

TCP End-to-End Performance Improvement over Wireless Networks via Early Packet Loss Recovery

A. Jayanathan

Electrical and Computer Engineering
University of Canterbury
New Zealand
a.nathan@elec.canterbury.ac.nz

Harsha R. Sirisena

Electrical and Computer Engineering
University of Canterbury
New Zealand
sirisehr@elec.canterbury.ac.nz

Abstract—A new technique is proposed that enables TCP to early detect packet losses, which cannot be detected and retransmitted using the standard fast retransmit mechanism, and to quickly retransmit those packets without waiting for a timeout to occur, thereby improving TCP performance. Early Packet Loss Recovery (EPLR) is achieved by considering the expected number of acknowledgements and the number of packets in a flight during the fast retransmit phase of the TCP mechanism. It adds a new flavor to the TCP fast retransmit and recovery mechanism without requiring any other modification to the standard TCP implementation. The scheme is modeled using the OPNET tool, and the simulation results are presented and explained. The performance of the proposed scheme is compared with that of both TCP Reno and TCP New Reno over a UMTS network. Our results show that the proposed scheme, EPLR, significantly improves the TCP performance in a UMTS environment.

Keywords- TCP Throughput, Timeout Avoidance, Fast Retransmit, UMTS Network, OPNET Model

I. INTRODUCTION

Due to the scarce bandwidth resource and cost considerations, it is imperative to design and optimize wireless networks currently being deployed, such as UMTS, WiFi and WiMAX, to get maximum application layer performance. The Transmission Control Protocol (TCP) supports the most popular applications on the Internet today. TCP is a reliable connection-oriented protocol [1] employing a window-based congestion control mechanism. It is primarily designed for wired networks where the random bit error rate is negligible and the main cause of packet loss is network congestion. Communication over wireless links is, however, characterized by sporadic high bit-error rates, and intermittent connectivity due to handoffs. TCP performance in such networks suffers from significant throughput degradation and very high interactive delays [2] because the assumptions made by TCP do not hold in a wireless environment.

A number of approaches, such as Explicit Congestion Notification (ECN) [3], Split connection protocols [4] and the Snoop protocol [5] have been proposed to improve the TCP performance over an unreliable wireless network. The aforementioned protocols use two different approaches to

improve the TCP performance over wireless links; one is to hide non-congestion related wireless losses from the TCP sender and the other is to make the TCP sender aware of these losses. However, they all fail to completely recover from the wireless effects [5, 6].

In this paper, we propose a new scheme, called Early packet loss recovery (EPLR), that improves the TCP performance over lossy networks by minimizing the number of TCP timeouts, resulting in high TCP throughput and application response performances.

The rest of the paper is organized as follows. An overview of different TCP fast retransmit and recovery mechanisms is given in Section II. The proposed scheme and its OPNET implementation are given in Sections III. The network model and simulation results are presented in Section IV. The conclusions drawn and ideas for future work are presented in Section V.

II. TCP OVERVIEW

There are many TCP flavors, such as Tahoe [7], Reno [7], New Reno [8] and SACK [9], which differ in how they react to packet loss. A packet loss is detected either by the arrival of three duplicate acknowledgements (dupacks), or the absence of an acknowledgement (ack) for the packet within a timeout interval called retransmission timeout (rto). All TCP implementations reset cwnd after rto expiration to one maximum segment size (MSS). However, they may proceed differently after dupacks are received. The missing segment is always retransmitted immediately, but transmission of new or unacknowledged data depends on the selected TCP flavor. Consider a scenario shown in Figure 1, where packets P_N , P_{N+4} and P_{N+5} are lost while there is F number of packets in the flight.

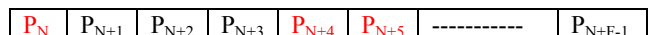


Figure 1. A flight of data in the network

TCP Tahoe [7] retransmits the lost packet P_N and enters into slow start phase, setting its cwnd to one MSS. The next ack that acknowledges the packets up to P_{N+3} allows the sender to increase its cwnd to two MSS and resend the packets P_{N+4} and P_{N+5} . The ack for P_{N+4} increases the sender's cwnd to three MSS and packets P_{N+6} and P_{N+7} can

be sent. The ack for P_{N+5} acknowledges packets up to P_{N+F-1} . The sender then continues transmitting new data. Notice that the packets P_{N+6} and P_{N+7} are unnecessarily retransmitted, assuming $F > 7$, and TCP Tahoe recovers from the packet losses within a window of data without retransmission timeout expiration.

