DIFFERENTIATION OF WIRELESS AND CONGESTION LOSSES IN TCP

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree
of
MASTER OF TECHNOLOGY
in
INFORMATION TECHNOLOGY

By

VIJENDER BUSI REDDY

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
JUNE, 2006

[illegible]

CANDIDATE'S DECLARARTION

I hereby declare that the work which is being presented in this dissertation entitled "**Differentiation of Wireless and Congestion Losses in TCP**" in partial fulfillment of the requirement for the award of the degree of Master of Technology in Information Technology, Submitted in the Department of Electronics and Computer Engineering of the Indian Institute of Technology Roorkee, Roorkee, is an authentic record of my own work carried out during the period july 2005 to june 2006, under the supervision of Prof. Anil K. Sarje.

The matter embodied in this dissertation has not been submitted by me for the award of any other degree.

Date: June 2006

Place: Roorkee

Vijender.B

(Vijender Busi Reddy)

CERTIFICATE

This is certify that the above statement made by the candidate is correct to the best of my knowledge.

[illegible]
21/6/06

(Prof. Anil K. Sarje)
Professor
Department of E &C
I.I.T. Roorkee

ACKNOWLEDGEMENT

It is my privilege to express my profound sense of gratitude and thanks to my guide **Dr. Anil K. Sarje, Professor, E&C Dept., IIT Roorkee** for his valuable guidance and suggestions throughout my dissertation work.

I owe a great deal of appreciation to **Dr D.K. Mehra, HOD, Department of Electronics and Computer Science Engineering** for providing all the resources needed for successful completion of this work.

I am also thankful to **Computer Center staff** members for their cooperation.

Next I feel indebted to all those endless researchers all over the world whose work I have used in my dissertation report. Their sincerity and devotion motivates me the most.

Finally, I would like to thank all my **friends** who cooperated all the time during this dissertation work.

Vijender.B

(Vijender Busi Reddy)

M.Tech(IT)

Dept. of E&C

ABSTRACT

Transmission Control protocol (TCP), the most widely used transport layer protocol on Internet, has attained significant maturity over the last few years and the popularity of wireless communication and computing systems is on the rise. Efforts are underway to extend TCP to wireless to enable smooth integration of the two technologies. Since TCP was developed for wired medium, wireless medium posed an altogether a new set of challenges to TCP. For this reason TCP requires improvements or modifications.

TCP assumes every packet loss is a congestion loss hence decreases the sending rate. This will decrease the sender's throughput when there is an appreciable rate of loss due to link error. This issue is significant for wireless links. We present an extension of TCP-Casablanca which improves TCP performance over wireless links. We proposed a new discriminator which not only differentiates congestion and wireless losses, but also identifies the congestion level in the network, i.e., whether the network is lightly congested or heavily congested and throttles the sender's rate according to the congestion level in the network.

CONTENTS

1.1 Introduction:

TCP is a popular protocol for reliable data delivery in the internet. Most current applications use TCP/IP protocols for data transfer. TCP assumes that every packet loss is an indication of network congestion and throttles the sender's rate. Although TCP was initially designed and optimized for wired networks, the growing popularity of wireless data applications has lead third generation wireless networks such as CDMA2000 and UMTS networks to extend TCP to wireless communications as well. The initial objective of TCP was to efficiently use the available bandwidth in the network and to avoid overloading the network (and the resulting packet losses) by appropriately throttling the senders' transmission rates. Network congestion is deemed to be the underlying reason for packet losses. Consequently, TCP performance is often unsatisfactory when used in wireless networks.

Wireless environments with transmission errors are becoming more common and TCP may perform poorly when wireless link subject to transmission errors. The reason for this is the implicit assumption in TCP that all packet losses are treated as congestion losses by decreasing the transmission window. Reno and Tahoe TCP implementations and many proposed alternative solutions [9, 10, 18] use packet loss as a primary indication of congestion; a TCP sender increases its window size, until packet losses occur along the path to the TCP receiver. Decreasing the congestion window when packet loss occurs due to lossy wireless links leads the performance degradation of TCP.

Currently, a TCP sender considers all losses as congestion signals. When loss occurs sender decreases its sending rate by halving its congestion window which is a part of congestion control activity. Taking congestion control actions may be appropriate when a packet loss is due to congestion, however, it can unnecessarily reduce sending rate if packet losses happen to be due to wireless transmission errors. Ideally, it would help the sender to differentiate between packet losses due to congestion from the packet losses due to wireless transmission errors using some technique. Once a sender knows that the packet loss is due to congestion or wireless transmission error, it can respond appropriately. The sender does not know exactly which packets are lost. The receiver has a better view of the losses; it knows exactly which packets are lost. This observation led us to consider schemes which can enable the receiver to distinguish between congestion losses and wireless error losses.

We extend TCP-Casablanca [12] to differentiate congestion and wireless losses, and identifying the network state by using RTT. Estimating network condition by using RTT (Round Trip Time) is not new, Brakmo and Malley proposed TCP-Vegas [3], which estimates network condition by using RTT and according to that it increases/decreases the congestion window. We use simulation to study the ability of discrimination of our method. Then we modify TCP-New Reno integrates our scheme and study the throughput enhancement induced. The TCP-New Reno modified with our scheme will be called TCP-RoS. We compare the performance of our scheme, TCP-RoS, with the TCP-Casablanca [12] and TCP-New Reno.

1.2 Statement of Problem:

Whenever there are wireless transmission errors in wireless networks TCP performance will degrade. This is because sender unnecessarily throttles the congestion window when packet loss occurs due to wireless error. As a result the bandwidth of the network gets under utilized. Our aim is to improve the performance of TCP in presence of wireless losses by de randomizing the congestion losses.

1.3 Organization of Report:

This paper is organized as follows.

Chapter 2 presented detailed description about TCP, TCP Congestion control algorithms and different discriminators used for differentiating wireless and congestion losses.

Chapter 3 describes the proposed discriminator, TCP-RoS (TCP with Router Support).

Chapter 4 describes implementation details, Network model and performance metrics used for evaluating the proposed discriminator.

Chapter 5 discusses the simulation results in terms of throughput and Accuracies.

Chapter 6 gives a brief conclusion and is followed by scope of future work.

2.1 TCP overview:

Transmission Control Protocol (TCP) was first introduced in early 1980s to provide reliable operation over a variety of transmission media. TCP is a means for building a reliable communications stream on top of the unreliable packet Internet Protocol (IP). TCP is the protocol that supports Internet applications. The combination of TCP and IP is referred to as TCP/IP.

TCP [16] has been widely used in today's Internet. The protocol supports reliable data transport by establishing a connection between the transmitting and receiving ends. The transmitter starts a timeout mechanism when sending a packet to the receiver. The transmitter constantly tracks the round-trip times (RTTs) for its packets as a means to determine the appropriate timeout period. At the receiver, each received packet is acknowledged implicitly or explicitly to the transmitter. If the transmitter does not receive an acknowledgment for a given packet when the corresponding timeout period expires, the packet is deemed to be lost and subject to retransmission. A congestion window with dynamically adjusted size is used by the protocol to regulate the traffic flow from the transmitter to the receiver.

TCP is being used on wired networks with stationary host for the last 2-3 decades. It has been adapted significantly to optimize its performance on these networks. The wired networks are highly reliable and less than 1 % of the packet losses can be attributed to link errors. TCP attributes packet loss on the network to congestion. TCP sender buffers the packets sent to the receiver. The receiver sends cumulative acknowledgments to indicate the receipt of the packets. The sender retransmits the lost packets to guarantee reliable delivery. For this purpose, it maintains a running average of the estimated round trip delay and the mean linear deviation from it.

The sender identifies the loss of a packet either by the arrival of several duplicate cumulative acknowledgments or by the absence of an acknowledgment for a packet within a timeout interval that is equal to the sum of the smoothed round-trip delay and four times its mean deviation. TCP reacts to packet losses by dropping its transmission (congestion) window size before retransmitting packets, initiating congestion control or avoidance mechanisms [7] (e.g., slow start) and backing off its retransmission timer (Karn's algorithm [20]). These measures result in a reduction in the load on the intermediate links, thereby controlling the congestion in the network.

2.2 TCP in Wireless Environment

The characteristics of wireless medium differ significantly from that of wired medium. The major factors affecting TCP performance in wireless environment are [8]:

1. **Limited Bandwidth:** Bit rates of 100 Mbps are common on wired LANs. Optical links provide data rate of the order of gigabits per second. As compared to this the current wireless standards for example the IEEE 802.11b standard for Wireless LAN offers raw bit rates of up to only 11 Mbps. Thus available bandwidth is one of the major bottlenecks that degrade the throughput of TCP on wireless medium.

2. **Long Round Trip Times:** In general, wireless media exhibit 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 in the case of wireless links. This directly limits the throughput of TCP on wireless links.

3. **Random Losses** The transmission losses on wireless medium are significantly higher than that on wired medium. These losses result in packet drops and hence the sender does not receive acknowledgments within retransmit timeout. This causes the sender to retransmit the segment, exponentially back off its retransmit timer and closes its congestion window to one segment. Repeated errors ensure a low throughput. The loss of packet on wireless link, which in general is the last hop,

results in end-to-end retransmission. This causes traffic overload on the wired links also.

Forward Error Correction (FEC) can be employed to bring the False Alarm Error Rate (FAER) down to the order of 10. However, FEC achieves such low FAER only under certain conditions at the expense of significant bandwidth, but bandwidth already a scarce resource in wireless medium. Hence FEC is usually not preferred. In addition, FEC cannot solve all problems because terrain type and natural and man-made objects can handicap wireless connectivity altogether.

4. **User Mobility** In the case of cellular networks when a user (mobile host) moves from one cells to another, all the necessary information has to be transferred from the previous base station to the new base station. This 2 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 the case of ad hoc networks, mobile nodes can move randomly causing frequent topology changes. This causes packet losses and forces mobile hosts to initiate route discovery algorithms frequently. The overall result is significant throughput reduction.

5. **Short Flows** Services like web browsing and e-mail involve small amount of data transfer between the client and the server. A major portion of the wireless networks data transfer fall in this category. The TCP sender increases its congestion window progressively as it receives acknowledgments from the receiver (Slow Start). There is a high probability that the transfer completes even before the sender's window reaches the maximum possible size. This results in under utilization of the network capacity.

6. **Power Consumption** The retransmissions caused by frequent packet losses result in longer connection duration and hence higher power consumption. Power consumption is a very important factor in case of battery operated devices like laptops, PDAs and wireless phones.

2.3 TCP Congestion control [20]:

TCP uses end-to-end congestion control. This means that congestion control is not network-assisted. There is no explicit feedback from the network and congestion is inferred from the observed end-systems loss and delay. Using congestion control, transmission rate is limited by the congestion window size (in short cwnd) over segments as shown in the following figure 2.1 [20].

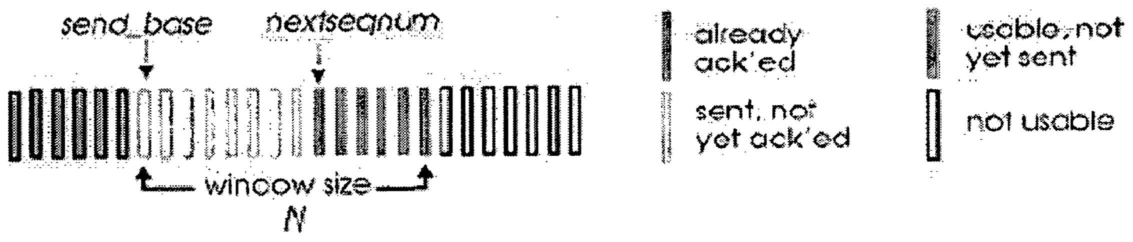

Fig 2.1[20]: TCP congestion window

TCP congestion control starts with probing for usable bandwidth. Ideally someone should transmit as fast as possible without loss. Instead cwnd increases until loss. When loss occurs, we decrease cwnd and then begin probing (increasing cwnd) again. During this process there are two phases. The first is the slow start and the second is the congestion avoidance. Important variables are *cwnd* and the *threshold* that defines the boundary between the slow start phase and the congestion avoidance phase.

2.4 Congestion Control Algorithms:

There are four congestion control algorithms: slow start, congestion avoidance, fast retransmit and fast recovery. In some situations it may be beneficial for a TCP sender to be more conservative than the algorithms allow, however a TCP must not be more aggressive than the following algorithms allow (that is, must not send data when the value of cwnd computed by the following algorithms would not allow the data to be sent).

2.4.1 Slow Start and Congestion Avoidance [1]

The slow start and congestion avoidance algorithms must be used by a TCP sender to control the amount of outstanding data being injected into the network. To implement these algorithms, two variables are added to the TCP per-connection state. The congestion window (*cwnd*) is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgment (ACK), while the receiver's advertised window (*rwnd*) is a receiver-side limit on the amount of outstanding data. The minimum of *cwnd* and *rwnd* governs data transmission.

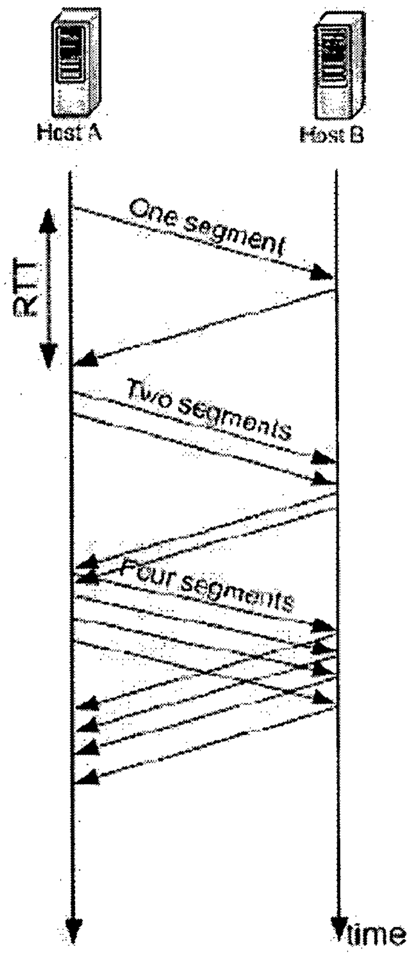

Fig 2.2: Slow Start

Another state variable, the slow start threshold (*ssthresh*), is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission, as discussed below.

