performance of our proposed algorithm. Figure 5.1 shows the network topology used in the simulations.

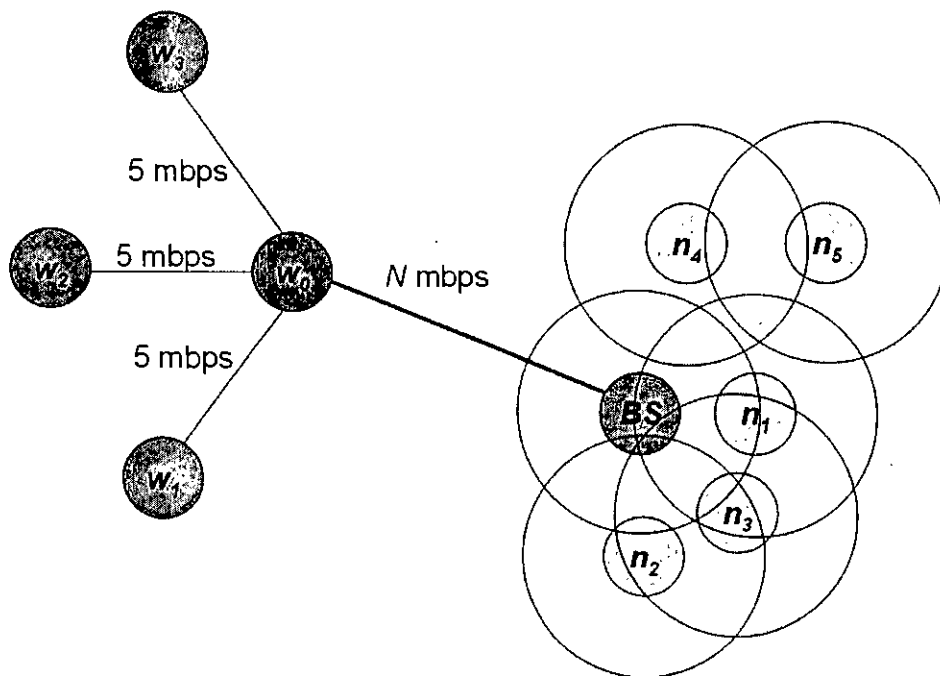


Figure 5.1 Simulation setup of wired-cum-wireless network

In Figure 5.1, all the nodes starting with 'w' represent nodes in a wired network and all the nodes starting with 'n' represent wireless nodes. The node labeled "BS" acts as a gateway between the wired and wireless part of the network. BS is connected to  $w_0$  using a link having bandwidth of  $N$  mbps where  $N$  can be 7 or 12. This link acts as the bottleneck link in our simulation. All other nodes in the wired domain are connected to  $w_0$  using individual link of 5 mbps bandwidth. We have generated the following traffics in different simulations.

- A TCP connection between  $w_1$  (sender) and  $n_1$  (receiver)
- A TCP connection between  $w_2$  (sender) and  $n_2$  (receiver)
- A UDP connection between  $w_3$  (sender) and  $n_3$  (receiver)
- A UDP connection between  $n_4$  (sender) and  $n_5$  (receiver)

We have run the simulation to test the performance of TCP K-Reno for both single TCP connection and two simultaneous TCP connections. UDP connections generating constant bit rate traffics are used to create congestion in the network. In order to evaluate the performance of different TCP implementations we have used the number of unique segments transmitted by the sender as the comparison parameter. This number represents the throughput of a connection and if it is larger in one TCP implementation than that is in another TCP implementation, the former implementation denotes the superiority over the later implementation.

We have run the simulation for TCP Tahoe, TCP Reno, TCP NewReno, TCP Westwood and TCP K-Reno. All the runs have been continued for 250 seconds. When a single TCP connection is concerned, we have only used the TCP connection between  $w_1$  and  $n_1$ , removing the TCP connection between  $w_2$  and  $n_2$ . We have used this scenario to analyze how TCP K-Reno behaves when it does not have to compete with other concurrent TCP connections. During this test we have set the bottleneck link bandwidth to 7 mbps. In order to analyze the performance of TCP K-Reno in a multi-connection scenario we have run simulations using TCP connections between  $w_1$  and  $n_1$  and between  $w_2$  and  $n_2$ . The focus of this scenario was to evaluate how TCP K-Reno behaves when it has to co-exist with other similar TCP connections. In this case, the bottleneck link bandwidth is set to 12 mbps. In both cases, we have kept the bottleneck bandwidth slightly higher than that is required by the TCP connection(s) so

that we can introduce different levels of congestion by customizing the data rate of background UDP traffics. We have run our simulations by keeping the background UDP traffics both on and off.

