New Strategy for Congestion Control based on Dynamic Adjustment of Congestion Window

Gamal Attiya

Department of Computer Science and Engineering,
Faculty of Electronic Engineering,
Minoufiya University, Egypt

Abstract
This paper presents a new mechanism for the end-to-end congestion control, called EnewReno. The proposed mechanism is based on the enhancement of both the congestion avoidance and the fast recovery algorithms of the TCP NewReno so as to improve its performance. The basic idea of the proposed mechanism is to adjust the congestion window of the TCP sender dynamically based on the level of congestion in the network so as to allow transferring more packets to the destination. The performance of the proposed mechanism is evaluated and compared with the most recent mechanisms by simulation studies using the well known Network Simulator NS-2 and the realistic topology generator GT-ITM.

Keywords: *TCP, Congestion Control, Congestion Avoidance, Fast Recovery algorithm, Network Protocols.*

1. Introduction

Today, the majority of traffic over the Internet is carried out by the Transmission Control Protocol (TCP). TCP is a window based reliable data transfer protocol providing data transport between two end hosts of a connection. The original TCP is officially defined in [1]. It has a simple sliding window flow control mechanism. The essential strategy of TCP is sending packets to a network without a reservation and then reacting to observable events.

Over years, the use of Internet services has experienced dramatic growth and Internet applications have evolved from standard document retrieval functionality to multimedia services. The rapid growth of the Internet and the increasing of the traffic demand lied to a serious problem called congestion collapse [2]. Internet congestion occurs when the aggregate demand for resources exceeds the available capacity of the network. This problem leads to unacceptable long response times particularly for real-time applications. Indeed, 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 untimely death. Congestion control is thus required to prevent congestion collapse in the network. Without congestion control, a sending node could be busy transmitting packets that may be dropped later. Therefore, one of the keys to the success of the Internet is relying on using congestion control mechanisms.

After observing a series of congestion collapses, continuous efforts are being done to avoid congestion. The most essential is the congestion control mechanism provided by Jacobson in 1988 [3]. This mechanism is called TCP Tahoe. It includes three algorithms; namely; slow start, congestion avoidance, and fast retransmit. In 1990, a new TCP version called TCP Reno was developed by adding fast recovery algorithm to Tahoe [4]. TCP Reno can be thought of as 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 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. In [5], the authors have shown that 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 Multiplicative Decrease (AIMD) strategy of TCP Reno leads to periodic oscillations in the aspects of the congestion window size (cwnd), round-trip delay, and queue length of the bottleneck node. The oscillation may induce chaotic behavior in the network, thereby adversely affecting overall network performance.

Several proposals have been put forward to improve TCP congestion control and to alleviate the performance degradation problem of packet loss. Some researchers attempted to refine the fast retransmit and fast recovery algorithms while other researchers attempted to refine the slow start and congestion avoidance algorithms. New proposals included TCP NewReno [6], Forward Acknowledgment (FACK) [7], Selective Acknowledgment (SACK) [8], dynamic recovery [9], an Extension to the SACK Option (D–SACK) [10], TCP with Faster Recovery (FR–TCP) [11], Reordering–Robust TCP (RR–TCP) [12], Duplicate Acknowledgment Counting (DAC) [13], and

TCP SACK+ [14] and TCP Vegas [15, 16]. The TCP New-Reno, FACK, and SACK are based on TCP Reno, while dynamic recovery, FR–TCP and DAC are based on TCP NewReno; and TCP DSACK, RR–TCP, and TCP SACK+ operate at the sender of SACK.

Comparative studies between the different proposals for congestion control are presented in [17-19]. The comparative studies show that the TCP NewReno provides better performance than previous TCP variants. However, it has been found that the TCP NewReno is inefficient in terms of utilization of link capacity and unfair in its throughput performance [20, 21]. The problem with NewReno is that, it halves its congestion window, as long as a packet loss is detected, irrespective of the state of the network. Another problem 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 [22, 23].