Beginning transmission into a network with unknown conditions requires TCP to slowly probe the network to determine the available capacity, in order to avoid congesting the network with an inappropriately large burst of data. The slow start algorithm (shown in Fig 2.2) is used for this purpose at the beginning of a transfer, or after repairing loss detected by the retransmission timer.

IW, the initial value of *cwnd*, must be less than or equal to 2*SMSS (Sender Maximum Segment Size) bytes and must not be more than 2 segments. We note that a non-standard, experimental TCP extension allows that a TCP may use a larger initial window (IW), as defined in equation 1

$$IW = \min(4*SMSS, \min(2*SMSS, 4380 \text{ bytes})) \quad \ldots\ldots \quad (1)$$

With this extension, a TCP sender may use 3 or 4 segment initial window, provided the combined size of the segments does not exceed 4380 bytes. We do not allow this change as part of the standard defined by this document.

The initial value of *ssthresh* may be arbitrarily high (for example, some implementations use the size of the advertised window), but it may be reduced in response to congestion.

The slow start algorithm is used when *cwnd* < *ssthresh*, while the congestion avoidance algorithm is used when *cwnd* > *ssthresh*. When *cwnd* and *ssthresh* are equal the sender may use either slow start or congestion avoidance.

During slow start, a TCP increments cwnd by at most SMSS bytes for each ACK received that acknowledges new data. Slow start ends when *cwnd* exceeds *ssthresh* or when congestion is observed. During congestion avoidance, *cwnd* is incremented by 1 full-sized segment per round-trip time (RTT). Congestion avoidance continues until congestion is detected. One formula commonly used to update *cwnd* during congestion avoidance is given in equation 2:

$$cwnd = cwnd + \mathrm{SMSS}*\mathrm{SMSS}/cwnd \qquad (2)$$

This adjustment is executed on every incoming non-duplicate ACK. Equation (2) provides an acceptable approximation to the underlying principle of increasing cwnd by 1 full-sized segment per RTT.

2.4.2 Fast Retransmit and Fast Recovery [1]

A TCP receiver should send an immediate duplicate ACK when an out-of-order segment arrives. The purpose of this ACK is to inform the sender that a segment was received out-of-order and which sequence number is expected. From the sender's perspective, duplicate ACKs can be caused by a number of network problems. First, they can be caused by dropped segments. In this case, all segments after the dropped segment will trigger duplicate ACKs. Second, duplicate ACKs can be caused by the re-ordering of data segments by the network. Finally, duplicate ACKs can be caused by replication of ACK or data segments by the network. In addition, a TCP receiver should send an immediate ACK when the incoming segment fills in all or part of a gap in the sequence space. This will generate more timely information for a sender recovering from a loss through a retransmission timeout, a fast retransmit, or an experimental loss recovery algorithm, such as NewReno. The TCP sender should use the "fast retransmit" algorithm to detect and repair loss, based on incoming duplicate ACKs. The fast retransmit algorithm uses the arrival of 3 duplicate ACKs (3 identical ACKs without the arrival of any other intervening packets) as an indication that a segment has been lost. After receiving 3 duplicate ACKs, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire.

After the fast retransmit algorithm sends what appears to be the missing segment, the "fast recovery" algorithm governs the transmission of new data until a non-duplicate ACK arrives. The reason for not performing slow start is that the receipt of the duplicate ACKs not only indicates that a segment has been lost, but also that segments are most likely leaving the network (although a massive segment duplication by the network can invalidate this conclusion). In other words, since the receiver can only generate a duplicate ACK when a segment has arrived, that segment has left the network and is in the receiver's buffer, so we know it is no longer consuming network resources.

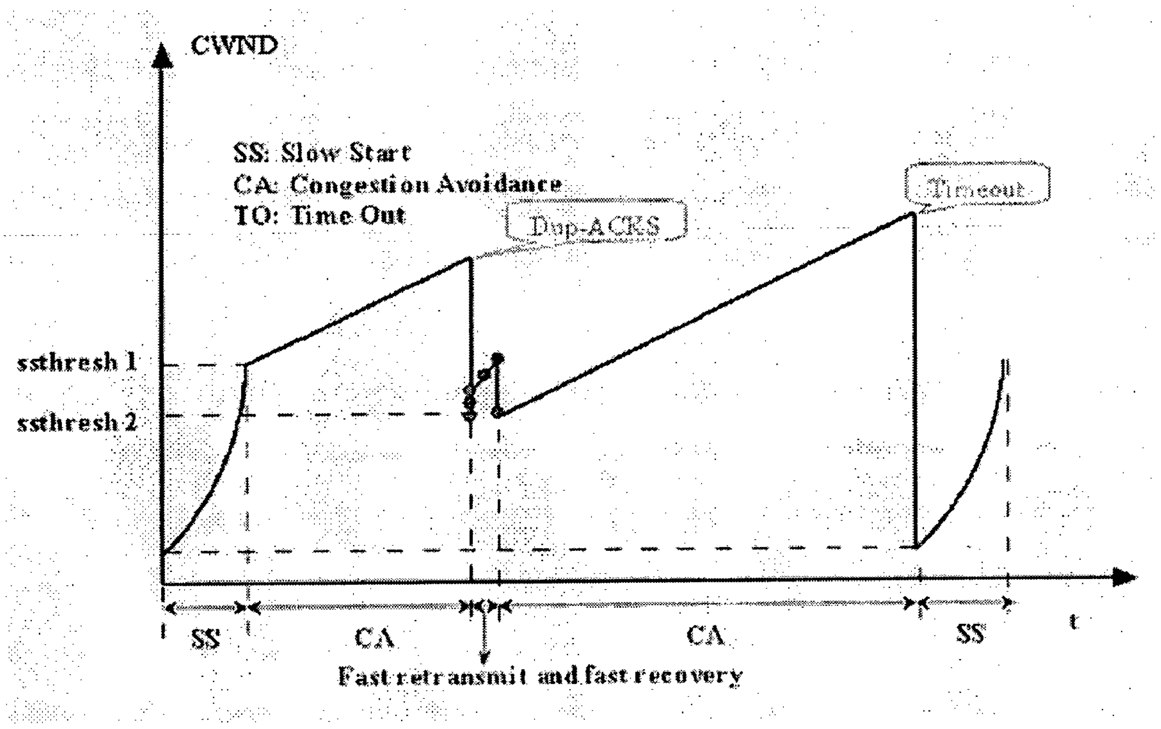

Fig 2.3: TCP Congestion Window

Furthermore, since the ACK "clock" is preserved, the TCP sender can continue to transmit new segments (although transmission must continue using a reduced *cwnd*). The fast retransmit and fast recovery algorithms (Shown in Fig 2.3) are usually implemented together as follows.

1. When the third duplicate ACK is received, set *ssthresh* to max (FlightSize / 2, 2*SMSS). Where, FlightSize is the amount of outstanding data in the network

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

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

4. Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.

5. When the next ACK arrives that acknowledges new data, set cwnd to ssthresh (the value set in step 1). This is termed "deflating" the window.

This ACK should be the acknowledgment elicited by the retransmission from step 1, one RTT after the retransmission (though it may arrive sooner in the presence of significant out-of-order delivery of data segments at the receiver). Additionally, this ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost.

2.4.3 TCP Tahoe, Reno, New Reno, Vegas and SACK:

The error control mechanism of TCP is primarily oriented towards congestion control. Congestion control can be beneficial to the flow that experiences congestion, since avoiding unnecessary retransmissions can be lead to better throughput-delay tradeoff. TCP utilizes acknowledgments to pace the transmission of segments and interprets timeout events as signs of congestion. In response to congestion, the TCP sender reduces the transmission rate by shirking its window. There are four major versions of TCP (Tahoe, Reno, New Reno and Vegas). In the following lines we discuss each version:

TCP Tahoe [20]: TCP Tahoe is the oldest version of TCP but on the other hand one of the most famous versions. TCP Tahoe congestion algorithm includes Slow Start and

Congestion Avoidance. In order to declare a loss event, three timeouts have to be passed. This is its main drawback as when a segment is lost; the sender side of the application may have to wait a long period of time for the timeout. After timeout it again starts from Slow Start phase.

TCP Reno [20]: TCP Reno, except from Slow Start and Congestion Avoidance also includes Fast Retransmit and Fast Recovery. A receiver sends a duplicate ACK immediately on reception of each out-of-sequence packet. The Reno TCP transmitter interprets reception of three duplicate ACKs as a congestion packet loss, and sets the slow start threshold size (*ssthresh*) to one-half of the current congestion window (*cwnd*) and retransmits the missing packet. The *cwnd* is then set to *ssthresh* plus three times the segment size (one per each duplicate ACK). *cwnd* is increased by one segment on reception of each duplicate ACK, which continues to arrive after fast retransmission. This allows the transmitter to send new data when *cwnd* is increased beyond the value of the *cwnd* before the fast retransmission.

When an ACK arrives, which acknowledges all outstanding data sent before the duplicate ACKs were received, the *cwnd* is set to *ssthresh* so that the transmitter slows down the transmission rate and enters the linear increase phase.

TCP Reno's Fast Recovery can be effective when there is only one segment drop from a window of data, given the fact that Reno retransmits at most one dropped segment per RTT. The problem with the mechanism is that is not optimized for multiple packet drops form a single window, and this could negatively impact performance.

TCP New Reno [5]: If two or more packets have been lost from the transmitted data (window), the fast retransmission and fast recovery algorithms (Reno) will not be able to recover the multiple losses without waiting for retransmission time out. Hoe proposed a modification to Reno TCP usually called New-Reno to overcome this problem. New-Reno introduces the concept of a fast retransmission phase, which starts on detection of a packet loss (receiving three duplicate ACKs) and ends when the receiver acknowledges reception of all data transmitted at the start of the fast retransmission phase. The transmitter assumes reception of a partial ACK (acknowledging some, but not all, packets outstanding at the start of fast retransmission phase) during the fast retransmission phase as an indication that another packet has been lost within the window, and retransmits that packet immediately to prevent expiry of the retransmission timer.

TCP Vegas [3]: TCP Vegas approaches the problem of congestion from another perspective. The basic idea is to detect congestion in the routers between source and destination before packet loss occurs and lower the rate linearly when this imminent packet loss is detected. The longer the round-trip times of the packets, the greater the congestion in the routers. Every two round trips delays the following quantity is computed:

$$\rho = (WindowSizeCurrent - WindowSizeOld) * (RTTCurrent - RTTOld)$$

```
If $\rho > 0$
    The window size is decreased by 1/8.
Else
    The window size is increased by one segment size.
```

One problem that it does not seem to overcome is the path asymmetry. The sender makes decisions based on the RTT measurements, which, however, might not accurately indicate the congestion level of the forward path. Furthermore, packet drops caused by retransmission deficiencies or fading channels may trigger a Slow Start. However, this problem is common to all of the above versions. Another drawback is that, Vegas algorithm is very new (1999) and is not fully embedded in the most popular TCP implementations.

TCP Sack [9]: The selective acknowledgment (SACK) option for Reno TCP has been introduced to further enhance TCP performance. When the receiver buffer holds in-sequence data packets, the receiver sends duplicate ACKs bearing the SACK option to inform the transmitter about which packets have been correctly received. This allows the transmitter to modify the retransmission procedure to selectively retransmit only lost packets, without retransmitting already Sacked packets. The transmitter is also able to accurately estimate the number of transmitted packets that have left the network by using the explicit information carried by SACKS. This provides efficient transmission of more packets to utilize the network. The SACK option has been implemented in most of the recent releases of operating systems, while New-Reno implementations are still emerging.

If the receiver does not acknowledge the packet and, instead, acknowledges (SACK), a subsequently transmitted packet, the transmitter considers that this is a good indication of loss of the retransmitted packet. All these modifications to TCP assume that every packet loss is an indication of network congestion and take measures to avoid further congestion in the network by reducing the transmission rate. This results in a very low utilization of the link when there is an appreciable rate of losses due to link errors.

2.5 Queuing types:

2.5.1 Drop-Tail queue:

The most well known operation of the queue is the First-In-First-Out (FIFO) queue process. FIFO queue shared by all packets to be transmitted over an out-bound link. The queue simply provides some capacity for tolerating variability in the load (*i.e.*, *B*ursty traffic) on the outbound link. A short burst of packet arrivals may exceed the available bandwidth of the link even when the average load is well below the link bandwidth. However, when the load exceeds the available capacity of the link for sustained periods of time, the queue capacity is exceeded. Router implementations using a simple fixed-size FIFO queue typically just drop any packet that arrives to be enqueued to an already-full outbound queue. This behavior is often called *drop-tail* packet discarding.

2.5.2 Random Early Detection (RED)

The RED algorithm uses a weighted average of the total queue length to determine when to drop packets. When a packet arrives at the queue, if the weighted average queue length is less than a minimum threshold value, no drop action will be taken and the packet will simply be enqueued. If the average is greater than a minimum threshold value but less than a maximum threshold, an early drop test will be performed as described below.

An average queue length in the range between the thresholds indicates some congestion has begun and flows should be notified via packet drops. If the average is greater than the maximum threshold value, a forced drop operation will occur. An average queue length in this range indicates persistent congestion and packets must be dropped to avoid a persistently full queue. Note that by using a weighted average, RED avoids over-reaction to bursts and instead reacts to longer-term trends. Furthermore, note that because the thresholds are compared to the weighted average (with a typical weighting of 1/512 for the most recent queue length), it is possible that no forced drops will take place even when the instantaneous queue length is quite large.

2.6 Wireless and Congestion loss discriminators:

There has been considerable work characterizing the benefits of differentiating wireless losses from congestion losses for TCP connections, and developing various techniques for preventing TCP from reacting to wireless losses as if they indicated congestion. A *discriminator* is any technique which distinguishes congestion losses from wireless losses.

A. Biaz discriminator [11]:

Biaz proposed one discriminator uses packets inter arrival time to differentiate between loss types. The concept here is that based on the arrival time of P_i, if P_{i+n+1} arrives right around the time that it should have arrived, we can assume the missing packets were properly transmitted and lost due to wireless errors. If P_{i+n+1} arrive much earlier than it should, then at least some packets ahead of it (P_{i+1}... P_{i+n}) probably were dropped at a buffer, and if it arrives much later than expected, then it is likely that queuing times at buffers have increased. Either way, we can attribute the loss to congestion.

