TCP k-SACK: A Simple Protocol to Improve Performance over Lossy Links

Abhay Chrungoo, Vishu Gupta, Huzur Saran and Rajeev Shorey
IBM India Research Laboratory,
Block 1, Indian Institute of Technology,
Hauz Khas, New Delhi 110016, India.

Abstract— We propose k-SACK, a TCP variant that has considerably improved throughput characteristics over lossy links. A k-SACK source does not consider every packet loss as an indication of congestion. It uses the Selective Acknowledgement option to estimate a parameter lookahead-loss, which is used in congestion detection. Our results demonstrate that k-SACK maintains a steady performance in a non-lossy environment while showing considerable performance improvement over paths spanning lossy links. Throughput gains of more than 90% are observed over links with packet loss rates of the order of 5%. We show that when k-SACK is used over wireline internet links, it does not harm the existing TCP variants significantly. k-SACK is simple to implement and does not involve any additional overheads.

Keywords— TCP, Selective Acknowledgements (SACK), Wireless networks, Lossy links, Bandwidth utilization, Throughput, Fairness, Drop tail, Congestion

I. Introduction

TCP [1] interprets a packet loss in the network as an indication of congestion. Over lossy links, packet losses are random and not due to congestion in the network. Due to this, TCP performance is good over wireline links but degrades over lossy links [5] such as wireless and satellite channels. Wireless links are often characterized by high bit error rates due to channel fading, noise, interference, and intermittent connectivity due to handoffs. TCP performance in such networks suffers from throughput degradation and very high interactive delays, because the sender misinterprets random packet losses as congestion. There have been studies on the performance of various versions of TCP over internet links [1] as well as lossy links. Various mechanisms have been put forward to alleviate the effects of non congestive losses on TCP performance over lossy links. These mechanisms adopt a variety of schemes such as Forward Error Correction (FEC), split TCP connections [3], snoop protocols [4], local retransmissions, link aware protocols, Explicit Loss Notification (ELN) [8]. However, it is desirable to develop an efficient TCP variant that works well over lossy as well as non-lossy wireline links.

In this paper, we propose k-SACK, a TCP variant with a simple modification to the TCP SACK congestion detection mechanism, whereby the sender does not interpret every packet loss as a sign of congestion. The proposed modification results in throughput improvement over lossy links, while maintaining a comparable performance over normal wireline links. We have compared and contrasted k-SACK with the SACK implementation over TCP New Reno (which we refer to as SACK over New Reno) and the SACK implementation over Reno (which we refer to as SACK over Reno) using simulations. The proposed k-SACK does not exhibit any performance improvement or degradation over wireline links, but increasingly better performance with increasing loss rates in wireless links.

The rest of the paper is organized as follows. Section II outlines the proposed k-SACK modification to the existing TCP congestion detection and avoidance mechanism. Section III discusses the rationale behind the modification and the expected improvement. Extensive simulations to understand the behavior of TCP k-SACK are described in Section IV. Section V has the conclusion.

II. The k-SACK Protocol

k-SACK is a new implementation of the TCP protocol and is modeled very closely to TCP SACK over New Reno. k-SACK is only a sender side modification and fully inter-operable with any SACK capable TCP receiver. k-SACK differs from SACK over New Reno in only a small modification to its congestion detection and avoidance algorithms.

Congestion Detection: k-SACK interprets congestion only if it detects k losses within a loss-window *(lwnd),* where k and *lwnd* are appropriately chosen parameters of the protocol. The notion can be better stated as follows: if out of *lwnd* number of most recently transmitted packets, k or more are lost, then a TCP k-SACK sender assumes congestion, otherwise, the loss is assumed to be due to random errors.

The cumulative acknowledgement in a TCP header acknowledges all packets up to the cumulative acknowledgement. If we can devise a mechanism to measure the number of losses between the cumulative acknowledgement and the highest sequence number transmitted, we can implement the described congestion detection mechanism. To measure this, we define the notion of a lookahead-loss.

Lookahead-loss: Lookahead-loss is equal to the number of unacknowledged segments transmitted by the sender, such that for all such segments, either of the two holds: (i) Sender has received at least *max-dupacks* (typically three) duplicate cumulative acknowledgements for the segment, (ii) The sender has received selective acknowledgements for at least *max-dupsacks* (analogously taken to be three) segments with higher sequence numbers.