This paper presents a new mechanism for the end-to-end congestion control, called EnewReno. The proposed mechanism is based on the enhancement of both the congestion avoidance and the fast recovery algorithms of the current TCP NewReno. The main idea is to adjust the congestion window at the TCP sender dynamically according to the state of the network (i.e., the network load or the level of congestion at the network). The performance of the proposed mechanism is evaluated and compared with the most recent mechanisms by simulation studies using the well known Network Simulator NS-2 [24] and the realistic topology generator GT-ITM [25, 26].

The rest of this paper is organized as follows; Section 2 presents an overview of the widespread TCP mechanisms; TCP Tahoe, Reno, NewReno, Sack, and Vegas. Section 3 presents the weakness of TCP variants while the proposed mechanism is described in section 3. The simulation results and discussions are given in Section 5. Finally, the conclusions and future work are given in section 6.

2. TCP Variants

TCP has been refined several times aiming to improve its performance and ensure the internet stability. Among these mechanisms, which are of interest, are TCP Tahoe [3], Reno [4], NewReno [6], Sack [8], and Vegas [15, 16].

2.1 TCP Tahoe

TCP Tahoe is the first implementation that handles congestion control. It was released in 1988 by V. Jacobson [3]. TCP Tahoe controls congestion by adjusting its window size additively to increase and multiplicatively to decrease (AIMD). It uses three algorithms, namely; slow-start, congestion avoidance and fast retransmit. During slow-start, the congestion window (cwnd) increases exponentially by one for each acknowledgement received until it reaches the slow-start threshold (ssthresh), and during congestion avoidance the congestion window increases linearly by one per round trip time (RTT). The TCP sender goes into the fast retransmit mode when it receives 3 duplicate acknowledgements. During fast retransmit, the sender retransmits the lost packet and enters into the slow-start phase by setting the ssthresh to the half of the current congestion window and setting congestion window to 1. Whenever a timeout occur, the ssthresh is set to one half of the current congestion window and the congestion window is set to one and the sender enters into the slow-start phase. The problem of Tahoe is that, when the loss is due to sporadic channel error, Tahoe forgets all outstanding data transmitted earlier and switching to slow-start mode which causes the throughput to fall.

2.2 TCP Reno

TCP Reno [4] is similar to TCP Tahoe except that it includes the fast recovery algorithm for a single packet loss. When the TCP sender receives duplicate acknowledgements, instead of switching to slow-start after fast retransmit, TCP Reno enters into fast recovery. During fast recovery, the sender sets ssthresh to the half of the congestion window and the new congestion window to the new ssthresh plus the number of received duplicate acknowledgements. Each new duplicate acknowledgement increases the congestion window size by one. TCP Reno remains in fast recovery until the lost packet which triggered the fast retransmit has been acknowledged. When the sender receives new acknowledgement(s), it exits fast recovery and resets the congestion window to the ssthresh and thereby moves into congestion avoidance. So, the fast recovery mechanism keeps the average congestion window size high, resulting in better throughput performance compared to TCP Tahoe. Although TCP Reno works fine for single packet loss, in case of multiple losses from the same transmission window, the performance suffers, since it exits fast recovery and enters into it again in a repeated fashion or goes to timeout. In case of timeout, Reno works as Tahoe by setting its congestion window to one packet and entering into slow start mode.

2.3 TCP NewReno

TCP NewReno uses an augmented fast recovery mechanism to eliminate the Reno’s wait for a retransmit time-out whenever multiple packets are lost from the same transmission window [6]. In other words, TCP NewReno modifies the sender's behavior during fast recovery, where, it continues in fast recovery until all the packets which were outstanding during the start of the fast recovery have been acknowledged. This strategy helps to combat multiple losses without entering into fast recovery multiple times or causing timeout. In this case, a partial acknowledgement is considered as an indication that the packet following the acknowledged one has been dropped from the same transmission window, and therefore, TCP NewReno immediately retransmits the other lost packet indicated by the partial acknowledgement and remains in fast recovery. It exits fast recovery when all data in the window is acknowledged. One challenge with the NewReno mechanism is its inability to detect other lost packets until the ACK for the first retransmitted packet was received. In other words, it can not handle retransmission of more than one lost packet per RTT because it takes one round trip time (RTT) to detect each lost segment and to retransmit it. This implies that NewReno suffers from the fact that the detection of each packet loss takes one RTT [23].

