Modified Fast Recovery Algorithm for Performance Enhancement of TCP-NewReno

Hanaa Torkey
Department of Computer Science and Engineering,
Faculty of Electronic Eng.,
Minufiya University, Egypt.

Gamal Attiya
Department of Computer Science and Engineering,
Faculty of Electronic Eng.,
Minufiya University, Egypt.

Ibrahim Z. Morsi
Department of Electrical Engineering,
Faculty of Engineering,
Minufiya University, Egypt.

ABSTRACT
One of the keys to the success of the Internet is relying on using efficient congestion control mechanisms. Congestion control is required not only to prevent congestion collapse in the network, but also to improve network utilization. Without congestion control, a sending node may continue transmitting packets that may be dropped later due to congestion collapse. This paper presents a modified fast recovery algorithm to enhance the performance of the most widespread congestion control protocol; TCP-NewReno. The proposed mechanism is evaluated by using the network simulator NS-2 and compared with both the TCP-NewReno and the TCP-Reno. The simulation results show that the proposed mechanism improves the performance of the TCP-NewReno against throughput and packet delay.

General Terms
Computer Networks, Network Protocols.

Keywords
TCP, Congestion Control, Congestion Avoidance, NewReno, Fast Recovery Algorithm.

1. INTRODUCTION
Transmission Control Protocol (TCP) has been the dominant transport protocol for reliable data transfer over the Internet. It supports most of the popular Internet applications, such as the World Wide Web, file transfer and e-mail. However, the rapid growth of the Internet and the increasing demand of different traffics over the Internet lied to a serious problem called congestion collapse [1]. This problem occurs when the aggregate demand for resources exceeds the available capacity of the network.

Congestion is generally bad for network users, applications and network performance. When a packet encounters congestion, there is a good chance that the packet is dropped, and the dropped packet wasted precious network bandwidth along the path from its sender to its destination. Congestion control is thus required to prevent congestion collapse in the network and improve the network performance. Without congestion control, a sending node could be busy transmitting packets that may be dropped later due to congestion collapse.

After observing a series of congestion collapse, several congestion control algorithms are proposed and incorporated into the TCP to resolve the congestion collapse problem. In 1988, several innovative congestion control algorithms were introduced into TCP [2]. This TCP version is called TCP Tahoe. It includes three algorithms namely Slow Start, Congestion Avoidance, and Fast Retransmit. Two years later, a Fast Recovery algorithm was added to Tahoe to form a new TCP version called TCP Reno [3]. TCP Reno is a reactive congestion control scheme that uses packet loss as an indicator for congestion. In order to probe the available bandwidth along the end-to-end path, the TCP congestion window (cwnd) is increased until a packet loss is detected, at which point the congestion window is halved and a linear increase algorithm takes over until further packet loss is experienced. Generally, the congestion window is used to limit the amount of data that the sender can inject into the network in order to prevent the source from overrunning the capacity of the network. In Reno, the TCP sender changes its congestion window size according to the congestion control algorithms; Slow-Start, Congestion Avoidance, Fast Retransmit and Fast Recovery [4].

In [5], the authors have shown that the TCP Reno may periodically generate packet loss by itself and cannot efficiently recover multiple packet losses from a window of data. Moreover, the Additive Increase and Multiplicative Decrease (AIMD) strategy of TCP Reno leads to periodic oscillations in the aspects of the congestion window size, round-trip delay, and queue length of the bottleneck node. Indeed, the oscillation may induce chaotic behavior in the network, thereby adversely affecting overall network performance. To alleviate the performance degradation problem of packet loss, many researchers attempted to refine the Fast Retransmit and the Fast Recovery algorithms of the TCP Reno [6, 7]. In [8], a congestion control mechanism, called TCP NewReno, is developed using an augmented Fast Recovery algorithm to overcome the problem of TCP Reno and combat multiple packet losses from the same transmission window without entering into Fast Recovery multiple times. That is, TCP NewReno modifies the sender behavior during Fast Recovery algorithm, where, it continues in Fast Recovery until all the packets which were outstanding during the start of the Fast Recovery have been acknowledged.