### 5.3 Simulating Bit Error in the Wireless Channel

We have used IEEE 802.11 as the wireless medium access protocol in ns-2. IEEE 802.11 uses virtual channel sensing to avoid collision and to detect ongoing transmission. This is called the MACAW (Multiple Access with Collision Avoidance for Wireless). In this strategy, every wireless node first senses the radio channel to see whether the medium is free. If the channel is free then the sender transmits an RTS (Request to Send) frame to the destination. If the destination is ready to receive data, it replies with a CTS (Clear to Send) frame. These RTS and CTS frames also inform nearby stations (within the radio range of the sender and the potential receiver) about the imminent data transfer and hence they remain quiet during the entire data transmission period.

Currently ns-2 does not support any error modules for the wireless links although error models for wired networks are fully supported. In order to introduce random bit errors in the wireless channel we have incorporated two strategies. Firstly, we have placed two wireless nodes  $n_4$  and  $n_5$  beyond the reach of the  $BS$  and the TCP sinks ( $n_1$  and  $n_2$ ). We have set up a UDP traffic generator from  $n_4$  to  $n_5$ . As  $n_4$  and  $n_5$  are beyond the reach of the radio coverage of  $BS$ ,  $n_1$  and  $n_2$ , they do not hear the RTS and CTS messages exchanged by  $BS$  and  $n_1$  (or  $n_2$ ). So,  $n_4$  and  $n_5$  continues to transmit data even when  $BS$  and  $n_1$  (or  $n_2$ ) are trying to exchange TCP segments. Activities of  $n_4$  and  $n_5$  act as interference for the communication between  $BS$  and  $n_1$  (or  $n_2$ ) and add certain degree of error.

Moreover, to control the error rate more precisely, we have modified the code of class *TcpSink* in ns-2 a little bit. In the file *tcp-sink.cc* (which contains the implementation code of class *TcpSink*) we have added some variables and some decision blocks. One of the newly introduced variables holds the error rate that we want the TCP connection to experience. This error rate is configurable from the TCL (Toolkit Command Language) script. TCL script is used to describe the simulation scenario. We have also added a uniform random number generator that generates a value

between 0 and 1 whenever it is accessed. In the TCP sink, whenever we receive a segment from the sender, we generate a new random number and check this value against the error rate set at the TCP sink. If the generated value falls below the error rate (which can be 0.01 for 1% error, 0.025 for 2.5% error etc.) we simply drop the segment to simulate a corrupted segment. Here  $x\%$  error rate means  $x$  segments out of 100 segments will suffer error. This action causes the TCP sink to generate duplicate acknowledgements on the receipt of subsequent undamaged segments. By customizing the error rate we can control how frequently segments are dropped. This approach deals with introducing the error only into the forward channel. In order to incorporate error in the reverse channel we have also introduced another uniform random number generator. But the error rate for the reverse channel is set  $1/25$  th of the forward channel error rate. This has been done to address the relative sizes of forward TCP segment and reverse TCP segment. In ns-2 TCP segments sent from the sender to the receiver have default size of 1000 bytes. But the acknowledgement segments sent from the receiver to the sender are only 40 bytes. Due to their small size, we have assumed that acknowledgement segments will suffer less error compared to that of data segments. We have modified the portion of TCP sink where it sends the acknowledgement to the sender. We generate a new random value from the second random number generator and compare the value with the error rate set for the reverse channel. If the value falls below this error rate, we simply drop the acknowledgement segment. Otherwise, we supply the acknowledgement segment to the lower layer for sending it to the TCP sender.