2.4 TCP SACk

TCP SACK (TCP with Selective Acknowledgement) is an extension of the TCP Reno. It only modifies the fast recovery algorithm of Reno keeping the other algorithms unchanged [8]. Similar to NewReno, TCP SACK handles multiple packet losses from the same window but it has a better estimation capability for the number of outstanding packets. In SACK, instead of cumulative acknowledgement of packets as contained in Tahoe, Reno and NewReno, packets are acknowledged selectively. Where, the receiver can inform the sender about all packets that have arrived successfully. This enables the sender to figure out which packets have been acknowledged and which ones are still outstanding. So, the sender need retransmit only the packets that have actually been lost without needing to retransmit packets that have already been received successfully. To keep track of the acknowledged and lost packets, the sender maintains a data structure called scoreboard. Whenever the sender is allowed to transmit, it consults the scoreboard and transmits the missing packets. If there is no missing packet to retransmit, it transmits new packets. When a retransmitted packet is dropped, the sender detects it by a retransmit timeout. In case of timeout, it retransmits the packet and enters into the slow start phase. One major drawback of the TCP SACK is the relative difficulty in implementation of selective acknowledgement [23].

2.5 TCP Vegas

TCP Vegas is a proactive congestion control mechanism in which network congestion is predicted based on packet delay rather than packet loss [15, 16]. Where, it detects congestion at an incipient stage based on increasing Round Trip Time (RTT) values of the packets in the connection unlike other flavors like TCP Tahoe, Reno, and NewReno, which detect congestion only after it has actually happened via packet drops. In other words, the algorithm emphasizes packet delay as a signal to help determine the rate at which to send packets. It depends heavily on accurate calculation of the Base RTT values of the packets in the connection. If it is too small then throughput of the connection will be less than the bandwidth available, while if the value is too large then it will overrun the connection.

Vegas introduces three changes to Reno, confined to the sending side; (i) Modified slow start algorithm: the sender tries finding correct window size without causing a loss, where, it tries to find a connection's available bandwidth that does not incur packet losses. (ii) Modified congestion avoidance: The sender doesn't wait for a timeout but updates its congestion window based on end-to-end delay instead of using packet-loss as the window update parameter. It determines congestion states using the sending rate. If there is a decrease in calculated rate of transmission, as a result of large queue in the link, it reduces its window. When the sending rate increases, the window size also increases. (iii) New retransmission algorithm: Vegas extended Reno retransmission algorithm by a fine grained timer expiry calculation mechanism to support early switching to fast retransmit. This is done by monitoring how long it took each ACK to get back to the sender. For this, the sender reads and records the system clock each time a packet is transmitted. When an acknowledgement arrives, it reads the clock again and calculates the fine grained RTT. TCP Vegas uses this fine-grained RTT estimate to calculate Retransmission Time Outs (RTO). Whenever a duplicate ACK is received, the sender checks the Vegas expiry (Timeout) and if Vegas expiry occurs, the sender switches to fast retransmit algorithm to retransmit the packet without waiting for 3 duplicate ACK or a time out as in Reno [15]. Similar to the other TCP variants, it switches to fast retransmit when it receives number of duplicate acknowledgements. Also, the sender switches to slow-start whenever the usual timeout occurs. The problem of Vegas is that, when Vegas is interoperated with other versions like Reno, performance of Vegas degrades because Vegas reduces its sending rate before Reno as it detects congestion early and hence gives greater bandwidth to coexisting TCP Reno flows.