B. Spike discriminator [17]:

The Spike discriminator differentiated among degrees of congestion but did not explicitly differentiate wireless loss from congestion loss. ROTT is a measure of the time a packet takes to travel from the sender to the receiver and used to identify the state of the current connection. If the connection is in the *spike state*, losses are assumed to be due to congestion; otherwise, losses are assumed to be wireless. The spike state is determined as follows. On receipt of a packet with sequence number i, if the connection is currently not in the spike state and the ROTT for packet exceeds the threshold, then the algorithm enters the spike state. Otherwise, if the connection is currently in the spike state, and the ROTT for packet is less than a second threshold, the algorithm leaves the spike state. When the receiver detects a loss because of a gap in the sequence number of received packets, it classifies the loss based on the current state.

The problem in this method is, it is sensitive to threshold values. The distance between these two parameters determines the stability between spike and non-spike states. If d (difference between two threshold values) is small then the algorithm oscillates between two states. If d is large the algorithm is stable but misclassification of losses increases.

C. ZigZag discriminator [4]:

ZigZag classifies losses as wireless, based on the number of losses n and the difference between $rott_i$ and its $rott_{mean}$. A loss is classified as wireless if

$$(n=1 \text{ AND } rott_i < rott_{mean} - rott_{dev})$$
$$\text{OR } (n=2 \text{ AND } rott_i < rott_{mean} - rott_{dev}/2)$$
$$\text{OR } (n=3 \text{ AND } rott_i < rott_{mean})$$
$$\text{OR } (n>3 \text{ AND } rott_i < rott_{mean} - rott_{dev}/2).$$

Otherwise, the loss is classified as congestion loss. $rott_{mean}$ and $rott_{dev}$ calculated as follows:

$$rott_{mean} = (1-\alpha)*rott_{mean} + \alpha*rott$$
$$rott_{dev} = (1-2\alpha)*rott_{dev} + 2\alpha*|rott - rott_{mean}|.$$

By definition, ROTT has a high probability of having values greater than ($rott_{mean}$-$rott_{dev}$): 84% if it were a normalized Gaussian distributed random variable. As one packet loss is the most common loss pattern in a wired network, and congestion loss usually comes with higher delay, the threshold of $rott > rott_{mean}-rott_{dev}$ intuitively would classify most of the congestion loss correctly. The reasoning behind increasing the threshold with the number of losses encountered is that a more severe loss is associated with higher congestion and with higher ROTT. This way, a loss event containing four or more packets would be classified as congestion loss only when relatively large ROTT were observed.

Taking ROTT to classify wireless losses leads to the problem of high misclassification of congestion and wireless losses. The problem with this type of algorithms is, the sender and receiver must be synchronized.

D. Hybrid discriminator:

Cen et al. proposed a hybrid discriminator ZBS [4] which uses three loss discriminators: ZigZag, Biaz and Spike. ZBS dynamically switches between the three loss discriminators according to observed network conditions.

E. TCP Casablanca [12]:

Biaz and Vaidya proposed this discriminator that takes support from intermediate routers and uses different discard priority packets and biased queue management that first drops low priority packets if queue is full. Our algorithm is based on TCP-Casablanca [12].

This algorithm "de-randomizes" congestion losses such that the distribution of congestion losses differs from that of wireless error losses. In this algorithm the packets are divided in to two priorities *in* and *out.* One packet out of k packets labeled as *out.*

Let us denote all data packets a TCP sender sends as P_i^M: P_i^M is the *i*th packet, and it is marked with M (*in* or *out*). A retransmitted packet keeps the same index *i* but is always marked in to avoid jeopardy for packets initially marked *out*. It is assumed that routers first drop packets marked *out*, and start dropping packets marked *in* only if there are no more packets marked *out* in the queue.

TCP sender marks the packets such that one packet out of k packets is marked *out* all other packets are marked *in*. A biased queue management is implemented at routers which first drop *out* packets. At the receiver side when ever an out-of-order packet received, TCP receiver considers the patter of losses between the next expected packet P_{nxt} and the Packet P_{hi} with the highest sequence number seen so far and calculated the following function.

$$F(x, r, k) = 1- \left\lfloor k.\frac{x}{r} \right\rfloor$$

x = number of *out* packets lost.

r = total number of lost packets.

If F(x, r, k) $\geq$ 0 then the losses are diagnosed as wireless losses.

Otherwise they are diagnosed as congestion losses.

After identifying the loss type receiver sends signal to sender (by setting ELN flag) in duplicate ACK. If the ELN (Explicit Loss Notification) flag is set then the sender doesn't halve the congestion window (because the loss is due to wireless loss). Otherwise it decreases the congestion window to half.

If the congestion is high then F(x, r, k) mistakenly identifies the loss is due to wireless loss. If this happens then the sender doesn't decrease it's sending rate, due to this the network performance will degrade.

The biased queue management raises the issue of fairness between flows that mark some of their packets *out* and the flows that do not. If some flows do not mark *out* any of their packet, then packets will be dropped only from flows that mark packets *out*. Only flows that mark packets *out* would be responsive to light congestion. Flows that do not mark packets *out* may monopolize the available link capacity.

Note that some performance improvement is usually obtained even with weak loss discriminators. This observation is supported by Barman and Matta [2] who studied the effectiveness of poor loss discriminators in improving TCP performance. Barman and Matta showed that despite a low accuracy in diagnosing congestion losses or wireless losses, TCP performance can still be significantly improved.

In this chapter we discussed overview of TCP, major factors affecting TCP performance in wireless medium, various congestion control algorithms in TCP and different discriminators used for differentiating wireless and congestion losses.

In this chapter we discuss our proposed discriminator TCP-RoS (TCP with Router Support).

3.1 Rationale of proposed discriminator

The problem of distinguishing congestion losses from random wireless losses is particularly hard when congestion is light: congestion losses themselves appear to be random. A biased queue management that first drops specifically marked packets will "*de-randomize*" congestion losses. The TCP-RoS discriminator is implemented at the TCP receiver because the receiver has a better "view" of the losses than the sender.

For every k packets which the sender sends, it marks one of the packets as *out* and all other packets as *in*. In case of congestion, the router at the bottleneck link must first drop packets marked *out* before dropping any packet marked *in*. Therefore, the *out* packets are dropped first by the biased queue management at the bottleneck. If there is no congestion, the pattern of dropping will likely be different as wireless links do not distinguish between *out* and *in* packets. As high proportion of packets are marked as *in*, it is expected that these will likely be dropped. From this observation, the receiver will diagnose a pattern of losses as *biased* if a high proportion of packets marked *out* are lost. If the pattern appears to be biased, the receiver concludes that the losses are due to congestion. In the following, it is formally shown that a biased queue management enables the design of a very accurate TCP-RoS loss discriminator. The sender also estimates congestion level of the network by using RTT.

3.2 Proposed discriminator (TCP-RoS):

Our proposed discriminator TCP-RoS, is using priority packets to de-randomize congestion losses. This new method also identifies congestion level in the network and also solves fairness problem which is in TCP-Casablanca [12].

Sender in the TCP-RoS assigns priority (*in* or *out*) to every packet according to the value of k (one *out* packet every k *in* packets). And when the sender receives ACK from receiver it checks *eln_flag* field in ACK. If *eln_falg* is 1, then sender doesn't decrease congestion window. Other wise it checks the following equation 3.

$$rtt <= (min\_rtt+max\_rtt)/2 \qquad (3)$$

Where *rtt* is the round trip time, *min_rtt* is the minimum rtt and *max_rtt* is the maximum rtt till now the sender experienced. If this equation satisfies then the sender assumes that the network is lightly congested so it decreases congestion window by ¼ th only. Otherwise it acts as same New Reno. The flow chart for sender is shown in Fig 3.1.

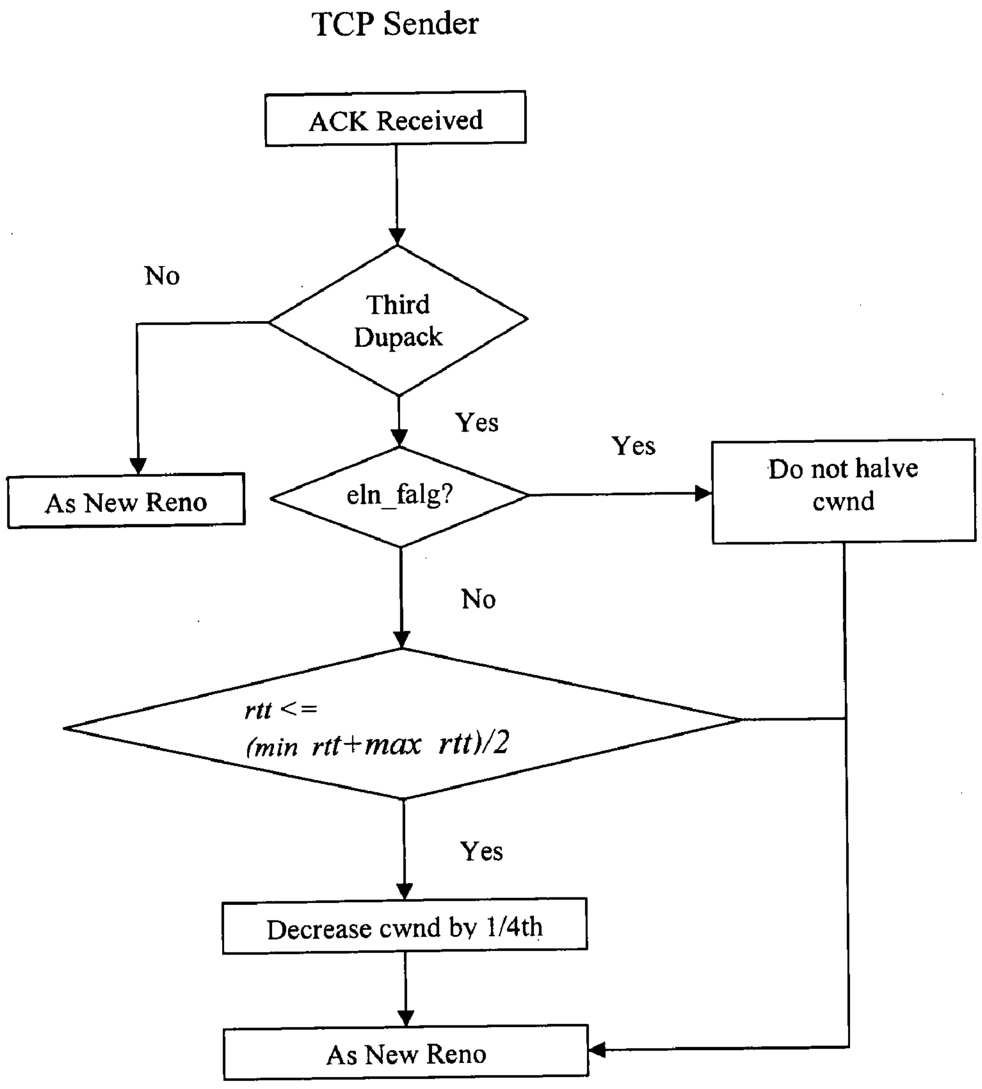

Fig 3.1: TCP-RoS Sender flow chart

We implemented discriminator at receiver side because receiver is better view for losses. When receiver receives out-of-order packet, it calculates two parameters x, r. where x is the no. of *out* packets lost. r is the total no. of packets lost. Receiver checks sequence no. from first lost packet sequence no. to maximum sequence no. seen up to now and calculates x and r parameters.

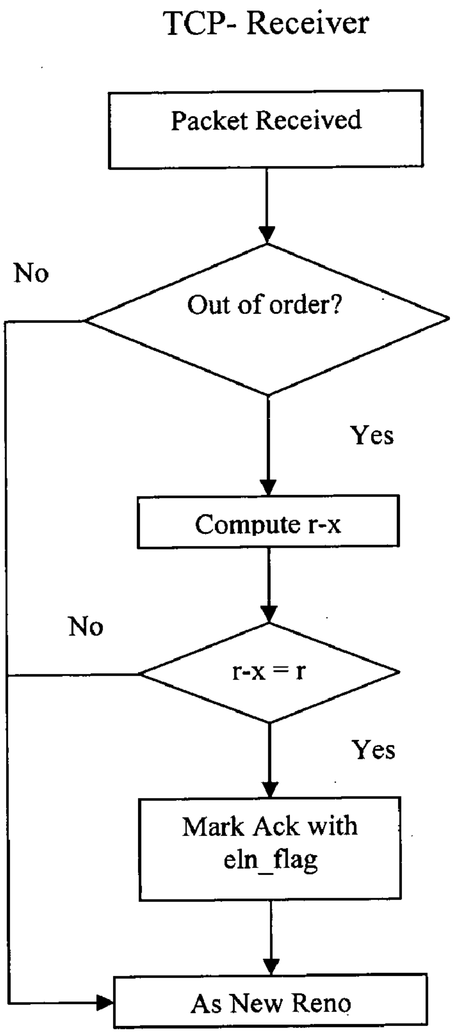

Fig 3.2: TCP-RoS Receiver flow chart

If all are *in* packets then receiver sends wireless loss signal to sender in third duplicate ACK by setting *eln_flag* field to 1 in ACK. Otherwise receiver sends congestion loss signal to sender by setting *eln_flag* field to 0 in ACK. The flow chart for receiver is shown in Fig 3.2.

We implemented a new queue management which first drops *out* priority packets. This queue management also solved fairness problem which is in TCP-Casablanca queue management. Our queue management works as follows.

When a packet comes to the router it checks the source and destination addresses of that packet, if this packet has priority *out*, then the sender drops this packet. Otherwise it checks with the packets which are in queue for the same source and destination number packets. If any *out* priority packet found in the queue it removes that packet otherwise this packet will be dropped. If there is no same source and destination address of that new incoming packet then it drops new incoming packet.

Simulation study is carried out using network Simulator -2. Overview of NS-2, the Network model used and the performance metrics used for evaluating the discriminators are as follows.

4.1 NS-2 overview:

NS-2 provides a frame work for simulation of wired and wireless networks, including some facility for emulation. NS-2 is the VINT project which is a joint effort by people from UC Berkely, USC/ISI, LBL, and Xerox PARC. The project is supported by the Defense Advanced Research Projects Agency (DARPA). The NS-2 simulator is written in C++ with a Tcl shell front-end that uses oTcl (object-oriented Tcl) libraries scenarios are run by feeding an oTcl script to the NS-2 executable. The output can be read directly or post-processed by an interactive graphics viewer called Network Animator (NAM). NAM does not allow changing parameters on the fly, it is for post-viewing of a simulation dump (a .nam file).