All segments contributing to lookahead-loss are marked lost. We extend the mechanism of detecting loss through a number of duplicate acknowledgements to detecting loss through a number of selective acknowledgements beyond a segment that has

\ Abhay Chrungoo is a graduate student in UCLA, USA, Vishu Gupta is a graduate student in UIUC, USA, Dr. Huzur Saran is a Professor in the Department of Computer Science and Engineering at I.I.T, New Delhi, India

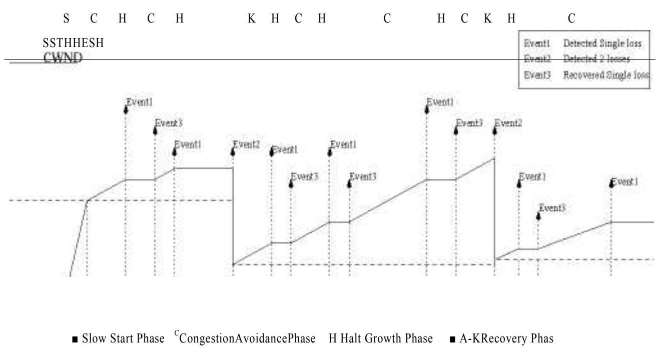


Fig. 1. An Example of congestion window evolution.

neither been acknowledged nor selectively acknowledged. This definition of lookahead-loss estimates the number of losses in the outstanding packets. The notion can be easily extended to detect the number of losses in any given *lwnd.* However, since all packets up to the cumulative acknowledgement have been successfully received, it is simplest and also most meaningful to estimate the lookahead-loss for a *lwnd* equal to the number of outstanding packets. The parameter k can also be chosen in a number of ways. However a large value of k increases the time it takes a TCP sender to detect and respond to congestion. Moreover high values are also likely to have an adverse effect on competing TCP connections in the internet. We have therefore chosen k equal to two for our current implementation. Behavior of the mechanism with a higher value of k is a subject of further research.

Note that since k-SACK is a close variation of TCP SACK over New Reno, all the mechanisms such as slow start, congestion avoidance, fast retransmit and fast recovery are common to them.

SACK over New Reno: SACK over New Reno works at the receiver side by sending selective acknowledgements for the segments received by it according to algorithms mentioned in [2].

The use of SACK over New Reno allows the sender to recover from multiple packet losses in a single round trip time. k-SACK uses the existing mechanisms [2], with a modified fast recovery algorithm. The essential features of fast recovery are preserved, with modification only to the *cwnd* and *ssthresh* behavior. The k-SACK protocol implements end-to-end congestion control using its concept of lookahead-loss and congestion detection as already discussed. The *cwnd* evolution for a TCP k-SACK sender is shown in Figure 1.

k-SACK Fast Recovery: In this phase, k-SACK performs fast recovery on the same lines as in New Reno [6], but with a modification in the *cwnd* and *ssthresh* dynamics. The *cwnd* and *ssthresh* behavior is determined based on the value of the lookahead-loss. For the purpose of comprehension, the fast recovery phase is separated into two different phases. On entering the fast recovery phase, if the lookahead-loss is less than k, the sender enters the halt growth phase of fast recovery, else it moves to the k-recovery phase of fast recovery.

(i) halt growth Phase: When in this state, the sender continues in fast recovery, but does not change the *cwnd* and *ssthresh* values. It freezes the congestion window and maintains it in that state until the lookahead-loss becomes at least k, whereupon the sender enters the k-recovery phase. Congestion window growth is unfrozen on moving to k-recovery Phase. After receiving an acknowledgement for up to or beyond *recover,* this phase is exited and the growth of the congestion window unfrozen. *(ii) k-recovery Phase:* A TCP sender enters this phase while in fast recovery, if the lookahead-loss is at least k. Upon entering this phase for the first time while in fast recovery, the *ssthresh* is set to , *2MSS),* and the congestion window is set to *ssthresh.* When lookahead-loss becomes less than k, the sender changes state to the halt growth phase. The sender may re-enter the k-recovery phase after making a transition to the halt growth phase, while within the same fast recovery. On such a re-entry, *cwnd* is set to *max(cwnd/2,1MSS)* and *ssthresh* is not changed. The sender continues in this phase until either the lookahead-loss is at least k or the completion of fast recovery.

Post Recovery: After the end of a New Reno fast recovery phase, the TCP sender continues in congestion avoidance. However, a k-SACK TCP sender may continue in the slow start phase depending on the *cwnd* and *ssthresh* values.