TCP Reno [7] retransmits the lost packet P_N and enters into fast recovery phase, setting its cwnd to $(F/2 + 3)$ times MSS. The sender then continues receiving more dupacks and increases its cwnd by one MSS for each ack. The ack for retransmitted packet P_N takes the sender out of fast recovery and the cwnd is set to $F/2$ times MSS. The sender then waits for another three dupacks to retransmit the packet P_{N+3} . If the sender receives three dupacks, it will retransmit the packet P_{N+3} otherwise, it has to wait for the retransmission timer to expire. This increases the application response time considerably.

TCP New Reno [8] retransmits the packet P_N , resets the cwnd and enters into recovery phase as it does in Reno. The process then continues receiving more dupacks and increases its cwnd by one MSS for each received ack. Unlike in Reno, the ack for the retransmitted packet P_N does not take New Reno out of the recovery process. Partial acks add one MSS to cwnd and decrease it by the amount of acknowledged data. Notice that during the fast recovery process, it retransmits the unacknowledged data packets whenever it receives a partial ack without waiting for the third dupack to arrive, which allows the process to resend the packets P_{N+4} and P_{N+5} .

TCP SACK [9], similar to the flavors explained above, retransmits packet P_N using the fast retransmit algorithm. However, it uses a different approach to determine when and which packets are sent out during fast recovery. It calculates the amount of in-flight data based on selective acks that it has received. Data can be sent only if the amount of outstanding data is lower than the size of the cwnd. Because it has information about which packet were received, it is able to resend only missing segments and then continue with transmission of unsent data.

III. PROPOSED SCHEME AND OPNET IMPLEMENTATION

Our proposed scheme takes the number of packets in a flight as the system metric to quickly fast retransmit the dropped packet that can only be recovered by TCP timeout with the existing fast retransmit and recovery algorithms, such as Reno and New Reno.

Let cwnd be W , the number of dropped packets be N , where $N \geq 1$, L_1 be the dropped packets within that window of data, as shown in Figure 2, D_1 be the number of packets from L_1 to P_W and the flight size be F . Number of packets can be recovered by TCP fast retransmit and recovery algorithms depend on W , N and D_1 . The TCP sender can only transmit new packets if minimum of the cwnd and the receiver window is greater than F , defined as the amount of data that has been sent but not yet acknowledged. We assume that the receiver window is bigger than W for the ease of analysis.

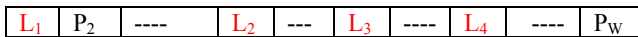


Figure 2. A window of data with multiple packet drops

A. TCP Reno Congestion Window Analysis

1) One packet recovery

In order to recover the first dropped packet, it requires that $W \geq N+3$. During this recovery process, there can be up to $W/2-N$ number of new packets transmitted, causing F to grow from W to $3W/2-N$. On receiving the ack for the retransmitted packet L_1 , F will be equal to $D_2+(W/2-N)$ and cwnd is set to $W/2$.

2) Two packets recovery

Notice that on receiving the first partial ack, there can be maximum of one new packet transmitted if and only if $N-D_2=1$. It makes $F = \text{cwnd} = W/2$. Assuming it is not the case, there will be only $W/2-N$ number of packets to be acked out of $(D_2+W/2-N)$ number of packets in the network. It requires $W \geq 2(N+3)$ to be able to recover the second dropped packet. During this recovery process, $W/4-D_2$ number of new packets can be transmitted, provided $W/4 > D_2$. On receiving the ack for the retransmitted packet L_2 , F will be equal to $D_3+(W/2-N)+(W/4-D_2)$, and cwnd is set to $W/4$.

3) Three or More packets recovery

Third packet recovery requires $W/4-D_2 \geq 3$ and allows $W/8-D_3$ number of new packets to be transmitted, provided $W/8 > D_3$. It can be generalized that it requires $W/2^{N-1}-D_{N-1} \geq 3$ to recover from the N^{th} packet, provided $(N-1)^{\text{th}}$ packet is recoverable and $N > 2$.

B. TCP New Reno Congestion Window Analysis

TCP New Reno only requires $W \geq N+3$ to initiate the fast retransmit and recovery algorithms. Because, once entered into the recovery phase, it can recover from multiple packet drops within a window of data by retransmitting the unacknowledged packets whenever it receives a partial ack.