## 5.4 Simulation Results and Analysis

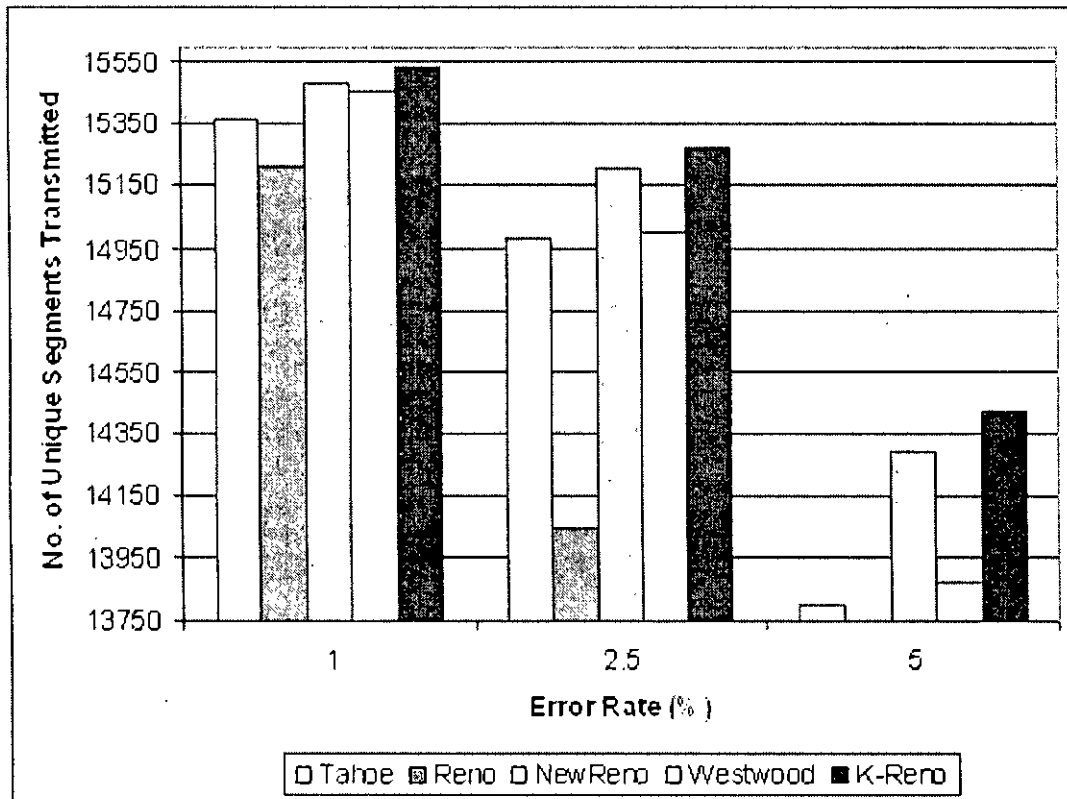
Table 5.1 shows simulation results obtained after running a single TCP connection using different TCP variants and TCP K-Reno.

**Table 5.1 Performance comparison of single TCP connection**

Error Rate (%)	Tahoe	Reno	NewReno ( $N$ )	Westwood ( $W$ )	K-Reno ( $K$ )	$K-N$	$K-W$
1	15365	15214	15481	15449	15531	50	82
2.5	14983	14041	15207	15002	15275	68	273
5	13799	12391	14290	13870	14418	128	548

In the above simulation runs all the UDP traffics were active. From the simulation results it is clearly evident that K-Reno outperforms all other TCP variants, even TCP Westwood, which was specially designed for wireless networks. Reasons behind TCP Westwood's poor performance will be described in a subsequent section.

Figure 5.2 shows the information presented in Table 5.1 graphically.



**Figure 5.2 Performance comparison of single TCP connection**

The performance improvement of K-Reno can be attributed to its less conservative reaction during segment losses due to random bit errors. Whenever K-Reno detects a possible non-congestion event it does not reduce its transmission rate too much. So it continues transmitting at a good rate and can deliver more segments in the midst of wireless bit errors. But as other TCP variants (Tahoe, Reno and NewReno) drastically reduces the congestion window whenever a segment loss is detected they fail to achieve a good throughput. In case of NewReno, fast retransmission and fast recovery are capable of ensuring a good throughput when multiple segments are dropped from the same window. However, if segment drops are sporadic in nature, consecutive reception of 3-dupacks will continue the halving of the congestion window even



though the segments are dropped due to bit error. K-Reno detects segment losses due to bit error with high precision and keeps a steady flow of segments towards the destination to ensure a good throughput. Again in real congestion, K-Reno does not behave aggressively and hence do not worsen the congestion in the network. This behavior is very significant where two concurrent TCP connections are used.

Table 5.2 shows the effect of UDP traffics on the performance of single TCP connection using different TCP variants.