Although the additional modifications to the Fast Recovery algorithm improve the performance of TCP NewReno, it has been found that the TCP NewReno is inefficient in terms of utilization of link capacity and unfair in throughput [9-12]. The problem with NewReno is that, within Fast Recovery algorithm, it halves its congestion window irrespective of the state of the network as long as a packet loss is detected. Another problem arises with NewReno is that when there are no packet losses, but packets are reordered by more than three duplicate acknowledgments; NewReno mistakenly enters Fast Recovery, and halves its congestion window [13, 14].

This paper presents a modified Fast Recovery algorithm to improve the TCP-NewReno performance. The basic idea is to adjust the congestion window (cwnd) of the TCP sender based on the level of congestion in the network so as to allow transferring more packets to the destination. The proposed mechanism, called Enhanced NewReno (EnewReno), is evaluated by using the network simulator NS2 and compared with both the TCP-NewReno and the TCP-Reno. The simulation results show that the proposed mechanism improves the performance of the TCP-NewReno against throughput and packet delay.

The rest of this paper is organized as follows; Section 2 presents an overview of the most widespread congestion control protocol; TCP-NewReno. The proposed mechanism is described in Section 3 while Section 4 presents the simulation results and discussions. Finally, the paper conclusions and the direction for future work are given in Section 5.

2. TCP NewReno

Modern TCP implementations incorporate various congestion control algorithms to adapt the sending rate at the sender site in order to overcome the congestion collapse. This section briefly describes the main algorithms of the TCP NewReno; Slow-Start, Congestion Avoidance, Fast Retransmit and Fast Recovery.

2.1 Slow Start and Congestion Avoidance

In TCP NewReno, when a TCP connection begins, the Slow Start algorithm initializes a congestion window (cwnd) to one segment [15]. The sender is then start transmitting packets based on the window size. For each acknowledgement returned from the receiver, the congestion window is increased by one segment. This behavior continues until the cwnd arrives to the slow start threshold (ssthresh). At this point, the NewReno enters into Congestion Avoidance phase to slow the increasing rate of the cwnd. During congestion avoidance, the congestion window increases linearly by one segment every round trip time (RTT) as long as the network congestion is not detected. The implementation of the Slow Start and the Congestion Avoidance algorithms is built in a manner so that the increasing rate of the cwnd goes on until an indication for congestion occurrence is reached. At this point, the transmission rate should be slowed down to resolve the network congestion [16].

2.2 Fast Retransmit and Fast Recovery

During Congestion Avoidance, the reception of duplicate acknowledgement or the expiration of retransmission timer can implicitly signal the sender that the network congestion is occurred. So, the sender has to slow down its transmission rate. If the congestion was indicated by a timeout, the ssthresh is set to one half of the current congestion window and the congestion window is set to one segment and the sender enters into the Slow Start phase. If the congestion was indicated by duplicate acknowledgements, the TCP sender goes into the Fast Retransmit mode to retransmit what appears to be lost packet without waiting for the retransmission timer to expire. Then, the sender sets the ssthresh to half of the current congestion window and the new congestion window to the new ssthresh plus the number of received duplicate acknowledgements and enters into the Fast Recovery phase. Upon entering Fast Recovery, the sender continues to increase the congestion window by one segment for each subsequent duplicate ACK received. The intuition behind the Fast Recovery algorithm is that duplicate ACKs indicate the reception of some segments by the receiver, and thus can be used to trigger new segment transmissions. The sender transmits new segments if permitted by its congestion window. During Fast Recovery, the TCP NewReno distinguishes between a "partial" ACK and a "full" ACK. A full ACK acknowledges all segments that were outstanding at the start of fast recovery, while a partial ACK acknowledges some but not all of this outstanding data. On receiving a partial ACK, NewReno retransmits the segment next in sequence based on the partial ACK, and reduces the congestion window by one less than the number of segments acknowledged by the partial ACK. This window reduction, referred to as partial window deflation, allows the sender to transmit new segments in subsequent RTTs of Fast Recovery. The NewReno continues in Fast Recovery until all the packets which were outstanding during the start of the Fast Recovery have been acknowledged. On receiving a full ACK, the sender sets the congestion window (cwnd) to ssthresh, terminates Fast Recovery, and resumes Congestion Avoidance [17].