NS is an object oriented simulator, written in C++, with an OTcl interpreter as a front end. NS uses two languages because simulator has two different kinds of things it needs to do. On one hand, detailed simulations of protocols require a systems programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (find bug, fix bug, recompile, re-run) is less important.

NS meets both of these needs with two languages, C++ and OTcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration.

NS-2 is a discrete event simulator. It does the Simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. This Simulator is written in C++. OTcl is used as command and configuration interface.

Components of NS:

- Ns, the simulator itself
- Nam, the network animator
    - Visualize *ns* output
    - Nam editor: GUI interface to generate ns scripts
- Pre-processing:
    - Traffic and topology generators
- Post-processing:
    - Simple trace analysis, often in Awk, Perl, or Tcl

Ns functionalities in Wired Networks:

- Routing DV, LS, PIM-SM
- Transportation: TCP and UDP
- Traffic sources: web, ftp, telnet, cbr
- Queuing disciplines: drop-tail, RED, FQ, SFQ, DRR
- Tracing, visualization, various utilities

To the C++ programmer, object-oriented programming in OTcl may feel unfamiliar at first. The difference between C++ and OTcl are:

- Instead of a single class declaration in C++,we write multiple definitions in OTcl.
- Instead of a constructor in C++, write an `init` instproc in OTcl. Instead of a destructor in C++, write a `destroy` instproc in OTcl.
- Unlike C++, OTcl methods are always called through the object. The name `self`, which is equivalent to `this` in C++, may be used inside method bodies.

C++ is used for per packet processing i.e. preprocessing for each packet of a flow. It is fast to run, detailed, familiar and easy to understand.

OTcl is used for control of execution. The simulation program written in OTcl is fast to write and change. OTcl is used in

- Simulation scenario configurations
- Periodic or triggered action
- Manipulating existing C++ objects

4.2 System Requirements:

This project was developed on a Linux machine running Fedora 2 with Linux kernel 2.6.5 & NS-2. The NS-2 [19] version was "ns-allinone-2.1b8a", which is a single tar ball with all the requisite packages that easily installs with one command.

4.3 Network Model:

Accuracies (A_c and A_w) of TCP-RoS discriminator and the improvement are measured using ns-2[19] simulations. This section presents the topology used for the simulations, the packet loss model, and the method used to collect the data. Fig. 6(a) shows the topology used. There are three types of pairs *sender-receiver*: TCP connections over type *sender-receiver* pair experience the longest propagation delay path with a wireless last hop. There are five routers. The dashed lines show the TCP transfers between senders and receivers. With this topology, a competing TCP traffic with different round-trip times is maintained. Bit rates on all links are set such that the bottleneck is the link $R_3 - R_4$.

All senders are TCP senders. Sources are fed with FTP traffic. The results presented in this paper have a bit rate on the wired bottleneck of 45 Mb/s and propagation delay is 5ms. Bit rate B_{ws} on the wireless link of 10 Mb/s and N1, N2, N3 are 6. Experiments were run with N1, N2 and N3 are varying from 3 to10.

For the wireless packet loss model, a two-state Markov model is used. In each state, the time between successive losses is exponentially distributed.

Two discriminators were added to TCP sink in ns-2: Casablanca discriminator and proposed discriminator (TCP-RoS). For these experiments, each experiment lasts 140s.

Accuracies A_c, A_w and throughput are collected from this experiment. Note that the same starting times are used to conduct the experiment with TCP-New Reno, TCP-Casablanca, and TCP-RoS. Five runs of the same experiment were run, changing only the starting times. The results reported here for the accuracies and the throughput are the average over the 5 runs.

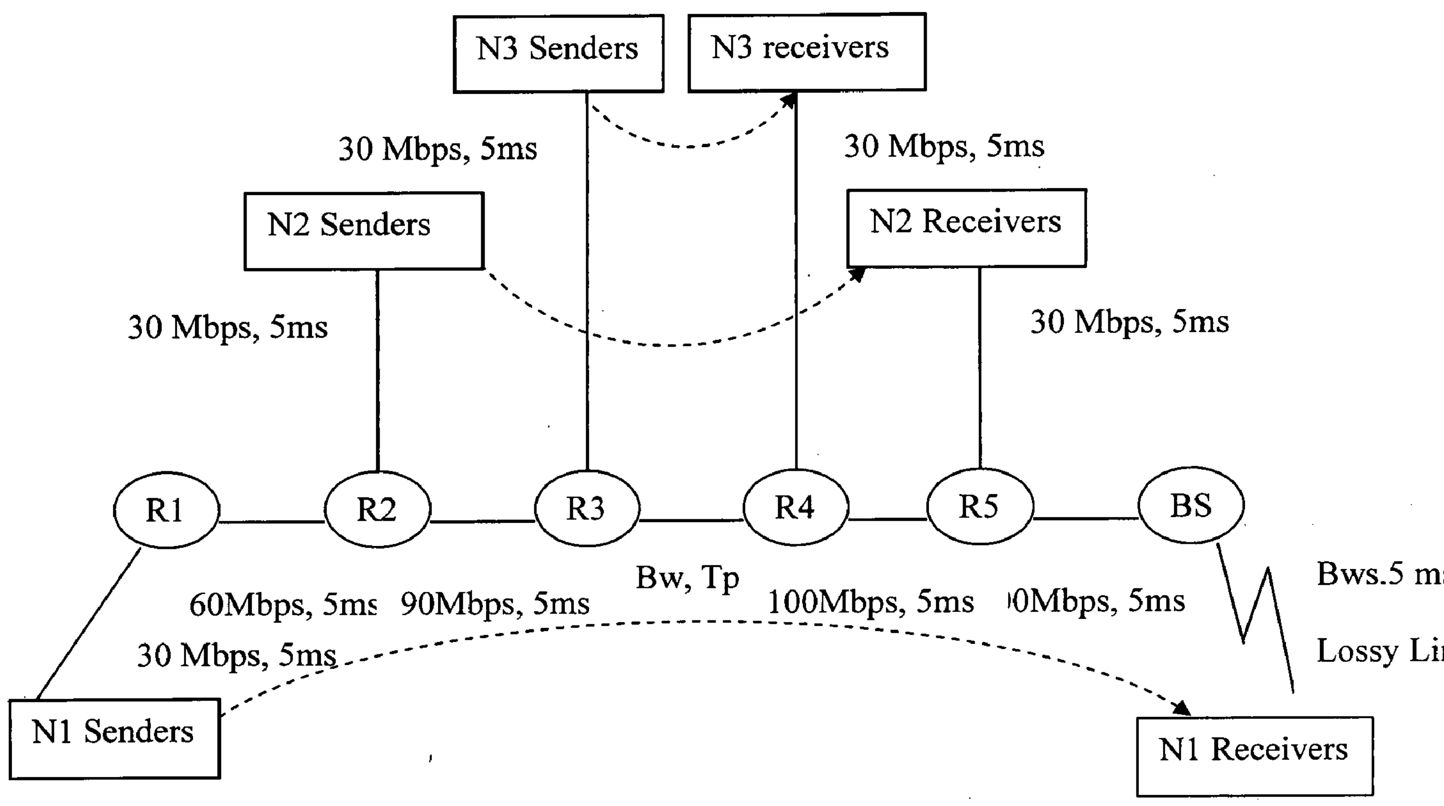

Fig 4.1: Network model

4.4 Performance metrics

An algorithm that attempts to classify each loss into one of two classes can be judged by its misclassification rate, the fraction of cases which are classified incorrectly. Since misclassifying a wireless loss as a congestion loss does not have the same impact as the other way around, we can judge performance by examining the two separate misclassification rates. However, our ultimate concern is with the throughput of the traffic stream that results from using the algorithm, and with whether the algorithm causes severe congestion and thereby diminishes the throughput of other traffic streams. This leads us to a set of three performance measures.

Throughput: The most important goal is high throughput, where we are concerned with the improvement compared with the original TCP-New Reno and TCP-Casablanca when transmitting through a network with a wireless link. Our experiments show that throughput of our model is higher than New Reno and Casablanca models.

Congestion Accuracy (A_c): A_c is the ratio of the number of congestion losses correctly diagnosed over the total number of congestion losses.

Wireless Accuracy (A_w): A_w is the ration of the number of wireless losses correctly diagnosed over the total number of wireless losses.

4.5 Description of the classes used

We have used the NS-2 simulator to modify the TCP. The TCP module already integrated in NS-2. The following functions are modified in various file in tcp *(See Appendix- for complete code).*

- **Modifications in tcpnewreno.cc**

output(seqno, reason) - The function sends one packet with the given sequence number and updates the maximum sent sequence number variable (`maxseq_`) to hold the given sequence number if it is the greatest sent so far. This function also assigns the various fields in the TCP header (sequence number, timestamp, reason for transmission). This function also sets a retransmission timer if one is not already pending. It assigns the priorities to every packet according to k value.

recv() - this function is the main reception path for ACKs. Note that because only one direction of data flow is in use, this function should only ever be invoked with a pure

ACK packet (i.e. no data). The function stores the timestamp from the ACK in ts_peer_, and checks for the presence of the ECN bit (reducing the send window if appropriate). If the ACK is a new ACK, it calls **newack()**, otherwise checks *eln_flag* whether the loss is due to wireless or congestion and calls dupackaction() by setting loss_type variable. It sends a packet by calling send_much.

Dupack_action()- This function decreases the congestion window according to the loss_type.

- **Modifications in tcpsink.cc**

recv() – This function is main reception path for packets. It checks whether the expected packet or out-of-order packet. This function also updates receive window when it received expected packet and sends ack for that packet. If the packet is out-of-order packet this function checks whether the loss is due to wireless or due to congestion and sets eln_flag field in third duplicate ack.

- **Modification in drop_tail.cc**

enque(Packet p) – This function enque the packet p in to the queue. Then it checks whether the present queue length is greater than queue length, if it is then this function calls dequebqm(), which is base class method.

- **Modification in queue.cc**

dequebqm() – This function deque the packet from queue. This function first drops low priority *(out)* packet for the same flow belongs to tail end packet.

5.1 Accuracies A_c and A_w:

It was shown in [12] that the accuracies Ac and A_w depend on the value k (one packet marked out every k packets, others being marked in). In this section, the relationship between the accuracies and k is verified through simulations.

Impact of k on A_c and A_w:

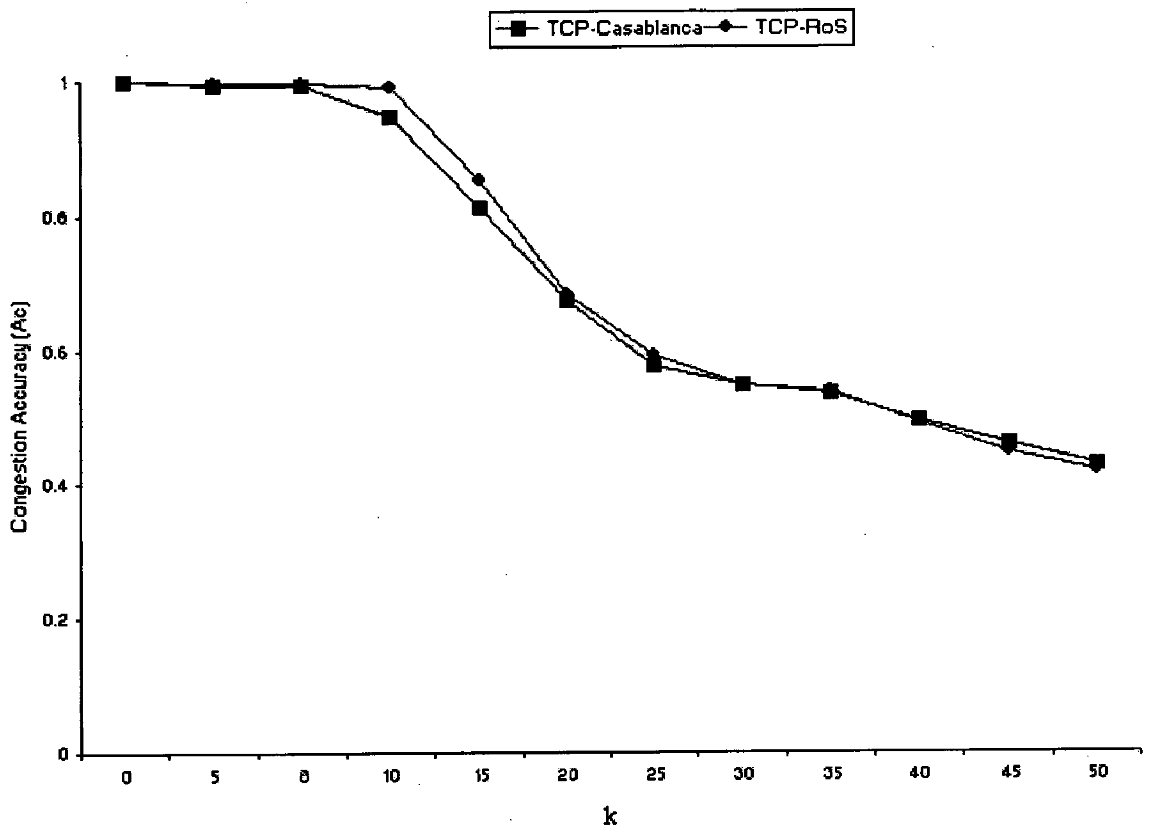

Fig. 5.1: Congestion accuracy A_c Vs k when p_w =0.01.

Fig 5.1 plots the measured Congestion accuracy versus k when wireless packet loss rate p_W =0.01, the key observation is that accuracy decreasing sharply after k=10. It was shown that a large decreases the expected number of packets marked *out* in the queue at the bottleneck. When congestion occurs, packets marked *out* get quickly exhausted for large values of k results large no. of *in* packet drops occur. If this situation will occur then the discriminator misclassifies the congestion loss as wireless loss.