3. Weakness of TCP Variants

The fundamental design philosophy of the most TCP congestion control mechanisms is that; (i) they are implemented end-to-end; (ii) they consider the network as a "black-box" and the congestion is detected by packet loss via duplicate acknowledgments or timeouts; and (iii) the sending rate at the sender is controlled indirectly by adjusting the congestion window. The importance of the end-to-end principle can never be overemphasized, because, this principle guarantees the delivery of data over any kind of heterogeneous network. However, considering the network as black-box, allow the TCP sender to update its congestion window blindly regardless to the current state of the network. This strategy makes the TCP variants inefficient in term of network utilization.

Briefly, the most important concept of TCP congestion control is the congestion window (cwnd) that determines the amount of data to be sent. Since the TCP sender does not receive any explicit congestion feedback from the network, hence, to determine the rate at which the source can transmit data, it must probe the path by progressively increasing the input load (through slow start and congestion avoidance) until implicit feedback signals such as timeouts or duplicate acknowledgments are arrived to indicate that the network capacity has been reached. The behavior of TCP variants, during slow start phase, makes them very expensive in terms of losses. This is because, the sender doubles the size of the cwnd every RTT while there are no losses - which is equivalent to doubling the attempted throughput every RTT. But, when it finally overruns the connection bandwidth, we can expect losses in the order of half the current congestion window. Also, the behavior of TCP variants, during congestion avoidance phase, degrades throughput because the linear increase of the cwnd by constant value makes the sending rate to be less than available bandwidth of the TCP connection. Another problem with both Reno and NewReno is that, within fast recovery algorithm, TCP sender halves its congestion window, as long as the network congestion is detected, irrespective of the state of the network. Indeed, 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. This strategy makes TCP NewReno inefficient in terms of network utilization.

4. Proposed Mechanism

The proposed mechanism is structurally similar to the TCP NewReno mechanism with two crucial differences in both the congestion avoidance and the fast recovery algorithms.

4.1 Principle Idea

The principle idea of the proposed mechanism is to adjust congestion window, and hence the transmission rate, at the TCP sender dynamically based on the network status. In other words, the size of the congestion window is dynamically calculated based on the level of congestion in the network instead of updating (increasing/decreasing) the congestion window blindly regardless the current load at the network. The main goal is to maintain the "right" amount of extra data in the network to improve the overall network performance. Obviously, if a connection is sending too much extra data, it will cause congestion; if it's sending too little extra data, it cannot respond rapidly enough to transient increase in the available bandwidth.

4.2 Enhanced Congestion Avoidance Algorithm

During congestion avoidance phase, most of the current congestion control mechanisms "blindly" increase the congestion window size linearly by constant value as long as no losses are detected. The key idea of the enhanced congestion avoidance algorithm is to adjust the congestion window at the TCP sender dynamically according to the available connection capacity at any time. In other words, the strategy is to adjust the source's sending rate based on the network load. To do so, the behavior of the TCP congestion control at the sender site can be viewed as cascaded control system with two feedback loops, as shown in Figure 1. The inner-loop determines when each packet is transmitted while the outer-loop reflects how the congestion window size could be changed, and hence the sending rate. Simply, the inner-loop defines when to send new data and the outer-loop defines how much data will be send. The feedback signals are the arrival of ACKs and the window size. The outer-loop adjusts the window size based on the received ACKs and other feedback information from the network such as losses indication. As shown in the figure, the inner-loop is controlled by the receiving acknowledgements while the outer-loop is controlled by the Observer that uses TCP timers and ACKs to adjust the window size (transfer rate) at the sender. This explains the dynamic relation between the window size, the sending rate and queue size. The question now is how to estimate the transfer rate at the TCP sender and how the transfer rate estimation is used to adjust cwnd during the congestion avoidance algorithm?