2.3 NewReno Implementation

The pseudo code of the TCP-NewReno is described in Table 1.

Table 1: Pseudo code of TCP-NewReno

Slow Start Algorithm:

```
Initial: cwnd = 1;
For (each packet Acked)
   cwnd++;
Until (congestion event, or, cwnd > ssthresh)
```

Congestion Avoidance Algorithm:

```
/* slow start is over and cwnd > ssthresh */
Every Ack:
    cwnd = cwnd + (1/cwnd)
    Until (Timeout or 3 DUPACKs)
```

Fast Retransmit Algorithm:

```
/* After receiving 3 DUPACKs */
Resend lost packet;
Invoke Fast Recovery algorithm
```

Fast Recovery Algorithm:

```
/* After fast retransmit; do not enter slow start */
ssthresh =  cwnd / 2;
cwnd = ssthresh + 3;
Each DACK received;
cwnd ++;
Send new packet if allow;
After receiving an Ack:
   If partial Ack;
      Stay in fast recovery;
      Retransmit next lost packet (one packet per RTT);
```

If Full Ack;

cwnd = ssthresh;

Exit fast recovery;

Invoke Congestion Avoidance Algorithm;

When Timeout:

ssthresh = cwnd /2;

cwnd = 1;

Invoke Slow Start Algorithm;

3. PROPOSED MECHANISM

As mentioned earlier, the problem with both Reno and NewReno is that within Fast Recovery algorithm, TCP halves its congestion window irrespective of the state of the network. Another problem with NewReno is that when there are no packets lost but packets are reordered by more than three duplicate acknowledgments, NewReno mistakenly enters Fast Recovery, and halves its congestion window [12-14]. Since the TCP's congestion window controls the number of packets that a TCP sender can send over the network at any time, hence, the process of setting the congestion window to half of its value make the TCP NewReno inefficient in terms of utilization of link capacity. The proposed mechanism avoids these problems by modifying the Fast Recovery algorithm of the TCP-NewReno. The basic idea is to adjust the congestion window (cwnd) of the TCP sender based on the level of congestion in the network so as to allow transferring more packets to the destination. The adjustment of congestion window has three main goals. The first one is to utilize the available network resources, the second is to minimize the probability of congestion, and the third is to offer a fair bandwidth among multiple connections. These goals could be achieved by adopting the congestion window size based on the network status.

3.1 Basic Idea

In TCP, all the transmitted packets make a round trip back from the sender to the receiver. The measurement of the Round Trip Time (RTT) between the two ends is the fundamental of timeout and the retransmission strategy of TCP. The RTT changes during the TCP connection as the network traffic load changes. In other words, the value of the RTT increases with the increasing of the network load. So, the RTT could be used to reflect the network status. The crux of idea is that, for a given network status, the modified mechanism determines the congestion degree in the network using the change in the Round Trip Time. In entering the Fast Recovery algorithm, it can detect the change in the *RTT* and decreases the congestion window (cwnd) by a value related to the increase in the *RTT*. The mechanism is that, the sender continuously monitors the *RTT* from the receiving acknowledgments and keeps up the measured values of the last few numbers of the *RTT*. When TCP enters the Fast Recovery algorithm, it computes the average value of RTT (RTT_{avg}) using the previously measured values of the RTT, as:

$$RTT_{avg} = \sum_{i=1}^{n} RTT_i / n$$

The sender then calculates the change in RTT (ΔRTT) by calculating the difference between the latest RTT (RTT_n) that calculated just right before detecting congestion and the average RTT (RTT_{avg}), as:

$$\Delta RTT = RTT_n - RTT_{avg}$$

Finally, the sender decreases its sending rate based on the change in RTT. In other words, the sender calculates the increasing factor (*Factor*) in the sending rate by dividing the current congestion window over the average RTT, as:

$$Factor = cwnd / RTT_{avg}$$

With increasing traffic load on the network, the cwnd will be decreased by an average number (Avg_{num}) determined as the product of the increasing factor and ΔRTT, as:

$$Avg_{num} = Factor * \Delta RTT$$

The new congestion window ($cwnd_n$) is thus determined as the maximum of two segments and the different between current congestion window and the average value (Avg_{num}), as:

$$cwnd_n = max \{2, (cwnd - Avg_{num})\}$$

3.2 Modified Fast Recovery Algorithm

During Congestion Avoidance, when the TCP sender receives three duplicate ACKs, it goes into the Fast Retransmit mode to retransmit what appears to be lost packet without waiting for the retransmission timer to expire. It then calculates the new congestion window ($cwnd_n$), sets the ssthresh to the maximum of two segments and $cwnd_n$, and sets the cwnd to the ssthresh value plus the number of received duplicate acknowledgements and continues with the Fast Recovery phase. The TCP sender increases the cwnd by one for each received duplicate acknowledgment, and sends new segment if allowed. With partial ACK, it retransmits the acknowledged segment and proceeds. With full ACK, it sets the cwnd to the ssthresh value and invokes the Congestion Avoidance algorithm. If the TCP sender detects losses by timeout expiration, it sets the ssthresh to the maximum of two segments and $cwnd_n$, and sets cwnd to one segment, and enters into the Slow Start algorithm. The modified Fast Recovery algorithm is described in Table 2.

Table 2: Modified Fast Recovery Algorithm

i-With 3 DUPACKs:

$cwnd_n$ = max {2, (cwnd − Avg_{num})};

ssthresh = max {2, $cwnd_n$};

cwnd = ssthresh + 3;

Each DACK received;

cwnd + +;

Send new packet if allow;

After partial Ack;

Stay in fast recovery;

Retransmit next lost packet (per RTT);

After Full Ack:

cwnd = ssthresh;

Exit Fast Recovery;

Invoke Congestion Avoidance Algorithm;

ii-When TimeOut:

$cwnd_n = \max \{2, (cwnd - Avg_{num})\}$;

$ssthresh = \max \{2, cwnd_n\}$;

cwnd = 1;

Invoke Slow Start Algorithm;

4. SIMULATION RESULTS

To test the behavior of the proposed congestion control mechanism, EnewReno, it is coded in C++ and incorporated into the Network Simulator NS-2 [18-20] to be used as the transmission control protocol in the simulated network. The EnewReno is formed by using the main three algorithms; Slow Start, Congestion Avoidance, and Fast Retransmit, in addition to the modified Fast Recovery algorithm.

Figure 1 shows the network topology that is used for the simulation. The topology has four TCP connections between 4 senders and 4 receivers. The network links are labeled with their bandwidth capacity and propagation delay.