Pipe Estimation: k-SACK uses the selective acknowledgement information for more accurate pipe estimation and measures the pipe as the number of unacknowledged segments which are either not marked *lost* or marked *lost* and retransmitted. The previous versions use the *cwnd* size and the number of duplicate acknowledgements received, to determine the number of packets in the link *(pipe),* thus failing to subtract subsequent segment losses and acknowledgement losses from the pipe estimation.

Retransmissions: Other than the fast retransmit, segments that are marked lost are retransmitted during *fast recovery.* These retransmissions are controlled such that the number of its packets in the link do not exceed *cwnd.* The retransmission timers are reset by the same algorithms observed in TCP New Reno.

Artificial Window Inflation and Deflation: [6] The window deflation and inflation algorithms of TCP New Reno are used during *fast recovery* to ensure that if there is data to send, *cwnd* number of packets are maintained in the pipe. k-SACK uses the improved pipe estimate for its window inflation and deflation and this allows a k-SACK sender to retransmit the lost segments earlier and therefore results in faster recovery. The *cwnd dynamics* already discussed, however, do not operate on the artificially inflated or deflated values but the actual values.

Timeout: On a timeout event, the k-SACK sender clears the SACK information and employs a go-back-n mechanism as observed in [2]. The lookahead loss is reset to zero and the sender goes into slow start.

III. RATIONALE

In the absence of any external notification, such as ELN [8], all previous TCP versions interpret a packet drop as a sign of congestion. This policy was adopted since the cumulative acknowledgements impose the constraint that only one packet loss can be detected at a time, and subsequent losses can be detected only after recovery of all previous losses. The use of the SACK option helps eliminate this constraint. The SACK information carries meaningful information about segments that have reached the receiver. It can be used to detect congestion in a more intelligent manner. The aim of k-SACK is to detect congestion only when multiple losses are detected close to each other. This prevents a k-SACK sender from reacting inappropriately to single random losses by cutting down its window. The SACK option is used to detect potentially lost segments beyond the cumulative acknowledgement.

The earlier TCP versions use *max-dupacks* (typically 3) duplicate acknowledgements for detecting a lost segment. k-SACK extends the same policy to detect packet losses beyond the duplicate or partial acknowledgements. A packet is also considered *lost* if it is neither acknowledged *(acked)* nor selectively acknowledged *(sacked)* and at least *max-dupsacks* (analogously taken to be 3) segments with higher sequence numbers have been *sacked.*

Thus, with the knowledge of lookahead-loss, a TCP sender can make better decisions about invoking congestion control and avoidance algorithms. We suggest the simplest mechanism of invoking special congestion control dynamics on detecting at least k segments using the lookahead-loss, and more liberal mechanisms for less than k losses and it would be inappropriate to allow the sender to continue sending increasingly more packets into the pipe until the detection of k losses. We recommend the freezing of congestion window until the losses are recovered. In case of random drops, an isolated drop is recovered in approximately one *rtt* and the sender continues in slow start or congestion avoidance depending upon the *cwnd* and *ssthresh* values. However, if the loss was actually caused due to congestion in the network, the sender will still continue to send more packets in the pipe, as per the discussed mechanism. This will result in further losses and the congestion control mechanism (k-recovery) will come into effect on detecting the kth loss. In such a case, the TCP sender sets the *ssthresh* to *max(cwnd/2, 2MSS)* and then effectively halves the *cwnd.* The sender then continues in k-recovery and waits either for the completion of fast recovery or the lookahead loss to fall below k, whereupon the halt growth phase of fast recovery is invoked. This is done in order to let the TCP sender complete the fast recovery and not suffer more losses resulting in a transition back to k-recovery phase. If the growth is not halted at this point the sender may oscillate between the halt growth phase and the k-recovery phase, in certain cases, which is not desirable. However, if a sender is made to enter the k-recovery phase again within the same fast recovery, the *cwnd* is halved but the *ssthresh* is not changed. In brief, the mechanism mimics the fast recovery phase of SACK over New Reno implementation, except for the behavior of *cwnd* and *ssthresh* for different values of lookahead-loss.

IV. Simulation Results

In our study, we have assumed that packet errors are independent identically distributed (i.i.d). It is to be noted that bursty losses in wireless links yield better results (i.e., higher TCP throughput) as compared to random i.i.d packet losses [5]. This

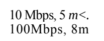


Fig. 3. Time taken for 100 Kb file transfer (DropTail gateway).