C. Proposed Scheme

In order to proceed with packet transmissions, dropped packets must be retransmitted as quickly as possible. TCP Reno and New Reno fast retransmit and recovery algorithms are well defined and designed to handle this effect. However, they fail to consider situations where the fast retransmit and recovery algorithms cannot be even initiated.

- If the window size is less than or equal to the dupack threshold, which is normally assigned to be three, either the TCP Reno or New Reno cannot even initiate the fast retransmit and leave this packet to be recovered by means of TCP timeout process.
- TCP Reno cannot initiate the fast retransmit to recover from multiple packets if $\text{cwnd} < 10$.

Our insight is that if the window size is too small to initiate the fast retransmit, it must be handled separately.

We modify the TCP Reno fast retransmit algorithm in order to recover from up to two packets within a window of data if cwnd is too small to initiate the fast retransmit. Now, it comes to deciding when to retransmit the unacknowledged packet. Clearly, the TCP sender cannot confirm a received dupack was due to packet loss because packets in a flight could take different route and reach the destination out of

order. Given the number of packets in a flight is F and is equal to the $cwnd$, we can safely assume that a packet is dropped if the sender receives $(F-1)$ number of dupacks and allow the sender to quickly retransmit that packet if the flight size is too small to initiate the fast retransmit and recovery algorithms. However, due to the sender's inability to confirm the packet loss, we decide to allow the sender to transmit a new data packet by increasing the $cwnd$ by one MSS. It enables the sender to receive threshold number of dupacks and either to fast retransmit the lost packet if a third dupack is received or to continue transmitting new data if a non duplicate ack is received. This will considerably increase the TCP throughput and application response time while minimizing the number of TCP timeouts.

D. OPNET Implementation

Our proposed scheme is implemented in OPNET[10] by adding a new TCP flavor, called Modified Reno, to the TCP fast retransmit and recovery algorithms. Figure 3 shows the simplest version of EPLR flow control for Modified Reno fast recovery and retransmit processing.

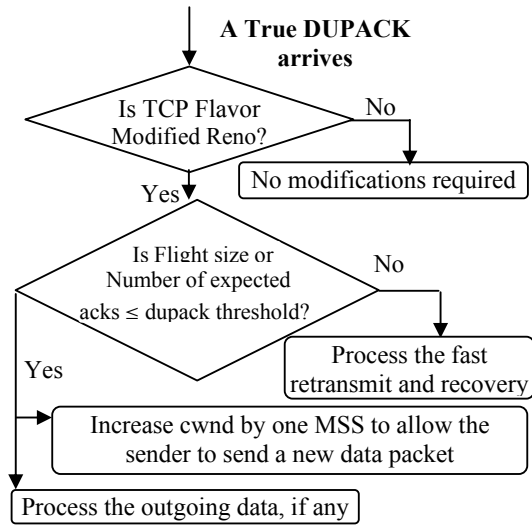


Figure 3. Simplest version of EPLR Flow control

IV. OPNET NETWORK MODELS

A. UMTS Network Model

To demonstrate the effectiveness of our proposed scheme, the UMTS network model shown in Figure 4 is implemented, in turn, with different TCP fast retransmit algorithms at the standard FTP server: Reno, New Reno and Modified Reno. The FTP server is configured to generate files of 1 Mbyte size. User equipments (UEs) are configured to download 1 Mbyte FTP files simultaneously with different packet drop rates as shown in Table I. Data Packets coming from FTP server are dropped in UEs, at the IP layer, using a uniform probability distribution. UMTS and TCP with their default parameters [10] are used in all simulation scenarios and an extract of the UMTS and TCP parameter values are given in Tables II and III, respectively.

TABLE I. UES CONFIGURATIONS

UEs	UE_1	UE_2	UE_3	UE_4
Packet drops (%)	4	6	8	10

TABLE II. SELECTION OF UMTS RNC PARAMETERS

Transmission Window Size	32
Receiver Window Size	32
SDU Discard Mode	Timer Based No Explicit
Timer MRW (milliseconds)	140
Timer Discard (milliseconds)	1500
Max MRW	6
Max DAT	4
In-Sequence Delivery	No
UL RLC Mode	Acknowledged Mode
DL RLC Mode	Acknowledged Mode

TABLE III. SELECTION OF TCP PARAMETERS

Receive Window Size at FTP server	8760 bytes
Receive Window Size at UEs	Default
Maximum ACK Delay	0.2 seconds
Maximum ACK Segments	2
Duplicate ACK Threshold	3
Maximum Segment Size (MSS)	1460 Bytes
Sack Option	Disabled
Slow-Start Initial Count	1 MSS
Minimum RTO	1 seconds
Maximum RTO	64 seconds