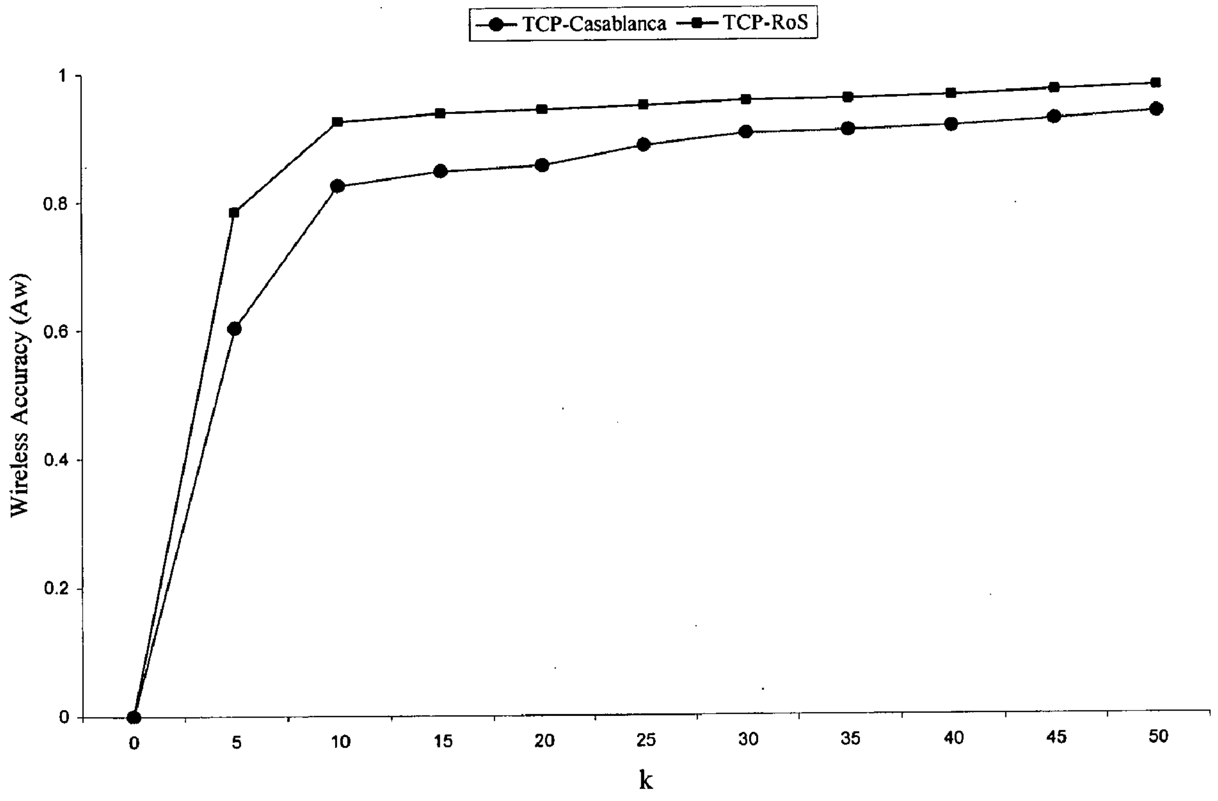

Fig 5.2: Wireless Accuracy A_w when p_w =0.01.

Simulations results on Fig. 5.2 shows that A_w increases as k increases. When k is large, packets marked *out* are rare and rarely get dropped on the wireless medium. Therefore, most losses appear to be random, leading to a high accuracy. Fig. 5.1 and 5.2 suggests that k should be chosen around 8 to achieve high values for and both Accuracies. For k=8, A_c is 0.997 and A_w is 0.88 for TCP-RoS, against A_c is 0.994 and A_w is 0.8 for TCP-Casablanca.

5.2 Throughput:

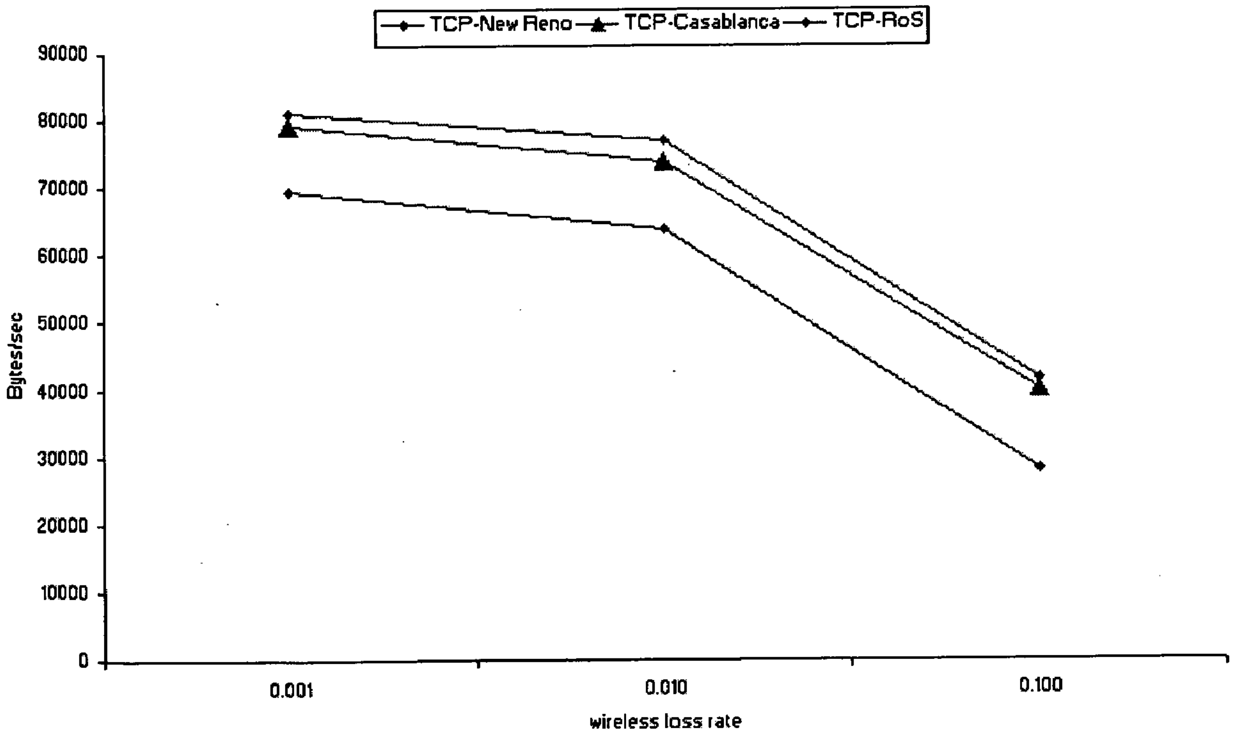

Fig 5.3: Throughput Vs wireless loss rate (P_w).

Simulations are carried out using wireless error rates (P_w) of 0.001, 0.01 and 0.1. Figure 5.3 shows the throughput achieved by TCP-RoS, TCP-Casablanca and TCP-New Reno in the presence of increasing error rates on the wireless link. Our results suggest that the proposed RoS algorithm identifies random losses on the wireless link. It outperforms TCP-New Reno and TCP-Casablanca in terms of throughput. TCP-RoS provides higher throughput than TCP-Casablanca and New Reno in all cases. According to the results, TCP-RoS has a higher throughput than TCP-Casablanca by 2.4% at P_w is 0.001. When P_w is 0.01 TCP-RoS has higher throughput than Casablanca by 4.21% and 3.8% higher throughput when P_w is 0.1.

From the graph we can observe that the throughput drops significantly for very high error rates. This can be explained as follows. The retransmitted packets are also dropped due to wireless transmission errors. When these retransmitted packets are dropped the congestion window of the sender decreases to one. As a result the throughput of TCP decreases.

Throughput in presence of non-priority flows:

Simulations are carried out in the presence of other flows (Flows that always sends *in* priority packets) to check fairness of our algorithm.

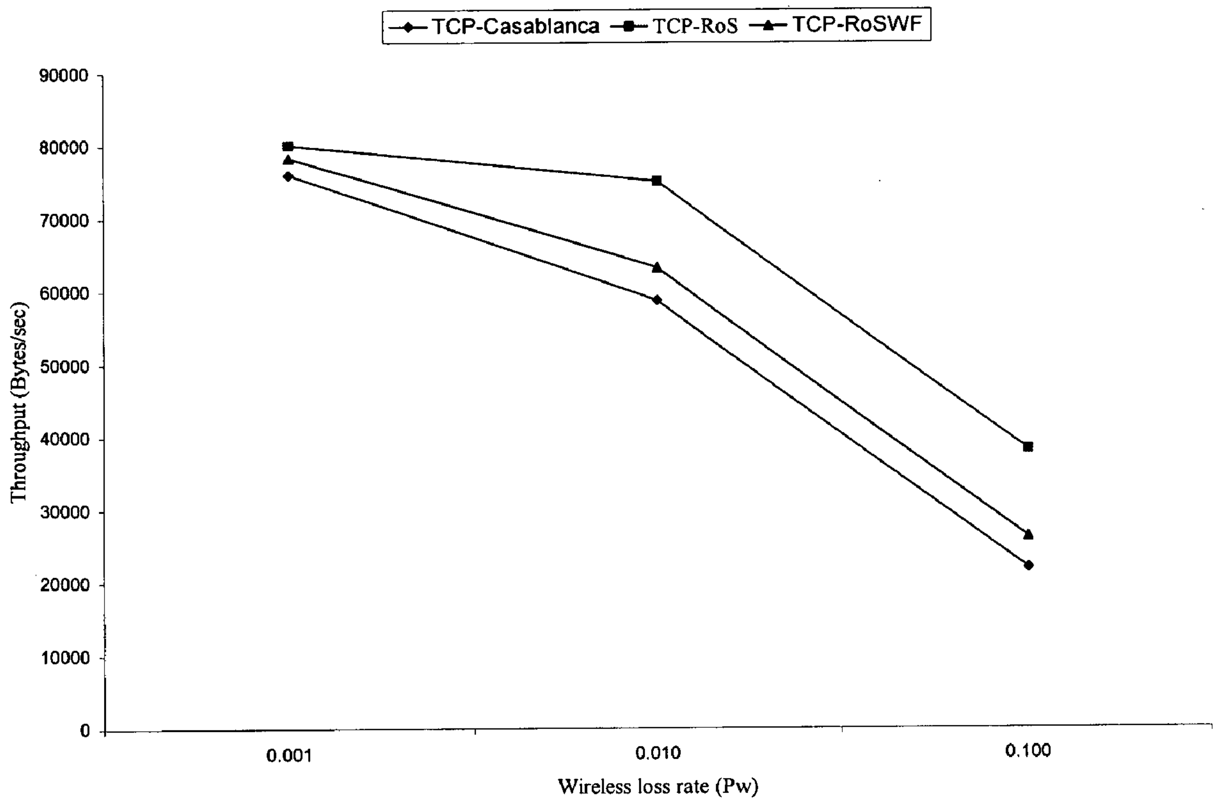

Fig 5.4: Throughput Vs wireless loss rate (p_w).

The biased queue management in TCP-Casablanca drops the packets only from flows that mark packets *out*. Only flows that mark packets *out* would be responsive to light congestion. Flows that do not mark packets *out* may monopolize the available link capacity. Due to this the throughput of the flows that mark the packet as *out* will degrade in the presence of other flows. Figure 5.4 shows the throughput achieved by TCP-RoS, TCP-Casablanca and TCP-RoSWF in the presence of increasing error rates on the wireless link. TCP-RoS throughput is high compare to TCP-Casablanca and TCP-RoSWF because the queue management drops the packet according to the incoming packet flow. TCP-RoS has a higher throughput than TCP-Casablanca by 5.2% when P_w is

0.001. When P_w is 0.01 TCP-RoS has a higher throughput than Casablanca by 27.6% and 74.4% higher throughput when P_w is 0.1. Hence we can conclude that our proposed TCP-RoS algorithm out performs TCP-Casablanca and TCP-RoSWF (TCP-RoS with out fairness) in presence of other flows.

6.1 Conclusion:

We developed a new discriminator TCP-RoS, which not only differentiate wireless and congestion losses but also identifies level of congestion in the network. We have shown through simulation that TCP-RoS is able to maintain high throughput in wireless error prone links than TCP-Casablanca. We solved the fairness problem which is in TCP-Casablanca.

6.2 Scope of future work:

Our future work will focus on discriminator to yield a better accuracy, a key factor in improving the performance of TCP in presence of random losses by using multiple dropping priorities. Multiple dropping priorities, used with a biased queue management, may well yield a higher wireless accuracy .Our further research to investigate the behavior of the proposed technique over different network topologies and asymmetric networks.

REFERENCES:

1. M. Allman, V. Paxson, and W. Stevens, "*TCP congestion control,*" IETF, RFC 2581, Apr. 1999.
2. D.Barman and I.Matta, "*Effectiveness of loss labeling in improving TCP performance in wired/wireless networks*" in Proc. 10th IEEE Int. Conf. Network Protocols (ICNP'2002), Paris, France, Nov. 2002, pp. 2–11.
3. L. S. Brakmo and S. O'Malley, "*TCP-Vegas: New techniques for congestion detection and avoidance*" in Proc. ACM SIGCOMM, Oct. 1994, pp. 24–35.
4. S. Cen, P. C. Cosman, and G. M. Voelker, "*End-to-end differentiation of congestion and wireless losses*" IEEE/ACM Trans. Networking, vol. 11, no. 5, Oct. 2003, pp. 703–717.
5. S. Floyd and T. Henderson, "*The NEWRENO modification to TCP's fast recovery algorithm*" IETF, RFC 2582, Apr. 1999.
6. HOE, J.C.:"*Improving the start-up behavior of a congestion control scheme for TCP*". Proceedings of SIGCOMM '96, 1996, (ACM), pp. 270-280.
7. JACOBSON, V.: "*Congestion avoidance and control*". Proceedings of SIGCOMM '88, 1988, (ACM), pp. 314-329.
8. Kostas Pentikousis, *TCP in Wired-Cum-Wireless Environments*, State University of New York at Stony Brook, IEEE Communications Surveys, Fourth Quarter 2000.
9. MATHIS, M., MAHDAVI, J., FLOYD, S., and ROMANOW, A.: '*TCP selective acknowledgment options*'. IETF, RFC 2018 (status-Proposed Standard), 1996.
10. MATHIS, M., and MAHDAVI, J.: "*Forward acknowledgment: refining TCP congestion control*". Proceedings of SIGCOMM '96, 1996, (ACM). pp. 281-291.
11. Saad Biaz and N.H. Vaidya, "*Discriminating congestion losses from wireless losses using inter-arrival times at the receiver,*" in Proc. IEEE Symp. Application-Specific Systems and Software Engineering and Technology (ASSET'99), Texas University, Mar. 1999, pp. 10–17.
12. Saad Biaz and N.H. Vaidya, "*"De-Randomizing" Congestion Losses to improve TCP performance over wired-wireless networks*", IEEE Trans. On Networking, June 2005, pp. 596-608.