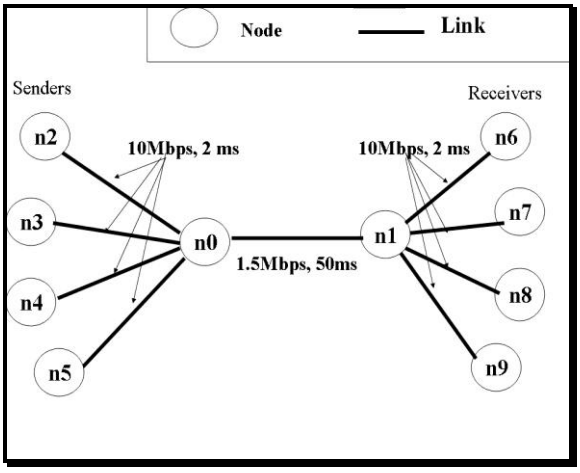

Figure 1: Network Topology

Figure 2 shows the network throughput for the TCP Reno, NewReno and EnewReno. From the figure, the TCP EnewReno provides higher throughput than both the TCP Reno and NewReno. This is because with the current implementation of TCP NewReno, acknowledgment causes the congestion window to additively increase, and packet losses cause the window to multiplicatively decrease, and that happens according to blind rate adaptation mechanism. Indeed, the TCP NewReno reduces its congestion window to half the current one disregarding to the degree of the congestion in the network. On the other hand, with EnewReno as soon as packet losses are detected, TCP EnewReno reduces the transmission rate and adjusts its congestion window according to the network status. So, it allows transferring of more packets to the destination.

It is clear from Figure 2 that the TCP Reno, NewReno, and EnewReno start up with the same throughput. This is because they have the same behavior during the Slow Start and the Congestion Avoidance phases. But, the EnewReno provides higher throughput than both Reno and NewReno in the Fast Retransmit and Fast Recovery phases. In case of packet loss, the throughput of Reno decreases because it reduces its window size to half of its value and with the first fresh acknowledge it gets out of Fast Recovery without recovering the multiple packet losses. TCP NewReno provides better throughput than Reno because it continues in Fast Recovery until all the packets which were outstanding during the start of the Fast Recovery have been acknowledged. Even through NewReno has better throughput than Reno, reducing the congestion window to half of its value affecting on the network throughput. With EnewReno, the congestion window decreases by the same level of congestion in the network. So, the TCP EnewReno can effectively transmit more packets and hence improves the network throughput.

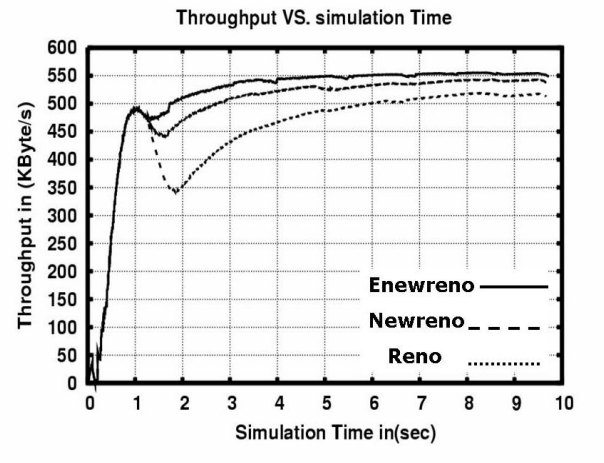

Figure 2: Throughput vs. Time

Figure 3 shows the behavior of congestion window (cwnd) for the congestion control mechanisms; TCP Reno, NewReno, and EnewReno. The figure shows the change of the congestion window with time. With TCP Reno and NewReno, when the packet losses are detected; these protocols set the window size to 50% of the current size. In TCP EnewReno, every time the packet losses are detected, it changes the window size by different value based on the network status.

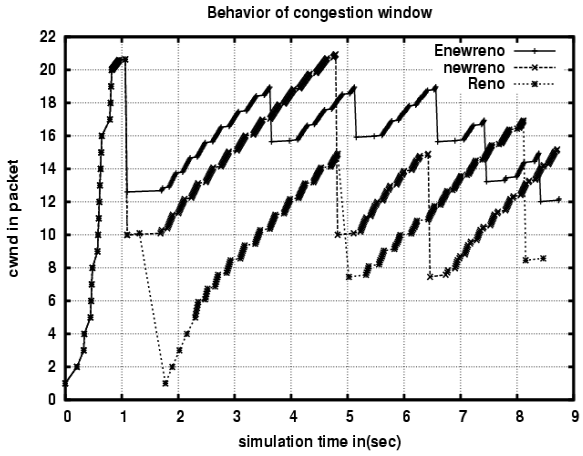

Figure 3: Behavior of the Congestion Window

Figure 4 shows the packet delay versus time for the TCP Reno, NewReno, and EnewReno. As shown in the figure, unlike NewReno which have higher delay than Reno,