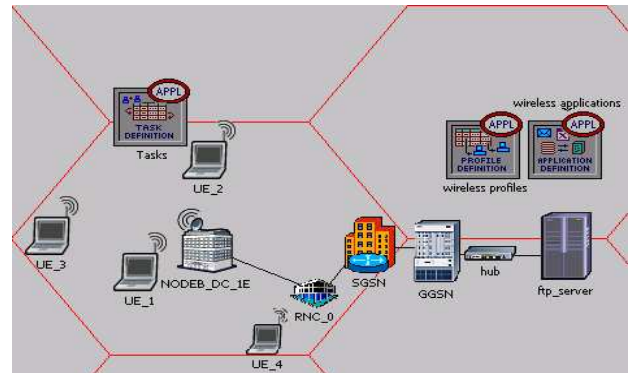


Figure 4. UMTS network model

B. Results and Observations

Figures 5 and 6, respectively, compare the TCP sent segment sequence number and the TCP $cwnd$ size responses of our proposed scheme with that of TCP Reno and New Reno implementations for packet drop rates of 4%, 6%, 8% and 10% respectively. A summary of the average TCP throughput and the TCP performance improvements with our proposed scheme over that of Reno and New Reno are given in Table IV and Table V respectively. It can be seen that our proposed scheme improved the TCP throughput compared to that of TCP Reno significantly in all cases. This improvement can be directly attributed to the reduction of TCP timeouts, which can be observed in Figure 6.

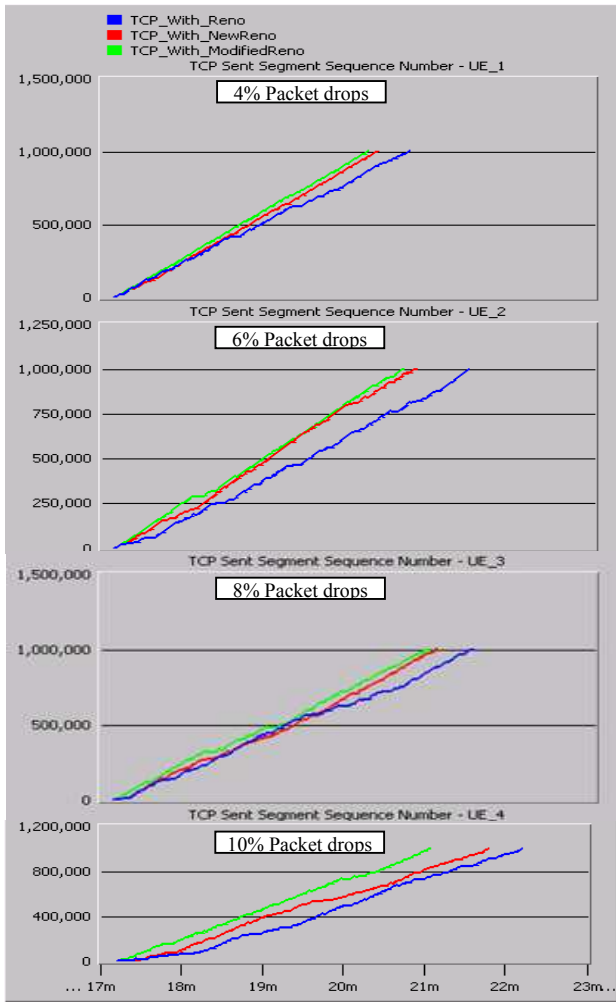


Figure 5. TCP sent segment sequence number

TABLE IV. SUMMARY OF AVERAGE TCP PERFORMANCE RESULTS

TCP Throughput in (Kbps) with	Packet Drop Rates (%)				Total Throughput
	4	6	8	10	
Reno	37.38	32.14	29.51	28.54	127.57
New Reno	41.73	36.55	33.36	32.68	144.32
Modified Reno	43.47	38.12	35.30	34.80	151.69