**Table 5.2 Effect of UDP traffics on single TCP connection**

Error Rate (%)	UDP Traffic	NewReno ( <i>N</i> )	Westwood ( <i>W</i> )	K-Reno ( <i>K</i> )	<i>K-N</i>	<i>K-W</i>
0	None	18971	18971	18971	0	0
	2	15719	15719	15719	0	0
1	None	18621	18503	18650	29	147
	2	15481	15449	15531	50	82
2.5	None	18299	17814	18332	33	518
	2	15207	15002	15275	68	273
5	None	16815	16102	17022	207	920
	2	14290	13870	14418	128	548

From the data presented in Table 5.2, we can see that irrespective of the presence or absence of UDP traffic, K-Reno performs better than both TCP NewReno and TCP Westwood. Improvement in throughput of K-Reno compared to TCP Westwood is better in the absence of UDP traffic than that is in the presence of the same. Table 5.2 also shows that in the presence of random bit error TCP Westwood performs badly compared to TCP NewReno.

Figure 5.3 illustrates graphically the information presented in Table 5.2 (showing only the case where UDP traffics were not present).

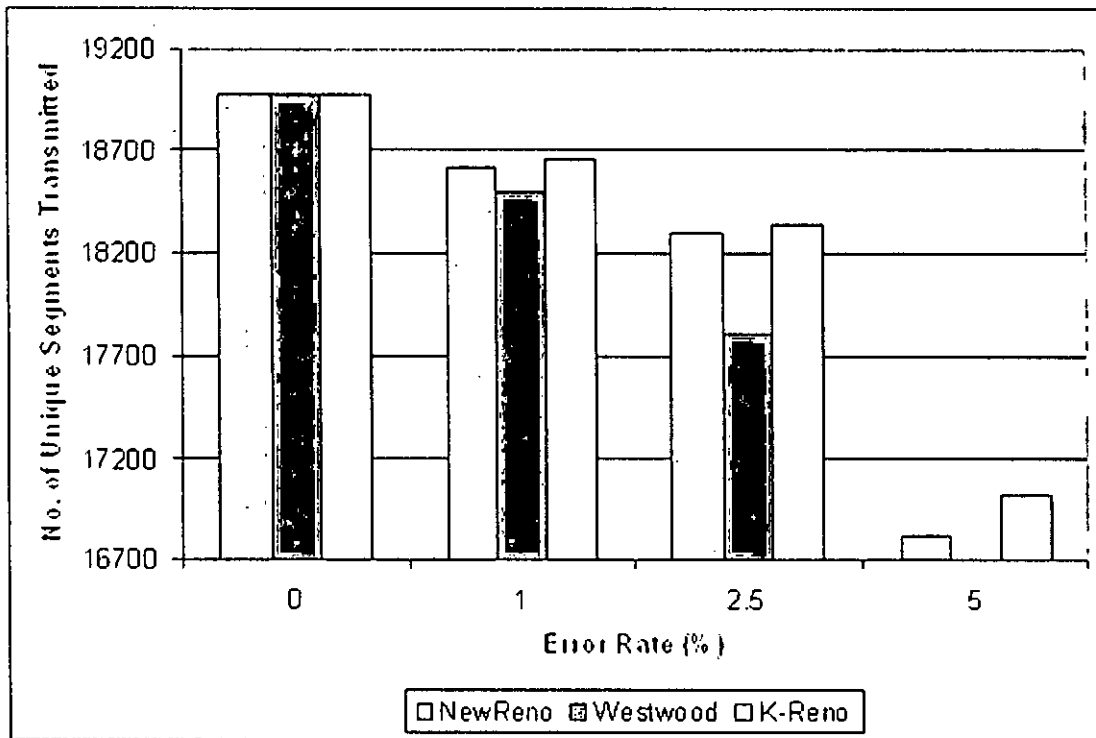


Figure 5.3 Performance of single TCP connection (with no UDP traffic)

Table 5.3 shows the simulation results obtained from running two concurrent and homogeneous TCP connections. Here, the average number of unique segments sent by the two connections has been recorded. All the UDP traffics were present during these simulation runs.