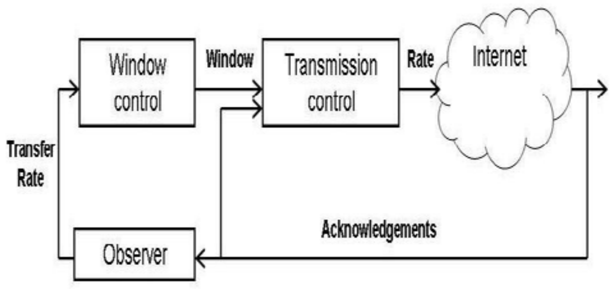


Figure 1: Dynamic Adjustment of Window Size

a) Estimating Transfer Rate

The idea of estimating transfer rate is that the sender monitors the acknowledgment stream that it receives from the destination and estimates the data transfer rate currently available by the TCP connection. This may be done by calculating the rate at which data was delivered to the destination. More precisely, the TCP sender monitors ACKs and uses the information in the ACKs, and the rate at which the ACKs are received to determine the amount of data delivered to the destination. In other words, since the ACK received by the source conveys the information that an amount of data corresponding to a specific transmitted packets was delivered to the destination. Then, if the transmission process is not affected by losses, the required estimation of the transfer rate can be calculated simply by dividing the amount of the delivered data by the acknowledgement interval time. That is, the transfer rate currently (*TRc*) achieved by the TCP connection is:

$$TR_c = ACKed_{data} \,/\, Time_{interval} \qquad (1)$$

Where, $ACKed_{data}$ is the amount of the data delivered to the destination, and $Time_{interval}$ is the acknowledgement interval time which is the difference between the last ACK received time and the current one. As the ACK reception rate depend on the network status and the transfer rate continuously estimated every time an ACK received, then the change of transfer rate will also depend on the network status. In other words, since the source performs an end-to-end estimate of the transfer rate achieved along a TCP connection, the greater the bandwidth of a given path, the higher the data transfer rate.

b) Adjustment of Congestion Window

As soon as the cwnd crosses the ssthresh, TCP goes into congestion avoidance phase. In this phase, for each ACK received, the sender estimates the data transfer rate currently available by the connection and uses the difference between the current transfer rate estimated and the last one to properly set the congestion window size.
Let ΔTR to be the difference between the current transfer rate (TR_c) and the previous transfer rate (TR_p), as:

$$\Delta TR = TR_c - TR_p \qquad (2)$$

Hence, an incremental (increment) value is calculated to be added to the congestion window, as:

$$\text{Increment} = \begin{cases} 2 * (1/\text{cwnd}); & \text{if } \Delta TR > \alpha_e \\ 1/\text{cwnd}; & \text{if } \Delta TR > \beta_e \\ 0; & \text{otherwise} \end{cases} \qquad (3)$$

Where, α_e and β_e are two thresholds such that $\alpha_e > \beta_e$. At the start of a connection, α_e is set to 3 and β_e is set to 1. These values are then changed dynamically based on the network status.

The incremental value is used to update the cwnd, as:

$$cwnd_n = cwnd + increment \qquad (4)$$

It could be noted here that the cwnd size increased by two segment each RTT if the current Transfer Rate (TR) sufficiently higher than the last one estimated, i.e., ΔTR > α_e or by one segment per RTT if that difference is not high enough. Otherwise, if ΔTR < β_e, the cwnd will be unchanged. This is because, the network is saturated and the sending rate should not increased than the current.

c) Algorithm Description.

Figure 2 shows the flow chart of the Enhanced Congestion Avoidance algorithm.