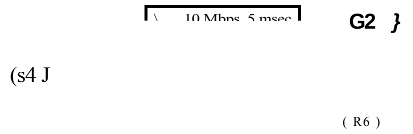
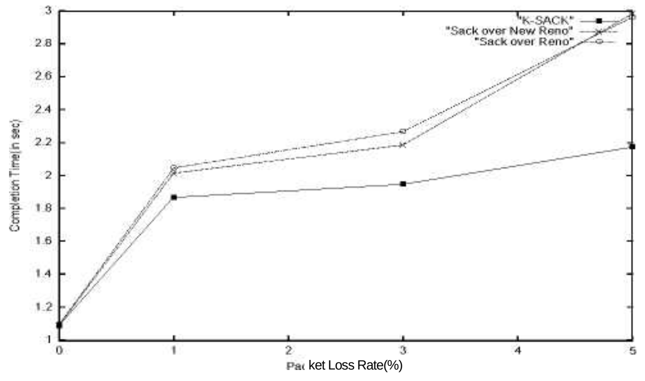


Fig. 2. The Network Topology



paper therefore considers the worst case scenario.

We have developed a simulation model to compare the performance of k-SACK with TCP SACK over New Reno and TCP SACK over Reno using the *ns-2* [7] simulator. We have implemented k-SACK for k equal to two. Higher values of k may make the TCP version aggressive and result in a disadvantage to other TCP versions. However a k-SACK implementation with higher k values will perform much better over links with very high error rates.

The network topology is shown in Figure 2. G1 and G2 are gateways sitting on opposite sides of the bottleneck link, with a bandwidth of 10 Mbps and 5 ms delay. The other links are very high capacity links. S1 through S6 are TCP sources and R1 through R6 are the corresponding sinks. To simulate the effect of non-TCP traffic in the internet, we have chosen to use the nodes U1 and U2 as a source and sink, the source sending Pareto distributed UDP traffic to the sink through the bottleneck link. The bottleneck link is modeled as a lossy link with varying packet loss rates. The default packet size of 576 bytes is used to carry the data.

We have measured three different metrics to characterize the behavior of TCP k-SACK as compared to SACK over New Reno and SACK over Reno. The simulation experiments and results are discussed in the following sections.

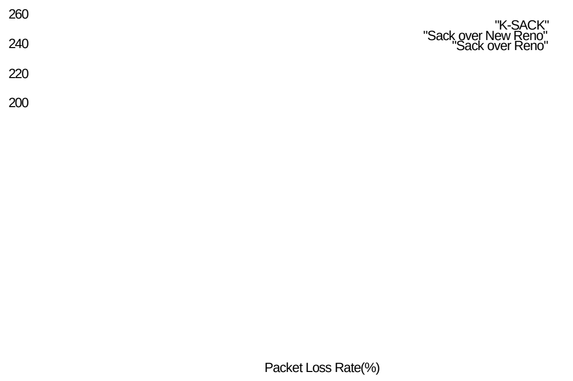


Fig. 4. Time taken for 10 Mb file transfer (DropTail gateway)

A. Completion Times

We have measured the average time taken by the above mentioned versions for fixed file transfers of sizes 100 Kb (short transfer; see Figure 3) and 10 Mb (large data transfer; see Figure 4) across the bottleneck link. For these simulations, the bottleneck link also carries 15-20 % Pareto distributed UDP traffic. We have performed simulations using DropTail (graphs shown in Figure 3, Figure 4), as well as RED gateways, at the boundaries of the bottleneck link. In our simulations, acknowledgements are never dropped since there is no congestion in the reverse path. The time taken is averaged over six simultaneous connections and also over multiple simulation runs. Other than due to random losses, the simulation behavior was forced to change over multiple runs by varying the starting times of the TCP connections.

The results show that k-SACK takes almost the same time as Sack over New Reno and SACK over Reno to transfer the same amount of data in a non-lossy environment (0% loss rate). As seen in Figure 3, the performance gain with increasing loss rates is significant across DropTail gateways. The same observation holds for RED gateways. For short data transfers, the performance of k-SACK does not degrade by more than 0.2 seconds with increasing error rates, while the other versions take upto one second more for the same data transfer. This is to be expected since, with increasing loss rates, there are more random drops, which the other TCP versions interpret as congestion and decrease their sending rate. k-SACK on the other hand uses the lookahead-loss and does not reduce its congestion window for many instances of such random losses and hence shows a performance gain. A marked improvement is observed in end-to-end data transfer time for long data transfers (see Figure 4). The performance for 10 Mb file transfer over a lossy link with 5% packet loss rate is almost two-fold for DropTail gateways. This is clearly attributed to the superior congestion detection scheme implemented by k-SACK.