13. SAMARAWEERA, N., and FAIRHURST, G.: "*Reinforcement of TCP error recovery for wireless communication*", Computer Communication (ACM), 1998, pp. 30-38.

14. SAMARAWEERA, N., and FAIRHURST, G.: "*Robust data link protocols for connection-less service over satellite links*", Int J. Satellite Communication, 1996, pp. 427-437.

15. SAMARAWEERA, N., and FAIRHURST, G.: "*Explicit loss indication and accurate RTO estimation for TCP error recovery using satellite links*", IEEE Proc., Communication. 1997, pp. 47-53.

16. STEVENS, W.R.: "*TCP/IP illustrated*" vol 1 (Addison Wesley, New York, 1994, 1st edn.).

17. Y. Tobe, Y. Tamura, A. Molano, S. Ghosh, and H. Tokuda, "*Achieving moderate fairness for UDP flows by path-status classification*" in Proc. 25th Annu. IEEE Conf. Local Computer Networks (LCN), Tampa, FL, Nov. 2000, pp. 252–261.

18. J. Waldby, U. Madhow, and T. Lakshman. "*Total acknowledgements:a robust feedback mechanism for end-to-end congestion control*". In Sigmetrics '98 Performance Evaluation Review, volume 26, 1998.

19. *ns-2* network simulator. [Online]. Available: http://www.isi.edu/nsnam/ns/.

20. J. F. Kurose and K. W. Ross, *Computer Networking,* Addison Wesley, New York, 2001

APPENDIX

A.1 NAM output and Trace file sample output:

Figure A.1.1 shows the sample NAM output.

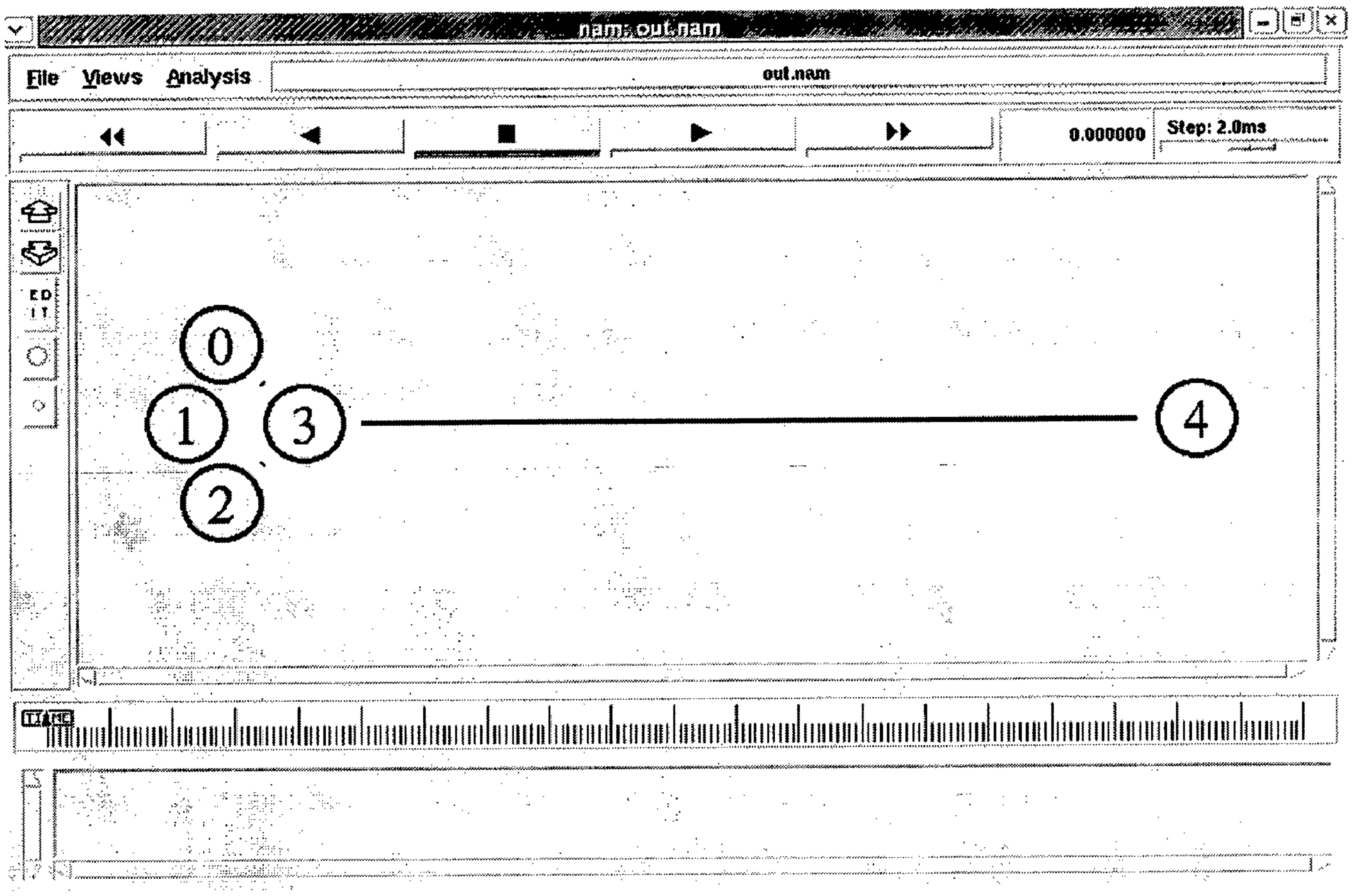

Figure A.1.1: Sample NAM output.

The following is the sample trace file generated by the NAM for TCP.

```
+ 1 0 3 tcp 40 ------- 2 0.0 4.0 0 0
- 1 0 3 tcp 40 ------- 2 0.0 4.0 0 0
r 1.00014 0 3 tcp 40 ------- 2 0.0 4.0 0 0
+ 1.00014 3 4 tcp 40 ------- 2 0.0 4.0 0 0
- 1.00014 3 4 tcp 40 ------- 2 0.0 4.0 0 0
r 1.10054 3 4 tcp 40 ------- 2 0.0 4.0 0 0
+ 1.10054 4 3 ack 40 ------- 2 4.0 0.0 0 1
- 1.10054 4 3 ack 40 ------- 2 4.0 0.0 0 1
+ 1.2 1 3 tcp 40 ------- 3 1.0 4.1 0 2
- 1.2 1 3 tcp 40 ------- 3 1.0 4.1 0 2
r 1.20014 1 3 tcp 40 ------- 3 1.0 4.1 0 2
+ 1.20014 3 4 tcp 40 ------- 3 1.0 4.1 0 2
- 1.20014 3 4 tcp 40 ------- 3 1.0 4.1 0 2
r 1.20094 4 3 ack 40 ------- 2 4.0 0.0 0 1
+ 1.20094 3 0 ack 40 ------- 2 4.0 0.0 0 1
- 1.20094 3 0 ack 40 ------- 2 4.0 0.0 0 1
r 1.20108 3 0 ack 40 ------- 2 4.0 0.0 0 1
+ 1.20108 0 3 tcp 1500 ------- 2 0.0 4.0 1 3
- 1.20108 0 3 tcp 1500 ------- 2 0.0 4.0 1 3
r 1.20268 0 3 tcp 1500 ------- 2 0.0 4.0 1 3
+ 1.20268 3 4 tcp 1500 ------- 2 0.0 4.0 1 3
```

- The first field is event type.

+	For enqueue
-	For dequeue
r	For receive

- Second column is simulation time at which each event occurred.
- The next two fields indicate between which two nodes tracing is happening
- The next field is a descriptive name for the type of packet seen (tcp, ack).
- The next field is the packet size as encoded in its IP header.
- The next four characters represent special flag bits, which may be enabled.
- The next field gives the IP flow identifier field as defined for IP version 6.
- The subsequent two fields indicate the packets source and destination node addresses respectively.

- The following field indicates the sequence number.
- The last field is a unique packet identifier. Each new packet created in the simulation is assigned a new unique identifier.

A.2 Source Code Listing

tcp.h:

The following class is added in tcp.h.

```
struct hdr_tcp {

#define NSA 3
        double ts_;     /* time packet generated (at source) */
        double ts_echo_;/* the echoed timestamp (originally sent by
                              the peer) */
        int seqno_;       /* sequence number */
///assigning priority to the packet for TCP-RoS.
    int priority_;   /*  prority for packet  */
      int eln_;        /*  loss notification in TCP-RoS*/
///////////////////////////////////////////
        int reason_;           /* reason for a retransmit */
        int sack_area_[NSA+1][2];/*sack blocks:start,end of block*/
        int sa_length_;  /* Indicate the number of SACKs in this  *
                    /* packet.  Adds 2+sack_length*8 bytes   */
        int ackno_;          /* ACK number for FullTcp */
        int hlen_;         /* header len (bytes) for FullTcp */
        int tcp_flags_;        /* TCP flags for FullTcp */
        int last_rtt_; /* more recent RTT measurement in ms, */
                              /*   for statistics only */
        static int offset_; // offset for this header
        inline static int& offset() { return offset_; }
        inline static hdr_tcp* access(Packet* p) {
                return (hdr_tcp*) p->access(offset_);
        }

        /* per-field member functions */
        double& ts() { return (ts_); }
        double& ts_echo() { return (ts_echo_); }
        int& seqno() { return (seqno_); }
       int& priority(){return (priority_);}
       int& eln(){return (eln_);}
        int& reason() { return (reason_); }
        int& sa_left(int n) { return (sack_area_[n][0]); }
        int& sa_right(int n) { return (sack_area_[n][1]); }
        int& sa_length() { return (sa_length_); }
```

```
                int& hlen() { return (hlen_); }
                int& ackno() { return (ackno_); }
                int& flags() { return (tcp_flags_); }
                int& last_rtt() { return (last_rtt_); }
        };


        class BqmTcpAgent : public virtual NewRenoTcpAgent {

         public:
            int cong_loss_notify;
                BqmTcpAgent();
                virtual void recv(Packet *pkt, Handler*);
                virtual void dupack_action();
                virtual void output(int seqno, int reason = 0);
                wireless_loss loss_type;
                };
```

tcpnewreno.cc

The following functions are added in tcpnewreno.cc:

```
static class BqmTcpClass : public TclClass {
        public:
                BqmTcpClass() : TclClass("Agent/TCP/Newreno/Bqm") {}
                TclObject* create(int, const char*const*) {
                        return (new BqmTcpAgent());
                }
} class_bqm;

BqmTcpAgent::BqmTcpAgent()
{
}
void
BqmTcpAgent::dupack_action()
{
            int recovered = (highest_ack_ > recover_);
                int recovered1 = (highest_ack_ == recover_);

                int allowFastRetransmit  =allow_fast_retransmit(last_cwnd_action_);
                if (recovered || (!bug_fix_ && !ecn_) || allowFastRetransmit) {
                   goto reno_action;
            }
                if (bug_fix_ && less_careful_ && recovered1) {
                        /*
```

```
          * For the Less Careful variant, allow a Fast Retransmit
                  *  if highest_ack_ == recover.
                  * RFC 2582 recommends the Careful variant, not the
                  *  Less Careful one.
                  */
           goto reno_action;
         }

     if (ecn_ && last_cwnd_action_ == CWND_ACTION_ECN) {
           last_cwnd_action_ = CWND_ACTION_DUPACK;
           /*
   *What if there is a DUPACK action followed closely by ECN
            * followed closely by a DUPACK action?
          * The optimal thing to do would be to remember all
            * congestion actions from the most recent window
            * of data.  Otherwise "bugfix" might not prevent
            * all unnecessary Fast Retransmits.
            */
           reset_rtx_timer(1,0);
           output(last_ack_ + 1, TCP_REASON_DUPACK);
                dupwnd_ = numdupacks_;
           return;
     }

     if (bug_fix_) {
                if (bugfix_ts_ && tss[highest_ack_ % tss_size_] == ts_echo_)
                       goto reno_action;
                else if (bugfix_ack_ && cwnd_ > 1 && highest_ack_ -
prev_highest_ack_ <= numdupacks_)
                       goto reno_action;
                else
           /*
            * The line below, for "bug_fix_" true, avoids
            * problems with multiple fast retransmits in one
            * window of data.
            */
                return;
     }

reno_action:
     recover_ = maxseq_;
     reset_rtx_timer(1,0);
     if (!lossQuickStart()) {
           trace_event("NEWRENO_FAST_RETX");
                last_cwnd_action_ = CWND_ACTION_DUPACK;
/////////////////////code written by Vijender//////////////////////////////////////////////
```

```
                if(loss_type==WIRELESS)
          ///if loss is due to wireless
                    slowdown(CWND_ACTION_WIRELESSERROR);
                else if(loss_type==LESS_CONGESTION)
            //if loss is due to congestion and network is lightly congested
                    slowdown(THREE_QUARTER_CWND);
              else

        slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_HALF);
                output(last_ack_ + 1, TCP_REASON_DUPACK);
                // from top
                dupwnd_ = numdupacks_;
          }
         return;
}
```