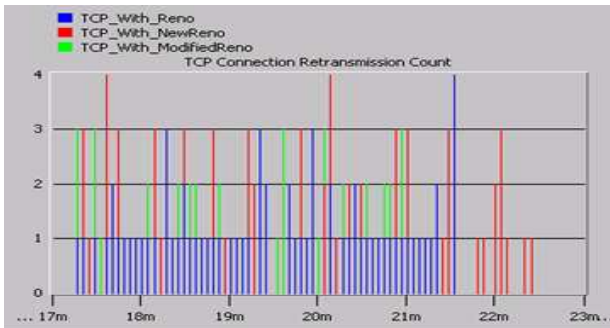


Figure 7. TCP Retransmission Count for UE_4

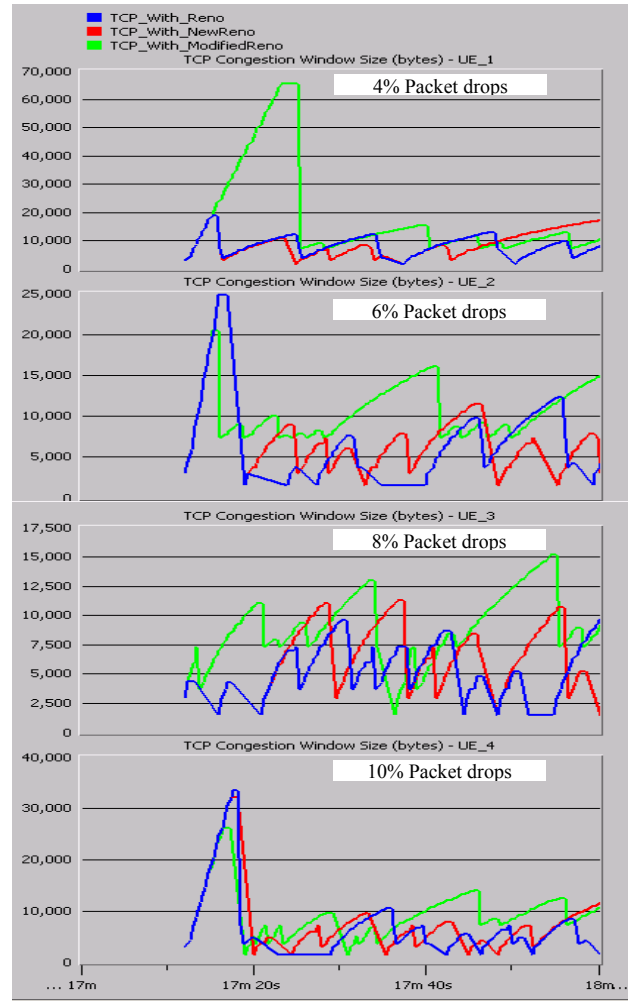


Figure 6. TCP cwnd response

TABLE V. AVERAGE TCP PERFORMANCE IMPROVEMENT

TCP Performance Improvement over	Packet Drop Rates (%)			
	4	6	8	10
Reno	16.29	18.61	19.62	21.93
New Reno	4.17	4.30	5.82	6.49

In order to prove the point that we have not overloaded the network, the number of retransmission counts during the file download by UE_4 is given in Figure 7. It can also be observed that our proposed scheme seems to start the retransmissions sooner than does TCP Reno. To explain and compare the proposed fast retransmit and recovery mechanisms with that of the standard TCP Reno, a snapshot of the TCP sent and ack sequence number responses, during UE_4 file download, with TCP Reno and Modified Reno is given in Figure 8.

In Case-1, the sender with Modified Reno receives the first dupack with the flight size equal to four packets. It therefore does wait for the third dupack to arrive to trigger the fast retransmit. Since it has not received the third dupack in time, it times out and retransmits using the TCP timeout

process. In Case-2, the sender receives the first ack with the flight size equal to two packets. The Modified Reno then allows the sender to transmit a new packet for each dupack. It enables the sender to receive three dupacks and to confirm the packet loss. As expected, it does receive the third dupack and the dropped packet gets retransmitted without waiting for a TCP timeout that would occur with either the Reno or New Reno retransmit mechanisms.

In Case-3, the sender with Reno receives the first dupack with the flight size is equal to four packets. It therefore does wait for the third dupack to arrive to trigger the fast retransmit. Since it has received the third dupack in time, the dropped packet gets retransmitted without a TCP timeout. In Case-4, the sender receives the first ack with the flight size equal to three packets. The Reno sender then waits for the third dupack to arrive to confirm the packet loss. Since the sender will never receive three dupacks, this dropped packet can only be recovered by the TCP timeout process.