B. Link utilization/Throughput Performance

In our simulations, we have measured the throughput of each variant for a long (persistent source) data transfer. We have used six simultaneous connections all of the same variant at one time.

TABLE I
TABLE FOR BANDWIDTH UTILIZATION (IN MBPS) PER CONNECTION WITH DROPTAIL GATEWAYS (BANDWIDTH AVAILABLE=10MBPS, SIMULTANEOUS CONNECTIONS=6)

Loss%	k-SACK	Sack-Newreno	Sack-Reno
0%	1.66	1.66	1.66
1%	1.58	1.18	1.20
3%	0.98	0.60	0.56
5%	0.69	0.36	0.36

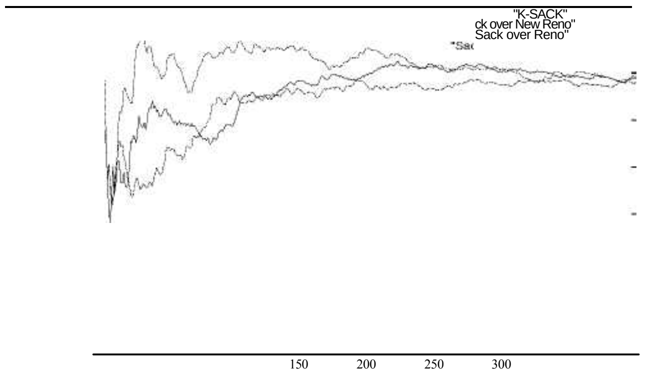


Fig. 5. Throughput in a non-lossy environment

The throughput behavior of a random TCP connection of each variant is plotted in Figure 5 (non-lossy environment), Figure 6 (low packet loss rate), Figure 7 (high packet loss rate). The throughput values averaged over multiple runs are also representative of the link utilization by the TCP senders.

As can be seen from Table I, k-SACK can occupy almost twice the bandwidth occupied by other TCPs over a lossy link with 5% packet loss rate. At 1% loss rates, the performance gain is again about 0.3 Mbps per connection. This significant improvement can be explained in terms of the different congestion control and avoidance algorithms. k-SACK with its lookahead loss and k parameter (chosen to be 2 in this paper) can achieve better throughput and link utilization.

C. Fairness

k-SACK is conservative in its estimation of a congested network and therefore has an advantage over the other TCP versions (see Figure 8). To characterize the advantage gained by k-SACK in a competing environment we define the notion of a performance ratio. The average throughput j_{nc} per connection is measured for n simultaneous connections, all of the same variant. The average throughput 7 $_c$ is also measured per connection per variant, for n connections with ^ connections of each TCP variant (three in our case). The former measurement refers to a non-competing environment and the latter to a competing environment.

The Performance ratio is measured as the percentage gain in performance as seen by a TCP variant in a competing environ-

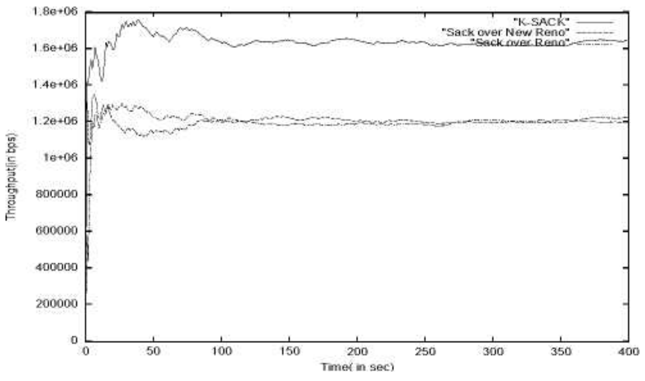


Fig. 6. Throughput with 1% packet loss rate.

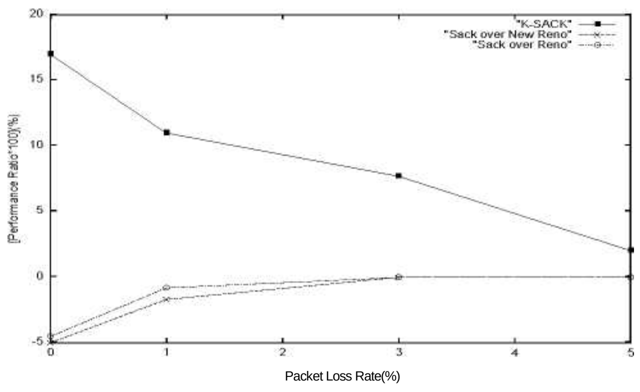


Fig. 8. Performance Ratio across a Drop Tail gateway.