Table 5.3 Performance comparison with two TCP connections

Error Rate (%)	NewReno ( <i>N</i> )	Westwood ( <i>W</i> )	K-Reno ( <i>K</i> )	<i>K</i> - <i>N</i>	<i>K</i> - <i>W</i>
1	7817	7820	7823	6	3
2.5	7702	7704	7712	10	8
5	7475	7479	7497	22	18

From the above data it is clearly evident that K-Reno is able to inject more unique segments into the network than both TCP NewReno and TCP Westwood. These observations confirm that TCP K-Reno does not affect the operation of concurrent TCP connections. If TCP K-Reno were too much aggressive then it would have adversely affected the other TCP connections, who are sharing the same bottleneck link. If a TCP connection mistakenly remains aggressive during network overload time, a large number of segments will be dropped at the congested node. These

dropped segments will consist of segments from the aggressive connection and also segments from other moderate TCP connections. So all the moderate connections will experience more timeout events and hence will throttle their transmission rate. Moreover, the aggressive connection will also experience more timeout events that will drastically reduce its rate of transmission. So in the long run the average throughput of the overall network will be low. This situation will continue each time a TCP connection shows aggressive behavior during real network congestion. TCP K-Reno does not reduce *cwnd* and/or *ssthresh* too much until it is ensured that the timeout or 3-dupack event has occurred due to a congestion related event. So the actions TCP K-Reno takes during false alarm of network congestion do not produce any burden on the network and concurrent TCP connections. All the simulation results presented above have also shown that the gain of our TCP K-Reno increases with the increase in the error rate in the network. This is desired when a TCP algorithm is designed to overcome the bit error problem.

In the following section we present a detailed comparative analysis of TCP K-Reno's performance with that of TCP Westwood as the later has been specially designed for wireless networks.

## 5.5 Extensive Performance Comparison with TCP

### Westwood

In Tables 5.1 and 5.2 we saw that TCP Westwood performs badly compared to both TCP NewReno and TCP K-Reno when we incorporate bit errors into the wireless channel though it was designed specially for wireless networks to improve TCP's performance in the presence of random bit errors.

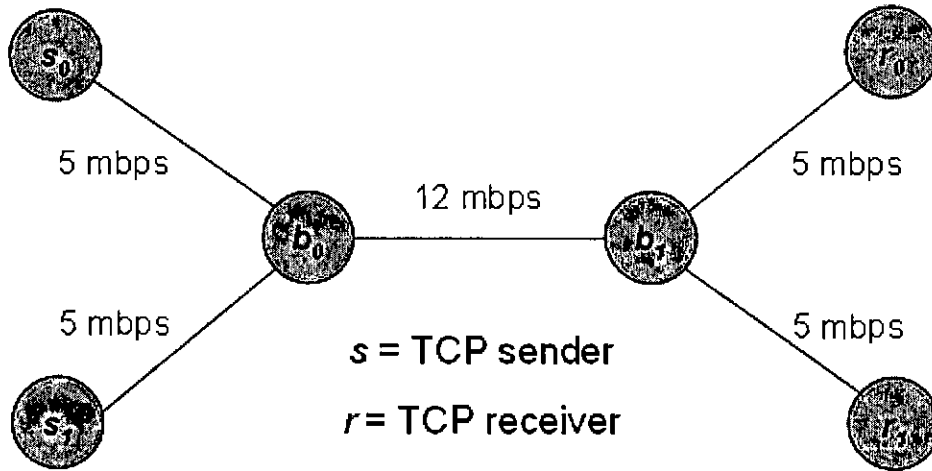
TCP Westwood relies on consistent supply of acknowledgement segments from the receiver to estimate the available bandwidth of the network. TCP Westwood's performance depends highly on the precision of the above estimation. If TCP Westwood fails to estimate the available bandwidth at any instant then it will suffer seriously. In case of a lower estimated value than the actual available bandwidth, TCP Westwood will not be able to utilize network resource properly. In case of a higher estimated value than the actual available bandwidth, TCP Westwood will act aggressively. It will introduce congestion in the network and segment drops will occur

at the bottleneck nodes. Eventually the TCP sender will experience multiple timeout events and will progressively reduce its transmission rate to a much lower value.