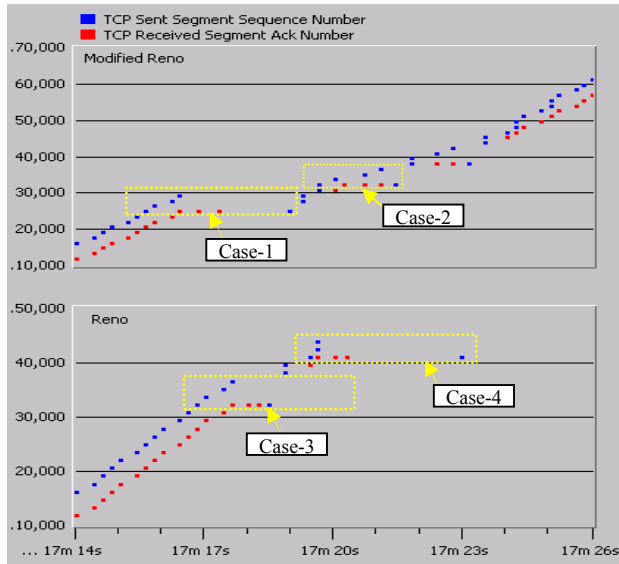


Figure 8. TCP Sent and ack number (UE_4)

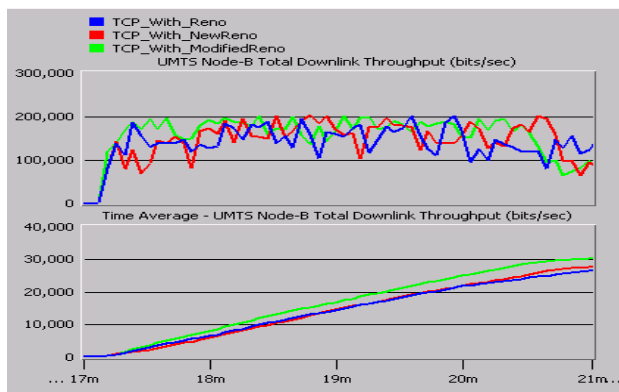


Figure 9. UMTS Node-B Downlink Throughput

Compared to TCP New Reno, our proposed scheme does always outperform in all scenarios. This is because New

Reno also undergoes the same problem when the flight size is too small to trigger the fast retransmit. This can be observed from the TCP cwnd response in Figure 6, where New Reno experiences more timeouts than the Modified Reno does. Finally, in order to show that our proposed scheme has utilized the available network resources efficiently, a comparison of the UMTS Node-B downlink throughput performance is shown in Figure 9.

V. CONCLUSIONS

TCP Reno fast retransmit algorithm was modified with a new EPLR mechanism to speed up the packet recovery process and to reduce the number of TCP timeouts over networks with heavy packet losses, such as wireless networks. Modified Reno was implemented in a UMTS network and its performance was compared with that of Reno and New Reno. Simulation results showed that Modified Reno improved the TCP performance and application response time significantly compared to that of both Reno and New Reno by reducing the TCP timeouts, which is the main cause of degradation of the TCP performance in a wireless environment. In the future work, we intend to further validate the model in other wireless networks such as WiFi and WiMAX.

REFERENCES

- [1] J. B. Postel, "Transmission control protocol," RFC, Information Sciences Institute, Marina del Rey, CA, vol. RFC-793, Sept. 1981.
- [2] R. CACERES, AND IFTODE, L., "Improving the performance of reliable transport protocols in mobile computing environments," *IEEE J. Select. Areas Commun.*, vol. 13, pp. 850--857, June 1995.
- [3] R. Ramani and A. Karandikar, "Explicit congestion notification (ECN) in TCP over wireless network," 2000.
- [4] A. Bakre and B. R. Badrinath, "I-TCP: indirect TCP for mobile hosts," 1995.
- [5] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving TCP performance over wireless links," *Networking, IEEE/ACM Transactions on*, vol. 5, pp. 756-769, 1997.
- [6] T. N. Prasun Sinha, Narayanan Venkitaraman, Raghupathy Sivakumar, Vaduvur Bharghavan "WTCP: a reliable transport protocol for wireless wide-area networks," Kluwer Academic Publishers, March 2002.
- [7] V. Jacobson, "Congestion avoidance and control," Univ. of California, Berkeley, 1988
- [8] S. Floyd and T. Henderson, "RFC 2582 - The NewReno Modification to TCP's Fast Recovery Algorithm," April 1999.
- [9] Mathis .M, Mahdavi .J, Floyd .S, and Romanow .A, "TCP Selective Acknowledgement Options," RFC 2018, April 1996.
- [10] OPNET, "Making Networks and Applications Perform," OPNET Technologies Inc, <http://www.opnet.com/>.