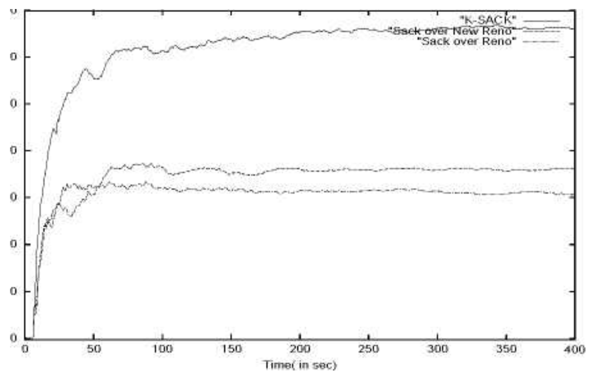


Fig. 7. Throughput with 5% packet loss rate.

ment as compared to that over a non-competing environment. More formally, $1Z = {}^{\tau}_{;}$ where, $1Z$ is the performance ratio, 7_C is the throughput in a competing environment and 7_{nc} is the throughput in a non-competing environment.

We have used six simultaneous connections to measure the Performance ratio, with two connections of each type for the competing environment. We have observed with multiple runs that at higher loss rates, the other TCPs do not suffer any performance degradation, while k-SACK shows an improvement between 5-10% (see Figure 8). This is expected since the other TCPs are not capable of occupying the entire bandwidth. However, the improvement in k-SACK is because it now occupies the bandwidth that is left unused by the other TCP versions. The same bandwidth was being occupied by competing TCP connections of the same variant, in experiments with all flows of the same variant. For close to 1% loss rate, The other TCPs suffer a performance fall of less than 2%. For a non-lossy environment, the other TCPs have a performance degradation of close to 5% for DropTail (Figure 8). Other TCP versions show a 5% further fall in performance ratio when a RED gateway is used instead of a DropTail gateway at G1. This is explained by the early packet drop policy in RED gateways.

Though the other TCPs show a performance degradation in non-lossy links, it can be argued that the performance gains over lossy links are much larger in comparison. Looking at a heterogenous network, k-SACK will perform much better than existing TCP versions. Further, it is evident that if all connections use the k-SACK implementation, the behavior in normal non-lossy links defaults to that shown by the existing implementations.

V. Conclusion

Motivated by the poor performance of TCP over lossy links (such as wireless and satellite systems), we have studied a variant of TCP SACK over New Reno, which we refer to as TCP k-SACK. The key idea of the protocol is its novel congestion detection mechanism wherein the k-SACK sender does not assume congestion on detecting a single packet loss, but waits for k losses in a loss-window to infer congestion. The proposed variant assumes a value of k equal to two and the loss-window equal to the congestion window. TCP k-SACK performs equally well as TCP SACK over non-lossy wireline links. The protocol does not need to be fine tuned separately for lossy and non-lossy environments, and also does not involve any additional overheads. TCP k-SACK has significantly improved throughput characteristics over lossy links. The behavior of k-SACK for different values k and loss-window is a subject of further research.

References

[1] W. Richard Stevens *TCP/IP Illustrated, Volume 1: TheProtocols.* Addison Wesley, 1994.

[2] M. Mathis, J. Mahdavi S. Floyd and A. Romanow. "TCP Selective Acknowledgement Options,", RFC-2018.

[3] A. Bakre and B. R. Badrinath. "I-TCP: Indirect TCP for Mobile Hosts,". *Proc. 15th International Conference on Distributed Computing Systems,* May 1995.

[4] H. Balakrishnan, V. N. Padmanabhan, S. Seshan and R. H. Katz, "A Comparison of Mechanisms for Improving TCP Performance,". *Proc. ACM Sigcomm 1996,*. Aug 1996.

[5] A. Kumar, "Comparative Performance Analysis of Versions of TCP in a Local Network with Lossy Link". *IEEE/ACM Transactions on Networking,* Dec. 1996.

[6] S. Floyd and T. R. Henderson. "The Newreno modification to TCP's Fast Recovery Algorithm,". *RFC 2582,* April 1999.

[7] S. McCanne ans S. Floyd. "ns-Network Simulator,". *http://www-nrg. ee. lbl.gov/ns*

[8] H. Balakrishnan and R. H. Katz. "Explicit Loss Notification and Wireless Web Performance,". *Proc. IEEE Globecom 1998,* Internet mini-conference, Sydney, Australia.