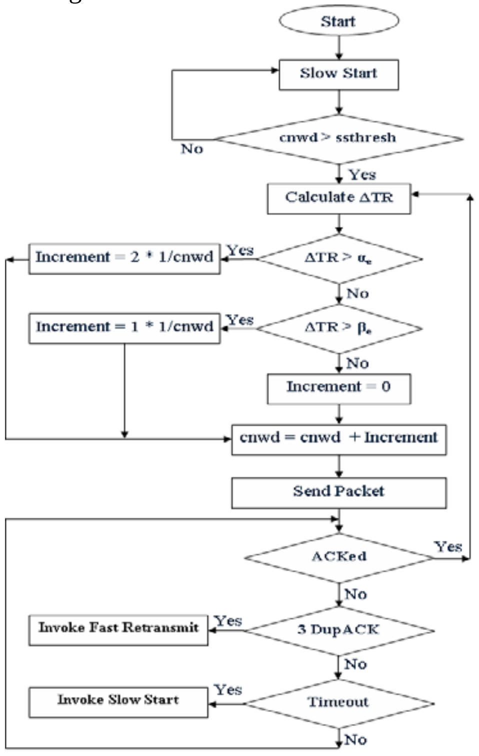


Figure 2: Flowchart of Enhanced Congestion Avoidance Algorithm

4.3 Enhanced Fast Recovery Algorithm

The problem of the current fast recovery algorithm is that, the TCP sender halves its congestion window, as soon as congestion detection, irrespective to the state of the network. This mechanism makes the current TCP variants inefficient in terms of network utilization because the congestion window controls the number of packets that a TCP sender can send over the network at any time. The crux of idea of the enhanced fast recovery is to decrease the congestion window based on the state of the network so as to allow transferring more packets to the destination.

The sender can determine the congestion degree in the network using the change in the Round Trip Time (RTT) and hence changes the congestion window accordingly. Since all packets make a round trip from the sender to the receiver and back to the sender, the RTT will change as the network traffic load changes. The value of RTT increases with the increasing of the network load. So, the RTT could be used to reflect the network status. Therefore, in entering the fast recovery algorithm, the sender can detect the change in the RTT and decrease the congestion window by a value related to the increase in the RTT.

a) Estimating Change in Round Trip Time

The sender continuously monitors the RTT and keeps up the last N values of RTTs. In entering fast recovery phase, the sender uses the queued values of the RTTs to compute the Average Round Trip Time (RTT_{avg}), as:

$$RTT_{avg} = \sum_{i=1}^{N} RTT_i \ / N \qquad (5)$$

Then, the sender calculates the change in RTT (ΔRTT) as the difference between the latest RTT (RTT_n), just right before detecting congestion, and the average RTT, as:

$$\Delta RTT = RTT_n - RTT_{avg} \qquad (6)$$

b) Updating Congestion Window

The sender uses the change in the RTT to update the congestion window by a value related to the ΔRTT. That is, the sender first calculates the increasing factor (INC_f) according to the change in RTT, as:

$$INC_f = (cnwd / RTT_n) * \Delta RTT \qquad (7)$$

The new congestion window ($cwnd_n$) is then determined as the maximum of two segments and the difference between the current congestion window (cwnd) and the increasing factor (INC_f). That is:

$$cwnd_n = max\ \{2, (cwnd - INC_f)\} \qquad (8)$$

c) Algorithm Description

In entering the fast recovery algorithm, the sender first determines the $cwnd_n$ as in Eq. 8, and sets the ssthresh to the maximum of $cwnd_n$ and two segments. The sender then sets cwnd to the value of the ssthresh plus 3, the number of received duplicate acknowledgements, and continues with fast recovery. The sender increases the cwnd by one for each duplicate acknowledgment received and send new segment if allowed. With partial ACK, the sender retransmits the segment that follows the Acked one and proceeds. With full ACK, it sets the cwnd to the ssthresh and invokes the fast recovery algorithm. If the sender detects losses by timeout expiration, it will set ssthresh to the maximum of $cwnd_n$ and two segments, and set the cwnd to one, and then goes into the slow start algorithm.