So, whenever the acknowledgment stream is disrupted in a TCP Westwood connection, it will show poor performance. In our simulated environment the acknowledgement stream of any TCP connection is disrupted in two ways. Firstly, an error introduced in the wireless channel drops some acknowledgement segments. Secondly, the acknowledgement stream from a wireless receiver towards a wired host (i.e. the original sender) is disrupted whenever other nodes in the wireless network are communicating. In our simulation we have used IEEE 802.11 as the wireless access protocol. This protocol allows multiple nodes to use the same wireless medium by employing MACAW (Multiple Access with Collision Avoidance for Wireless). In this technique, only one wireless station is permitted to transmit data while other nearby wireless nodes must refrain from doing so. The nodes that want to communicate uses RTS and CTS messages to ensure that they access the channel without disrupting the other ongoing communications. The RTS and CTS messages also inform other nodes about the possible duration of the transmission. During that period, which is called Network Allocation Vector (NAV), other nodes are not allowed to transmit any packet into the network. This is not the case in wired networks. In wired networks, a node can transmit a packet any time if the carrier is free, i.e., there is no NAV period. For this reason, when a TCP Westwood connection operates in a wired network it shows good performance even if we introduce some error in the wired channel. Table 5.4 shows the performance comparison of TCP Westwood, TCP NewReno and TCP K-Reno in a simulated wired network using ns-2. Figure 5.4 shows the network that was used to perform this comparison. In each simulation run, two TCP connections were running concurrently. One of them is between  $s_0$  and  $r_0$  and the other is between  $s_1$  and  $r_1$ . We have run separate simulations using TCP NewReno, TCP Westwood and TCP K-Reno for both the connections and the average number of unique segments transmitted by both connections is used as the performance indicator. We have also used the built-in error module of ns-2 to incorporate error in the wired link between  $b_0$  and  $b_1$ .

**Table 5.4 Performance of TCP variants in Wired Network (with error)**

Error Rate (%)	NewReno (N)	Westwood (W)	K-Reno (K)	$W = N$	$K = N$
1	20258	30284	24937	10026	4679
2.5	11924	21974	14801	10050	2877
5	7286	13862	7674	6576	388



**Figure 5.4 Simulation setup for testing performance of TCP variants in a wired network**

As in wired networks acknowledgements from the TCP receivers are traveling the link between  $b_1$  and  $b_0$  in a multiplexed fashion, TCP Westwood is able to predict accurately the available network bandwidth and set its *cwnd* and *ssthresh* accordingly. This ensures a better throughput than both TCP NewReno and TCP K-Reno, and hence a higher unique segment count is achieved as shown in Table 5.4.

Now let us go back to our original simulation setup shown in Figure 5.1. In this setup the base station (*BS*) is communicating with  $n_1$ ,  $n_2$  and  $n_3$  to forward packets received from  $w_1$ ,  $w_2$  and  $w_3$  respectively. In return *BS* also receives acknowledgement segments from both  $n_1$  and  $n_2$  to inject those into the wired network. No acknowledgement is received from  $n_3$  because of a UDP connection between  $w_3$  and  $n_3$ . But whenever *BS* is communicating with  $n_1$  or  $n_3$ ,  $n_2$  has to remain quiet for a certain period. Similar case occurs for  $n_1$  when the wireless channel is occupied for the communication between *BS* and  $n_2$  or between *BS* and  $n_3$ . This situation is exacerbated when random bit errors are introduced in the wireless channel. In the presence of bit errors, the wireless nodes will fail to successfully transmit

acknowledgement segments to their respective peers. So the multi-access nature of a wireless network and the presence of random bit errors refrain TCP Westwood from having an accurate estimate of available network bandwidth. That is why in Tables 5.1 and 5.2 TCP Westwood shows poor performance compared to both TCP NewReno and TCP K-Reno in our simulated wired-cum-wireless environment though TCP Westwood is proven to show better performance than other TCP variants in single access wireless channels (such as dedicated wireless link between a VSAT and a satellite).

This brings us to the end of our performance analysis. The following chapter summarizes our works and provides some pointers for future research works targeting TCP congestion control in mixed networks.

## 6 Conclusion

TCP is a part and parcel of current communication infrastructure. The services offered by TCP are suitable for different types of applications. But due to the advent of heterogeneous network systems in recent years, the performance of TCP in these new environments has been brought under question. It is not desirable to propose a completely new transport protocol for the new environments. So the focus of our current studies on TCP is to modify certain parts of TCP congestion control strategy and some other aspects so that TCP can react more appropriately based on the present condition of the network. New and improved congestion control strategy will certainly permit TCP to become more useful in different network situations. In this thesis, we have proposed a new congestion control algorithm that we call **TCP K-Reno** which can be incorporated with any existing TCP variant and is capable of performing well in heterogeneous networks (e.g. wired-eum-wireless network). TCP K-Reno is end-to-end in nature and modifies only the sender-side TCP implementation. It keeps the TCP receiver and the network unaware of the modifications. This feature makes TCP K-Reno suitable for deploying in real life scenario and does not impose any burden on the internal network.