EnewReno reduces the packet delay. This is because, during the Fast Recovery algorithm, TCP NewReno waits to recover all lost packets and send few new packets, but with EnewReno the congestion window changes based on the network status and it could send more new packets so it reduces the packet delay.

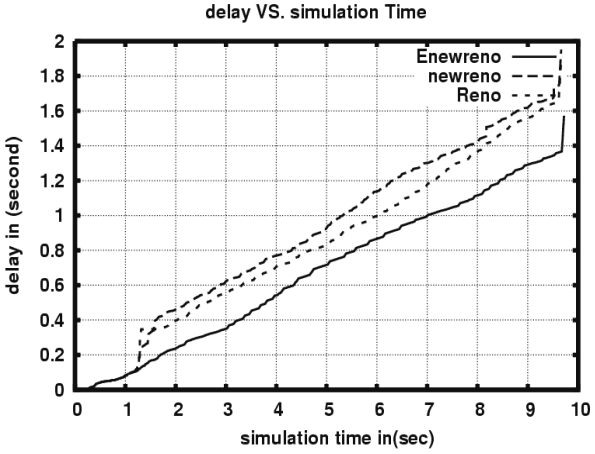

Figure 4: Delay vs. Time

Figure 5 shows the packet losses versus time for the TCP Reno, NewReno, and EnewReno. It is clear from the figure that the three protocols have the same behavior during Slow Start and Congestion Avoidance phases. However, with time, the rate of packet losses of EnewReno is increased. This is because, the congestion window size increases more than that of Reno and NewReno, and more packets are transmitted, so the probability of losses increased too. This appears in the figure, the EnewReno provides higher losses than NewReno. This behavior is noted also in the TCP NewReno compared with Reno.

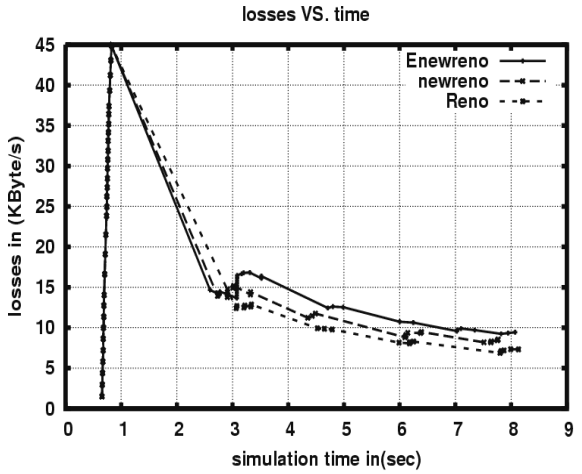

Figure 5: Losses vs. Time

5. CONCLUSIONS and FUTURE WORK

In this paper a modified Fast Recovery algorithm is proposed to improve the performance of the TCP NewReno. The mechanism is developed by adapting the congestion window of the TCP sender based on the level of congestion in the network. This level is determined by using the Round Trip Time (RTT) that represents as an indicator for the traffic loads on the network. The proposed mechanism is evaluated by using the network simulator NS2 and compared with both the TCP NewReno and the TCP Reno. The simulation results shown that, incorporating the modified Fast Recovery algorithm with the TCP NewReno improves its performance against both the throughput and the packet delay because of transferring more packets to the destination.

Although the additional modifications to the Fast Recovery algorithm improve the performance, the proposed mechanism, EnewReno, is inefficient in terms of packet losses, as shown in Figure 5. Additional enhancements should be considered in the future work to improve the EnewReno against packet losses.