void BqmTcpAgent::output(int seqno, int reason)

```
{
          int force_set_rtx_timer = 0,k_=8;
          Packet* p = allocpkt();
          hdr_tcp *tcph = hdr_tcp::access(p);
          hdr_flags* hf = hdr_flags::access(p);
          hdr_ip *iph = hdr_ip::access(p);
          int databytes = hdr_cmn::access(p)->size();
          tcph->seqno() = seqno;
        tcph->ts() = Scheduler::instance().clock();

        // Mark packet for diagnosis purposes if we are in Quick-Start Phase
          if (qs_approved_) {
                hf->qs() = 1;
          }
        ////code written by vijender for TCP-RoS//////////
         if(seqno % k_ ==0)
          {
           tcph->priority_ = k_ + 64;
           hdr_cmn::access(p)->priority_tcp_=1;
          }
         else
          {
          tcph-> priority_=k_;
          hdr_cmn::access(p)->priority_tcp_=0;
          }
         ////////////////////////////////////////////////////////////

        // store timestamps, with bugfix_ts_.  From Andrei Gurtov.
        // (A real TCP would use scoreboard for this.)
     if (bugfix_ts_ && tss==NULL) {
          tss = (double*) calloc(tss_size_, sizeof(double));
          if (tss==NULL) exit(1);
     }
     //dynamically grow the timestamp array if it's getting full
     if (bugfix_ts_ && window() > tss_size_* 0.9) {
          double *ntss;
          ntss = (double*) calloc(tss_size_*2, sizeof(double));
          printf("resizing timestamp table\n");
          if (ntss == NULL) exit(1);
          for (int i=0; i<tss_size_; i++)
               ntss[(highest_ack_ + i) % (tss_size_ * 2)] =
                    tss[(highest_ack_ + i) % tss_size_];
          free(tss);
          tss_size_ *= 2;
          tss = ntss;
     }

     if (tss!=NULL)
          tss[seqno % tss_size_] = tcph->ts();

        tcph->ts_echo() = ts_peer_;
        tcph->reason() = reason;
        tcph->last_rtt() = int(int(t_rtt_)*tcp_tick_*1000);

        if (ecn_) {
                hf->ect() = 1;  // ECN-capable transport
        }
        if (cong_action_) {
                hf->cong_action() = TRUE;  // Congestion action.
                cong_action_ = FALSE;
     }
        /* Check if this is the initial SYN packet. */
        if (seqno == 0) {
                if (syn_) {
                        databytes = 0;
                        curseq_ += 1;
                        hdr_cmn::access(p)->size() = tcpip_base_hdr_size_;
                }
                if (ecn_) {
                        hf->ecnecho() = 1;
//                      hf->cong_action() = 1;
                        hf->ect() = 0;
                }
                if (qs_enabled_) {
                        hdr_qs *qsh = hdr_qs::access(p);

                        // dataout is kilobytes queued for sending
                        int dataout = (curseq_ - maxseq_ - 1) * (size_ +
headersize()) / 1024;
                        int qs_rr = rate_request_;
                        if (qs_request_mode_ == 1) {
                                // PS: Avoid making unnecessary QS requests
                                // use a rough estimation of RTT in qs_rtt_
                                // to calculate the desired rate from dataout.
                                if (dataout * 1000 / qs_rtt_ < qs_rr) {
                                        qs_rr = dataout * 1000 / qs_rtt_;
                                }
                                // qs_thresh_ is minimum number of unsent
                                // segments needed to activate QS request
                                if ((curseq_ - maxseq_ - 1) < qs_thresh_) {
                                        qs_rr = 0;
                                }
                        }

                        if (qs_rr > 0) {
                                // QuickStart code from Srikanth Sundarrajan.
                                qsh->flag() = QS_REQUEST;
                                Random::seed_heuristically();
                                qsh->ttl() = Random::integer(256);
                                ttl_diff_ = (iph->ttl() - qsh->ttl()) % 256;
                                qsh->rate() = hdr_qs::Bps_to_rate(qs_rr * 1024);
                                qs_requested_ = 1;
                        } else {
                                qsh->flag() = QS_DISABLE;
                        }
                }
        }
        else if (useHeaders_ == true) {
                hdr_cmn::access(p)->size() += headersize();
        }
    hdr_cmn::access(p)->size();

        /* if no outstanding data, be sure to set rtx timer again */
        if (highest_ack_ == maxseq_)
                force_set_rtx_timer = 1;
        /* call helper function to fill in additional fields */
        output_helper(p);

    ++ndatapack_;
    ndatabytes_ += databytes;
```

```
            send(p, 0);
            if (seqno == curseq_ && seqno > maxseq_)
                    idle(); // Tell application I have sent everything so far
            if (seqno > maxseq_) {
                    maxseq_ = seqno;
                    if (!rtt_active_) {
                            rtt_active_ = 1;
                            if (seqno > rtt_seq_) {
                                    rtt_seq_ = seqno;
                                    rtt_ts_ = Scheduler::instance().clock();
                            }

                    }
            } else {
            ++nrexmitpack_;
                    nrexmitbytes_ += databytes;
            }
            if (!(rtx_timer_.status() == TIMER_PENDING) || force_set_rtx_timer)
                    /* No timer pending. Schedule one. */
                    set_rtx_timer();
}
```

void BqmTcpAgent::recv(Packet *pkt, Handler*)

```
{
         hdr_tcp *tcph = hdr_tcp::access(pkt);
            int valid_ack = 0;
            static double min_rtt=0,max_rtt=9999999999.00;

            /* Use first packet to calculate the RTT  --contributed by Allman */

         if (qs_approved_ == 1 && tcph->seqno() > last_ack_)
                    endQuickStart();
         if (qs_requested_ == 1)
              processQuickStart(pkt);
            if (++acked_ == 1)
                    basertt_ = Scheduler::instance().clock() - firstsent_;

            /* Estimate ssthresh based on the calculated RTT and the estimated
              bandwidth (using ACKs 2 and 3).  */

            else if (acked_ == 2)
                    ack2_ = Scheduler::instance().clock();
            else if (acked_ == 3) {
                    ack3_ = Scheduler::instance().clock();
                    new_ssthresh_ = int((basertt_ * (size_ / (ack3_ - ack2_))) / size_);
```

```
                if (newreno_changes_ > 0 && new_ssthresh_ < ssthresh_)
                        ssthresh_ = new_ssthresh_;
        }

#ifdef notdef
        if (pkt->type_ != PT_ACK) {
                fprintf(stderr,
                        "ns: confiuration error: tcp received non-ack\n");
                exit(1);
        }
#endif
     /* W.N.: check if this is from a previous incarnation */
     if (tcph->ts() < lastreset_) {
          // Remove packet and do nothing
          Packet::free(pkt);
          return;
     }
        ++nackpack_;
        ts_peer_ = tcph->ts();

        if (hdr_flags::access(pkt)->ecnecho() && ecn_)
                ecn(tcph->seqno());
        recv_helper(pkt);
        recv_frto_helper(pkt);
        if (tcph->seqno() > last_ack_) {
                if (tcph->seqno() >= recover_
                  || (last_cwnd_action_ != CWND_ACTION_DUPACK)) {
                        if (dupwnd_ > 0) {
                           dupwnd_ = 0;
                           if (last_cwnd_action_ == CWND_ACTION_DUPACK)
                                last_cwnd_action_ = CWND_ACTION_EXITED;
                           if (exit_recovery_fix_) {
                                int outstanding = maxseq_ - tcph->seqno() + 1;
                                if (ssthresh_ < outstanding)
                          cwnd_ = ssthresh_;
                    else
                          cwnd_ = outstanding;
                           }
                        }
                        firstpartial_ = 0;
                        recv_newack_helper(pkt);
                        if (last_ack_ == 0 && delay_growth_) {
                                cwnd_ = initial_window();
                        }
                } else {
                        /* received new ack for a packet sent during Fast
                    * Recovery, but sender stays in Fast Recovery */
                    if (partial_window_deflation_ == 0)
                            dupwnd_ = 0;
                    partialnewack_helper(pkt);
            }
    } else if (tcph->seqno() == last_ack_) {
            if (hdr_flags::access(pkt)->eln_ && eln_) {
                    tcp_eln(pkt);
                    return;
            }
            if (++dupacks_ == numdupacks_) {
//////////////////////////////////////////////////////////////////////////////////////////////////////
    if(tcph->eln_==1)
       {
            printf("wireless loss\n");
       loss_type=WIRELESS;
         }

     else
      {

        if( t_rtt_ <= (min_rtt+max_rtt)*0.5)
           /// checking rtt whether network is lightly congested or not
             loss_type=LESS_CONGESTION;
         else
           loss_type=CONGESTION;
          printf("congestion loss\n");
        }
                    dupack_action();

            if (!exitFastRetrans_)
                    dupwnd_ = numdupacks_;

       }
            else if (dupacks_ > numdupacks_ && (!exitFastRetrans_
                || last_cwnd_action_ == CWND_ACTION_DUPACK)) {
                    trace_event("NEWRENO_FAST_RECOVERY");
                    ++dupwnd_;  // fast recovery

                    /* For every two duplicate ACKs we receive (in the
                     * "fast retransmit phase"), send one entirely new
                     * data packet "to keep the flywheel going".  --Allman
                     */
                    if (newreno_changes_ > 0 && (dupacks_ % 2) == 1)
                            output (t_seqno_++,0);
            } else if (dupacks_ < numdupacks_ && singledup_ ) {
```

```
                        send_one();
            }
        }


    if (tcph->seqno() >= last_ack_)
        // Check if ACK is valid.  Suggestion by Mark Allman.
        valid_ack = 1;
      if(min_rtt==0)
        min_rtt=t_rtt_;
      if(min_rtt  > t_rtt_)
        min_rtt=t_rtt_;
      if(max_rtt==9999999999.00)
       max_rtt=t_rtt_;
      if(max_rtt < t_rtt_)
       max_rtt=t_rtt_;

      Packet::free(pkt);
#ifdef notyet
      if (trace_)
              plot();
#endif

      /*
       * Try to send more data
       */

    if (valid_ack || aggressive_maxburst_)
              if (dupacks_ == 0)
                      /*
                       * Maxburst is really only needed for the first
                       *  window of data on exiting Fast Recovery.
                       */
                      send_much(0, 0, maxburst_);
              else if (dupacks_ > numdupacks_ - 1 && newreno_changes_ == 0)
                      send_much(0, 0, 2);
}
```

tcpsink.cc

The following functions are modified in tcpsink.cc.

void TcpSink::ack(Packet* opkt)

```
{
            Packet* npkt = allocpkt();
            // opkt is the "old" packet that was received
            // npkt is the "new" packet being constructed (for the ACK)
            double now = Scheduler::instance().clock();
            hdr_flags *sf;

            hdr_tcp *otcp = hdr_tcp::access(opkt);
            hdr_ip *oiph = hdr_ip::access(opkt);
            hdr_tcp *ntcp = hdr_tcp::access(npkt);
          //for tcp-casablanca
           if(prevpktno_==ntcp->seqno())
                no_dup_++;
           else
              prevpktno_=ntcp->seqno();
           //if(no_dup_==3&&(F_>=0))
        ///assigning eln flag for TCP-RoS
            if(Fflag_==1)
            {
             Fflag_=0;
             if( F_ == 1)
               ntcp->eln_=1;
             else if(F_ == 2)
               ntcp->eln_=2;
              else
               ntcp->eln_=0;
              }
            /*if(F_>=0)
             {
              printf("ack marked as wireless\n");
                ntcp->eln_=1;
             }
            else
              {
               printf("ack marked as congestion\n");
               ntcp->eln_=0;
              }
             }*/
            ////////////////////////////////
```

```
if (qs_enabled_) {
        // QuickStart code from Srikanth Sundarrajan.
        hdr_qs *oqsh = hdr_qs::access(opkt);
        hdr_qs *nqsh = hdr_qs::access(npkt);
     if (otcp->seqno() == 0 && oqsh->flag() == QS_REQUEST) {
          nqsh->flag() = QS_RESPONSE;
          nqsh->ttl() = (oiph->ttl() - oqsh->ttl()) % 256;
          nqsh->rate() = (oqsh->rate() < MWS) ? oqsh->rate() : MWS;
     }
     else {
          nqsh->flag() = QS_DISABLE;
     }
}


// get the tcp headers
ntcp->seqno() = acker_->Seqno();
// get the cumulative sequence number to put in the ACK; this
// is just the left edge of the receive window - 1
ntcp->ts() = now;
// timestamp the packet

if (ts_echo_bugfix_)  /* TCP/IP Illustrated, Vol. 2, pg. 870 */
        ntcp->ts_echo() = acker_->ts_to_echo();
else
        ntcp->ts_echo() = otcp->ts();
// echo the original's time stamp

hdr_ip* oip = hdr_ip::access(opkt);
hdr_ip* nip = hdr_ip::access(npkt);
// get the ip headers
nip->flowid() = oip->flowid();
// copy the flow id

hdr_flags* of = hdr_flags::access(opkt);
hdr_flags* nf = hdr_flags::access(npkt);
if (save_ != NULL)
        sf = hdr_flags::access(save_);
        // Look at delayed packet being acked.
if ( (save_ != NULL && sf->cong_action()) || of->cong_action() )
        // Sender has responsed to congestion.
        acker_->update_ecn_unacked(0);
if ( (save_ != NULL && sf->ect() && sf->ce())  ||
                (of->ect() && of->ce()) )
                // New report of congestion.
                acker_->update_ecn_unacked(1);
        if ( (save_ != NULL && sf->ect()) || of->ect() )
                // Set EcnEcho bit.
                nf->ecnecho() = acker_->ecn_unacked();
        if (!of->ect() && of->ecnecho() ||
                (save_ != NULL && !sf->ect() && sf->ecnecho()) )
                 // This is the negotiation for ECN-capability.
                 // We are not checking for of->cong_action() also.
                 // In this respect, this does not conform to the
                 // specifications in the internet draft
                nf->ecnecho() = 1;
        acker_->append_ack(hdr_cmn::access(npkt),
                           ntcp, otcp->seqno());
        add_to_ack(npkt);
        // the above function is used in TcpAsymSink

      // Andrei Gurtov
      acker_->last_ack_sent_ = ntcp->seqno();
      // printf("ACK %d ts %f\n", ntcp->seqno(), ntcp->ts_echo());

        send(npkt, 0);
        // send it
}
```

```
void TcpSink::recv(Packet* pkt, Handler*)
{
        static double prev_time=Scheduler::instance().clock();
        int numToDeliver;
         float x=0,r=0;
        int numBytes = hdr_cmn::access(pkt)->size();
        double cur_time;
        double interarrivaltime;
        static double min_inter_time=0,max_inter_time=9999999999.00;
        // number of bytes in the packet just received
        hdr_tcp *th = hdr_tcp::access(pkt);
       int k=th->priority_ % 64;
         /* W.N. Check if packet is from previous incarnation */
        if (th->ts() < lastreset_) {
                // Remove packet and do nothing
                Packet::free(pkt);
                return;
        }
        acker_->update_ts(th->seqno(),th->ts(),ts_echo_rfc1323_);
        // update the timestamp to echo
```

```
numToDeliver = acker_->update(th->seqno(), numBytes);
// update the recv window; figure out how many in-order-bytes
// (if any) can be removed from the window and handed to the
// application
if (numToDeliver) {
     cur_time=Scheduler::instance().clock();
     interarrivaltime= cur_time-prev_time;
         bytes_ += numToDeliver;
         recvBytes(numToDeliver);
}

////code for setting minimum and max rtts for TCP-RoS
    if(min_inter_time==0)
      min_inter_time=interarrivaltime;
    if(min_inter_time  > interarrivaltime)
      min_inter_time=interarrivaltime;
    if(max_inter_time==9999999999.00)
      max_inter_time=interarrivaltime;
    if(max_inter_time < interarrivaltime)
      max_inter_time=interarrivaltime;
    // send any packets to the application

 // Caliculating whether the loss is due to congestion or wireless

   if(numToDeliver==0)
   {
       int k_temp=0;
       for(int i=acker_->Seqno()+1;i<acker_->Maxseen();i++)
     {
     if(acker_->seen_[i&acker_->wndmask_]==0)
       {
              printf("packet :%d:lost,%d\n",i,k_temp++);
         if(i%k==0)
              {
               printf("Packet is out\n");
            x++;
              }
         r++;
       }
     }
       printf("x,r=%f,%f\n",x,r);
        k_temp=0;
   }
      if(r==0)
       F_ = -1;
      else
```

```
                {
                 if(r-x==r)
                   F_ = 1;
               //else if(r-x < r&&r-x>=r/2)
                 //  F_ = 2;
                 else
                   F_ = 0;


              //F_ = 1.0 - (int) ( k * (x/r) );
                Fflag_=1;
                }
                      printf("F_=%f\n",F_);
              //Fflag_=1;
              //////////////////////////////////////////
            ack(pkt);
              // ACK the packet
              Packet::free(pkt);
              // remove it from the system
}
```