Our proposed TCP congestion control algorithm, i.e. TCP K-Reno has the following advantages.

- It is an end-to-end proposal.
- Only sender side TCP needs to be changed.
- Assistance from routers is not required.
- It does not impose too much processing overload in the TCP/IP protocol stack.

- Can be incorporated with any TCP variant.
- Proposed parameters can be tuned considering network condition of different environments to achieve better performance.
- Performs better than all TCP variants, including NewReno and Westwood, in multi-access wireless networks (e.g. wireless ad hoc networks).

TCP K-Reno is capable of distinguishing segment losses due to both congestion and non-congestion related issues. With the help of some newly introduced variables and decision blocks, it is able to determine whether segments are getting dropped in congested routers or are being damaged due to random bit errors. In case of real congestion, it simply behaves as original TCP NewReno algorithm. But after detecting a probable non-congestion event, unlike TCP NewReno, it does not throttle its transmission rate too much. It continues to transmit at a good pace so that the network capacity does not remain unutilized at the presence of random bit errors. We have compared our proposed algorithm with other major TCP variants (TCP Tahoe, Reno, NewReno and Westwood) using ns-2 and have found that TCP K-Reno performs better than any of them. In fact, the simulation results have shown that TCP K-Reno outperforms TCP Westwood, though it has been specially designed for wireless networks, by a big margin.

Currently, we are using some empirically derived values for different parameters used in the modified algorithm. For example, the values of the thresholds  $TDR_T$  and  $ITR_T$  have been determined using thorough analysis of numerous simulation runs. But a single value of  $TDR_T$  or  $ITR_T$  might not work well in all types of networks. Different network condition will demand different values of those thresholds for consistent throughput. So, some type of dynamism in the values of  $TDR_T$  and  $ITR_R$  may need to be introduced to make our algorithm more robust in changing network environments. Moreover, by considering more levels of  $TDR$  and  $ITR$  values we can perform the fine adjustment of  $cwnd$  and  $ssthresh$  to ensure optimum throughput of a TCP connection.

Fairness among concurrent TCP connections sharing the same bottleneck link is another issue of dominant concern. The goal of TCP fairness is to ensure a fair share of the available bandwidth in a shared link for all the TCP connections. Currently TCP K-Reno does not deal with ensuring fairness among multiple TCP connections.



Further research works are needed to shape TCP K-Reno so that it ensures fair share with all other TCP variants.

Changes in the estimated round-trip time (RTT) of a TCP connection sheds some light on the current network load. By observing the change pattern of RTT, a TCP source can deduce the optimum level of throughput that will enable the source to utilize the available bandwidth successfully without overburdening the network. So some type of record keeping of previous RTT values and decisions based on those records can be incorporated in the congestion control algorithm to improve TCP's performance in mixed networks. We will incorporate the change pattern of RTT in congestion control in our future work.

Complexity analysis of an algorithm is important to have a clear idea of its best, average and worst case execution time. Such analysis is not generally available for TCP congestion control algorithms, including our K-Reno algorithm. We have a plan to look into the complexity analysis of TCP K-Reno and other congestion control algorithms to compare them in our future research.

## References

- 205880
- [1] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
  - [2] Jacobson, V., "Congestion Avoidance and Control", Proceedings of SIGCOMM '88, ACM, pp. 314-329, Stanford, CA, August 1988.
  - [3] Allman, M., Paxson, V. and Stevens, W., "TCP Congestion Control", RFC 2581, April 1999.
  - [4] Hoe, J. C., "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", Proceedings of ACM SIGCOMM '96, pp. 270-280, Stanford, CA, USA, August 1996.
  - [5] "The Network Simulator – ns-2", <http://www.isi.edu/nsnam/ns/>.
  - [6] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., Paxson, V., "Stream Control Transmission Protocol", RFC 2960, October 2000.
  - [7] Floyd, S., Henderson, T., "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 2582, April 1999.
  - [8] Brakmo, L. S., Peterson, L. L., "TCP Vegas: End to End Congestion Avoidance on a Global Internet", IEEE Journal on Selected Areas in Communications, Vol. 13, No. 8, pp. 1465-1480, October 1995.