6. REFERENCES

[1] J. Nagle, "Congestion control in IP/TCP Internetworks," Request for Comments (RFC) 896, Internet Engineering Task Force, January 1984.

[2] V. Jacobson, and M. J. Karels, "Congestion Avoidance and Control," Proceedings of ACM SIGCOMM, Vol.18 (4), pp. 314-329, August 1988.

[3] V. Jacobson, "Berkeley TCP Evolution from 4.3-Tahoe to 4.3 Reno," Proceedings of the 18th Internet Engineering Task Force, University of British Columbia, Vancouver, BC, Aug. 1990.

[4] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," RFC 2001, January 1997.

[5] A. Veres, M. Boda, "The Chaotic Nature of TCP Congestion Control," Proceedings of IEEE INFOCOM, pp.1715–1723, 2000.

[6] S. Floyd, "A Report on Some Recent Developments in TCP Congestion Control," IEEE Communications Magazine, pp. 84-90, April 2001.

[7] B. Kim, and J. Lee, "Retransmission loss recovery by duplicate acknowledgment counting", IEEE Communications Letters, Vol.8 (1), pp. 69-71, January 2004.

[8] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 3782, April 2004.

[9] A. Karnik, and A. Kumar, "Performance of TCP Congestion Control with Explicit Rate Feedback," IEEE/ACM Transactions on Networking, Vol. 13 (1), pp. 108-120, February 2005.

[10] D. Roman, K. Yevgeni, and H. Jarmo, "TCP NewReno Throughput in the Presence of Correlated Losses: The Slow-but-Steady Variant," IEEE International Conference on Computer Communications INFOCOM, pp. 1- 6, April 2006.

[11] M. Niels, B. Chadi, A. Konstantin, and A. Eitan, "Inter-protocol fairness between TCP NewReno and TCP Westwood," The 3rd EuroNGI Conference on Next Generation Internet Networks, Vol.1, pp. 21-23, May 2007.

[12] Hanaa A. Torkey, Gamal M. Attiya and I. Z. Morsi, "Performance Evaluation of End-to-End Congestion Control Protocols," Menoufia journal of Electronic Engineering Research (MJEER), Vol. 18, no. 2, pp. 99-118, July 2008.

[13] Cheng-Yuan Ho, Yaw-Chung Chen, Yi-Cheng Chan, Cheng-Yun Ho, "Fast retransmit and fast recovery schemes of transport protocols: A survey and taxonomy," Computer Networks, Vol. 52, pp.1308–1327, 2008.

[14] Kolawole I. Oyeyinka, Ayodeji O. Oluwatope, Adio. T. Akinwale, Olusegun Folorunso, Ganiyu A. Aderounmu, and Olatunde O. Abiona, "TCP Window Based Congestion Control Slow-Start Approach," Communications and Network, Vol. 3, pp.85-98, , May 2011.

[15] M. Miyake, and H. Inamura, "TCP Enhancement Using Recovery of Lost Retransmissions for NewReno TCP," Transactions of Information Processing Society Journal, Vol. 46 (9), pp. 2185-2195, September 2005.

[16] D. A. Lima, M. da Fonseca, and N. De Rezende, ”On the Performance of TCP Loss Recovery Mechanisms”, IEEE International Conference on Communications, Vol.3, pp. 1812-1816, May 2003.

[17] N. Parvez, A. Mahanti, and C. Williamson, "TCP NewReno: Slow-but- Steady or Impatient?" IEEE International Communications Conference, Vol.3 (2), pp. 716-722, June 2006.

[18] S. McCanne and S. Floyd, "ns Network Simulator", http://www.isi.edu/nsnam/ns.

[19] L. Breslau, et al., "Advanced in Network Simulation," IEEE Computer, Vol. 33, No. 5, pp. 59-67, May 2000.

[20] K. Fall and K. Varadhan, "The ns Manual," UC Berkeley, LBL, USC/ISI, and Xerox PARC, December 2006.