droptail.cc

The following functions is modified in droptail.cc

```
void DropTail::enque(Packet* p)
{
          //char  tracer[]="pkt drop\n";
              if (summarystats) {
                Queue::updateStats(qib_?q_->byteLength():q_->length());
              }

              int qlimBytes = qlim_ * mean_pktsize_;
              if ((!qib_ && (q_->length() + 1) >= qlim_) ||
              (qib_ && (q_->byteLength() + hdr_cmn::access(p)->size()) >=
     qlimBytes)){
                       // if the queue would overflow if we added this packet...
                        //code here for biased queue management
                         q_->enque(p);
               Packet *delpack=q_->dequebqm();
              // printf("drp= %d\n",hdr_cmn::access(delpack)->priority_tcp());
                drop(delpack);

              } else {
                    q_->enque(p);
              }}
```

queue.cc

The following function is added in queue.cc

Packet* PacketQueue::dequebqm()

```
{
            Packet *k,*pp=0;
            int prio=0;
              ns_addr_t source;
              ns_addr_t dist;
            for(Packet *pk=head_;pk;k=pk,pk=pk->next_);
              source=hdr_ip::access(k)->src();
                dist=hdr_ip::access(k)->dst();
            for (Packet *p= head_; p;k=pp, pp= p, p= p->next_) {
        ///checking the source and destination addresses
                if(source==hdr_ip::access(p)->src()&&dist==hdr_ip::access(p)->dst())
                {
                  prio=hdr_cmn::access(p)->priority_tcp();

                 if(prio==1)
                 {
                   if(p==head_)
                     head_=p->next_;
                   else
                     {
                     pp->next_=p->next_;
                     if(p==tail_)
                       tail_=pp;
                     }
                    //remove(p);
                    --len_;
                    bytes_ -= hdr_cmn::access(p)->size();
                           printf("packet loss\n");
                          return p;
                   }

              }
              }
              remove(pp);
            --len_;
            bytes_ -= hdr_cmn::access(pp)->size();
            tail_=k;
              printf("packet loss \n");
            return pp;
}
```

tcpwireless.tcl

This tcl file used to create the topology shown in Fig 4.1.

```
set opt(chan)        Channel/WirelessChannel   ;# channel type
set opt(prop)        Propagation/TwoRayGround  ;# radio-propagation model
set opt(netif)       Phy/WirelessPhy         ;# network interface type
set opt(mac)         Mac/802_11              ;# MAC type
set opt(ifq)         Queue/DropTail          ;# interface queue type
set opt(ll)          LL                  ;# link layer type
set opt(ant)         Antenna/OmniAntenna      ;# antenna model
set opt(ifqlen)      50                  ;# max packet in ifq
set opt(nn)          3                   ;# number of mobilenodes
set opt(adhocRouting)  DSDV                  ;# routing protocol

set opt(cp)          ""                  ;# connection pattern file
set opt(sc)    "/home/vijay/ns/ns-allinone-2.28/ns-2.28/tcl/mobility/scene/scen-3-test"
;# node movement file.

set opt(x)     670                    ;# x coordinate of topology
set opt(y)     670                    ;# y coordinate of topology
set opt(seed)  0.0                   ;# seed for random number gen.
set opt(stop)  300                   ;# time to stop simulation

set opt(ftp1-start)     160
set opt(ftp2-start)     165

set num_wired_nodes     5
set num_bs_nodes        1

#===============================================================
==============
# check for boundary parameters and random seed
if { $opt(x) == 0 || $opt(y) == 0 } {
        puts "No X-Y boundary values given for wireless topology\n"
}
if {$opt(seed) > 0} {
        puts "Seeding Random number generator with $opt(seed)\n"
        ns-random $opt(seed)
}

# create simulator instance
set ns_   [new Simulator]
```

```
# set up for hierarchical routing
$ns_ node-config -addressType hierarchical
AddrParams set domain_num_ 2          ;# number of domains
lappend cluster_num 1 1               ;# number of clusters in each domain
AddrParams set cluster_num_ $cluster_num
lappend eilastlevel 5 4               ;# number of nodes in each cluster
AddrParams set nodes_num_ $eilastlevel ;# of each domain

set tracefd  [open wireless2-out.tr w]
set namtrace [open wireless2-out.nam w]
set cwnd [open cwnd-out.nam w]
$ns_ trace-all $tracefd
$ns_ namtrace-all-wireless $namtrace $opt(x) $opt(y)

# Create topography object
set topo   [new Topography]
#error model
set em [new ErrorModel]
 $em set unit pkt
 $em  set rate_ 0.01
 $em ranvar [new RandomVariable/Exponential]
 $em drop-target [new Agent/Null]

# define topology
$topo load_flatgrid $opt(x) $opt(y)

# create God
create-god [expr $opt(nn) + $num_bs_nodes]

#create wired nodes
set temp {0.0.0 0.0.1 0.0.2 0.0.3 0.0.4}
     ;# hierarchical addresses for wired domain
for {set i 0} {$i < $num_wired_nodes} {incr i} {
   set W($i) [$ns_ node [lindex $temp $i]]
}

# configure for base-station node
$ns_ node-config -adhocRouting $opt(adhocRouting) \
              -llType $opt(ll) \
              -macType $opt(mac) \
              -ifqType $opt(ifq) \
              -ifqLen $opt(ifqlen) \
              -antType $opt(ant) \
              -propType $opt(prop) \
              -phyType $opt(netif) \
              -channelType $opt(chan) \
                -topoInstance $topo \
            -wiredRouting ON \
                -agentTrace ON \
            -routerTrace OFF \
            -macTrace OFF

#create base-station node
set temp {1.0.0 1.0.1 1.0.2 1.0.3}   ;# hier address to be used for wireless
                              ;# domain
set BS(0) [$ns_ node [lindex $temp 0]]
$BS(0) random-motion 0            ;# disable random motion

#provide some co-ord (fixed) to base station node
$BS(0) set X_ 1.0
$BS(0) set Y_ 2.0
$BS(0) set Z_ 0.0

# create mobilenodes in the same domain as BS(0)
# note the position and movement of mobilenodes is as defined
# in $opt(sc)

#configure for mobilenodes
$ns_ node-config -wiredRouting OFF

  for {set j 0} {$j < $opt(nn)} {incr j} {
    set node_($j) [ $ns_ node [lindex $temp \
            [expr $j+1]] ]
    $node_($j) base-station [AddrParams addr2id \
            [$BS(0) node-addr]]
}

#create links between wired and BS nodes

$ns_ duplex-link $W(0) $W(1) 60Mb 5ms DropTail
$ns_ duplex-link $W(1) $W(2) 90Mb 5ms DropTail
#this link will change for congestion control
$ns_ duplex-link $W(2) $W(3) 45Mb 5ms DropTail
$ns_ duplex-link $W(3) $W(4) 100Mb 5ms DropTail

$ns_ duplex-link $W(4) $BS(0) 100Mb 5ms DropTail
$ns_ lossmodel $em $W(4) $BS(0)
$ns_ duplex-link-op $W(0) $W(1) orient right
$ns_ duplex-link-op $W(1) $W(2) orient right
$ns_ duplex-link-op $W(2) $W(3) orient right
$ns_ duplex-link-op $W(3) $W(4) orient right
$ns_ duplex-link-op $W(4) $BS(0) orient right-down
```

```
$ns_ queue-limit $W(0) $W(1) 30
$ns_ queue-limit $W(1) $W(2) 30
$ns_ queue-limit $W(2) $W(3) 30
$ns_ queue-limit $W(3) $W(4) 30
# setup TCP connections
## setup TCP connections N2
for {set j 0} {$j < 5} {incr j} {
set tcp(j) [new Agent/TCP/Reno]
set sink(j) [new Agent/TCPSink/DelAck]
$ns_ attach-agent $W(1) $tcp(j)
$ns_ attach-agent $W(4) $sink(j)
$ns_ connect $tcp(j) $sink(j)
set ftp(j) [new Application/FTP]
$ftp(j) attach-agent $tcp(j)
$ns_ at $opt(ftp1-start) "$ftp(j) start"
}
#for {set j 0} {$j < 5} {incr j} {
#set tcp(j) [new Agent/UDP]
#set sink(j) [new Agent/Null]
#$ns_ attach-agent $W(1) $tcp(j)
#$ns_ attach-agent $W(4) $sink(j)
#$ns_ connect $tcp(j) $sink(j)
#set ftp(j) [new Application/FTP]
#$ftp(j) set packetSize_ 1000
#$ftp(j) set interval_ 0.005
#$ftp(j) attach-agent $tcp(j)
#$ns_ at $opt(ftp1-start) "$ftp(j) start"
#}
#set up TCP connections N3
for {set i $j} {$i < 5+$j} {incr i} {
set tcp(i) [new Agent/TCP/Reno]
set sink(i) [new Agent/TCPSink/DelAck]
$ns_ attach-agent $W(2) $tcp(i)
$ns_ attach-agent $W(3) $sink(i)
$ns_ connect $tcp(i) $sink(i)
set ftp(i) [new Application/FTP]
$ftp(i) attach-agent $tcp(i)
$ns_ at $opt(ftp2-start) "$ftp(i) start"
}
#set up TCP connections N3
#for {set i $j} {$i < 5+$j} {incr i} {
#set tcp(i) [new Agent/UDP]
#set sink(i) [new Agent/Null]
#$ns_ attach-agent $W(2) $tcp(i)
#$ns_ attach-agent $W(3) $sink(i)
#$ns_ connect $tcp(i) $sink(i)
```

```
#set ftp(i) [new Application/FTP]
#$ftp(i) set packetSize_ 1000
#$ftp(i) set interval_ 0.005
#$ftp(i) attach-agent $tcp(i)
#$ns_ at $opt(ftp2-start) "$ftp(i) start"
#}
#set up TCP connections N1
#set tcp50 [new Agent/TCP]
#set sink50 [new Agent/TCPSink]
#$ns_ attach-agent $node_(0) $tcp50
#$ns_ attach-agent $W(0) $sink50
#$ns_ connect $tcp50 $sink50
#$ns_ add-agent-trace $tcp50 $cwnd
#$ns_ monitor-agent-trace  $tcp50
#$tcp50 tracevar cwnd_
#set ftp50 [new Application/FTP]
#$tcp50 set class_ 1
#$ftp50 attach-agent $tcp50
#$ns_ at $opt(ftp1-start) "$ftp50 start"

set tcp51 [new Agent/TCP/Newreno]
set sink51 [new Agent/TCPSink]
$ns_ attach-agent $W(0) $tcp51
$ns_ add-agent-trace $tcp51 $cwnd
$ns_ monitor-agent-trace  $tcp51
$tcp51 tracevar cwnd_
$ns_ attach-agent $node_(1) $sink51
$ns_ connect $tcp51 $sink51
set ftp51 [new Application/FTP]
$tcp51 set class_ 2
$ftp51 attach-agent $tcp51

$ns_ at $opt(ftp1-start) "$ftp51 start"
#$ns_ at $opt(ftp1-start) "$ftp51 send 20000000"
# source connection-pattern and node-movement scripts
if { $opt(cp) == "" } {
        puts "*** NOTE: no connection pattern specified."
     set opt(cp) "none"
} else {
        puts "Loading connection pattern..."
        source $opt(cp)
}
if { $opt(sc) == "" } {
        puts "*** NOTE: no scenario file specified."
     set opt(sc) "none"
} else {
        puts "Loading scenario file..."
        source $opt(sc)
        puts "Load complete..."
}


# Define initial node position in nam

for {set i 0} {$i < $opt(nn)} {incr i} {

   # 20 defines the node size in nam, must adjust it according to your
   # scenario
   # The function must be called after mobility model is defined

   $ns_ initial_node_pos $node_($i) 20
}

# Tell all nodes when the simulation ends
for {set i } {$i < $opt(nn) } {incr i} {
   $ns_ at $opt(stop).0 "$node_($i) reset";
}
$ns_ at $opt(stop).0 "$BS(0) reset";

$ns_ at $opt(stop).0002 "puts \"NS EXITING...\" ; $ns_ halt"
$ns_ at $opt(stop).0001 "stop"
proc stop {} {
   global ns_ tracefd namtrace  cwnd
#   $ns_ flush-trace
   close $tracefd
   close $namtrace
   close $cwnd
   exec nam wireless2-out.nam &
    exit 0
}

# informative headers for CMUTracefile
puts $tracefd "M 0.0 nn $opt(nn) x $opt(x) y $opt(y) rp \
        $opt(adhocRouting)"
puts $tracefd "M 0.0 sc $opt(sc) cp $opt(cp) seed $opt(seed)"
puts $tracefd "M 0.0 prop $opt(prop) ant $opt(ant)"

puts "Starting Simulation..."
$ns_ run
```