- [9] Mascolo, S., Casetti, C., Gerla, M., Lee, S. S. and Sanadidi, M., "TCP Westwood: congestion control with faster recovery", UCLA CSD Technical Report #200017, 2000.
- [10] Mascolo, S., Casetti, C., Gerla, M., Sanadidi, M., Y. and Wang, R., "TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links", Proceedings of ACM Mobicom 2001, pp. 287-297, Rome, Italy, July 16-21, 2001.
- [11] Mathis, M., Mahdavi, J., Floyd, S., Romanow, A., "TCP Selective Acknowledgment Options", RFC 2018, October 1996.
- [12] Floyd, S., Mahdavi, J., Mathis, M., Podolsky, M., "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, July 2000.
- [13] Pentikousis, K., "TCP in wired-cum-wireless environments", IEEE Communications Surveys & Tutorials, Vol. 3, No. 4, pp. 2-14, Fourth Quarter 2000.
- [14] Ayanoglu, E., Paul, S., LaPorta, T. F., Sabnani, K. K. and Gitlin, R. D., "AIRMAIL: A Link-Layer Protocol for Wireless Networks", ACM Wireless Networks, Vol. 1, No. 1, pp. 47-60, February 1995.
- [15] Balakrishnan, H., Padmanabhan, V. N., Seshan, S. and Katz, R. H., "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links", Proceedings of ACM SIGCOMM '96, Stanford, CA, August 1996.
- [16] Balakrishnan, H., Seshan, S., Amir, E. and Katz, R. H., "Improving TCP/IP Performance over Wireless Networks", Proceedings of 1<sup>st</sup> ACM International Conference on Mobile Computing and Networking (Mobicom), pp. 2-11, November 1995.
- [17] Stangel, M. and Bharghavan, V., "Improving TCP performance in mobile computing environments", International Conference on Communications '98, pp. 584-589, Atlanta, GA, 1998.

- [18] Subramanian, V., Kalyanaraman, S. and Ramakrishnan, K. K., "An End-to-End Transport protocol for Extreme Wireless Network Environments", Military Communications Conference, MILCOM 2006, pp. 1-7, 2006.
- [19] Lien, Y. and Chung, Y., "Design of TCP Congestion Control Techniques by Router-Assisted Approach", Proceedings of The 12<sup>th</sup> Mobile Computing Workshop, March 31, 2006.
- [20] Bakre, A. and Badrinath, B. R., "I-TCP: Indirect TCP for Mobile Hosts", Proceedings of the 15<sup>th</sup> International Conference on Distributed Computing Systems, IEEE, pp. 136-143, May 1995.
- [21] Tsaoussidis, V. and Badr, H., "TCP-probing: Towards an error control schema with energy and throughput performance gains", Proceedings of the 8<sup>th</sup> IEEE Conference on Network Protocols, Japan, November 2000.
- [22] Ludwig, R., Katz, R. H., "The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions", ACM SIGCOMM Computer Communication Review, Vol. 30, Issue. 1, January 2000.
- [23] Ludwig, R. and Meyer, M., "The Eifel Detection Algorithm for TCP", RFC 3522, April 2003.
- [24] Barakat, C., Altman, E. and Dabbous, W., "On TCP Performance in a Heterogeneous Network: A Survey", IEEE Communications Magazine, Vol. 38, Issue. 1, pp. 40-46, January 2000.
- [25] Caini, C. and Firrincieli, R., "TCP Hybla: a TCP enhancement for heterogeneous networks", International Journal of Satellite Communications and Networking, Vol. 22, No. 5, pp. 547-566, September 2004.
- [26] Daniel, L., "Introduction to TCP and its Adaptation to Networks with Wired-cum-Wireless Links", Seminar on Transport of Multimedia Streams in Wireless Internet, September 23, 2003.
- [27] Chandran, K., Raghunathan, S., Venkatesan, S. and Prakash, R., "A Feedback Based Scheme For Improving TCP Performance In Ad-Hoc

Wireless Networks”, Proceedings of the 18<sup>th</sup> International Conference on Distributed Computing Systems (ICDCS), Amsterdam, May 1998.

- [28] Jacobson, V., Braden, R., Borman, D., “TCP Extensions for High Performance”, RFC 1323, May 1992.