4.4 EnewReno Mechanism Descriptions

This section describes the proposed congestion control mechanism; EnewReno. The EnewReno uses the four algorithms: slow start, enhanced congestion avoidance, fast retransmit, and enhanced fast recovery. In the proposed algorithm, the fundamental of the slow start algorithm is unchanged. That is, when a new connection is established the congestion window size (cwnd) is initialized to one segment and the value of cwnd is updated to cwnd + 1 for each ACK received. As soon as the cwnd exceeds the ssthresh, the enhanced congestion avoidance algorithm is invoked. During this phase, the sender estimates the ΔTR and compares its value to αe and βe, then makes a decision of how to increase the congestion window size. The sender continues increasing its sending rate (cwnd) with each ACK received during this phase until congestion indication via timeouts or duplicate acknowledgements (DUPACKs). When the sender receives 3 DUPACKs, it enters into fast retransmit to resend the lost segments, and then invoke the enhanced fast recovery algorithm. When the sender receives new acknowledgement, it will exit that phase putting its cwnd to ssthresh and goes into congestion avoidance algorithm. Table 1 shows pseudo code the EnewReno congestion control mechanisms.

Table 1: pseudo code the EnewReno mechanisms

Slow Start Algorithm:
```
    Initial: cwnd = 1;
    For (each packet Acked)
    cwnd++;
    Until (congestion event, or, cwnd > ssthresh)
```
Enhanced Congestion Avoidance:
```
    /* slow start is over and cwnd > ssthresh */
    Every Ack:
        $\Delta TR = TR_c - TR_p$
        Calculate increment value;   (Eq., 3)
        cwnd = cwnd + increment
        Until (timeout or 3 DUPACKs)
```
Fast Retransmit Algorithm:
```
    After receive 3 DUPACKs
    Resend lost packet;
    Invoke Enhanced Fast Recovery algorithm
```
Enhanced Fast Recovery Algorithm:
```
    i-With 3 DUPACKs:
        $cwnd_n$ = MAX {2, (cwnd - $INC_f$)};
        ssthresh = MAX {2, $cwnd_n$};
        cwnd = ssthresh +3;
        Each DACK received;
        cwnd ++;
        Send new packet if allow;
        After Full Ack:
        cwnd = ssthresh;
        Invoke Congestion Avoidance Algorithm;

    ii-When TimeOut:
        $cwnd_n$ = MAX {2, (cwnd - $INC_f$)};
        ssthresh = MAX {2, $cwnd_n$};
        cwnd = 1;
        Invoke Slow Start Algorithm;
```

5. Simulation Results and Discussion

In this study, the performance of the proposed mechanism is evaluated by using the well-known network simulator NS-2. The evaluation is done on a simulation scenario of the US AT&T network topology, as shown in Figure 3. This topology is generated by using the GT-ITM realistic topology generator [25]. It has been proven that the created topology is similar to the real US AT&T continental IP backbone network by 86.66% [26].

In order to get a proper estimation for the steady state performance, the simulation time should be long enough. Furthermore, a number of performance matrices should be considered to show the benefits of using TCP EnewReno over other mechanisms. In this study, a simulation time of 40 seconds is used and the collection of data is started after 2 seconds from the beginning of the simulation to get over the transient phase. In addition, a number of performance matrices such as the instant and the aggregate throughput, the changes of delay with the time, and the packet losses, are studied to show the benefits of using EnewReno over the existing mechanisms NewReno and Vegas.

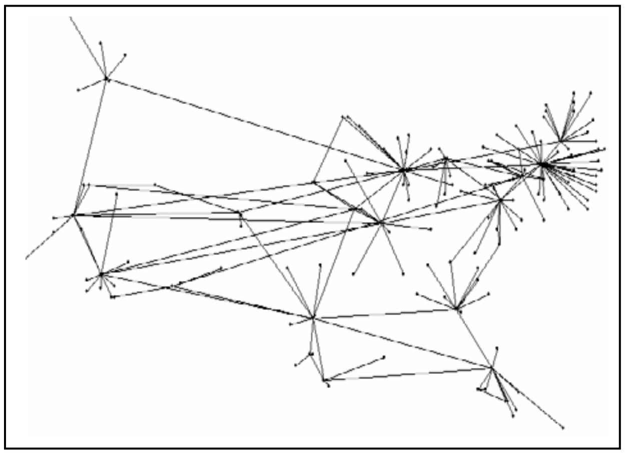
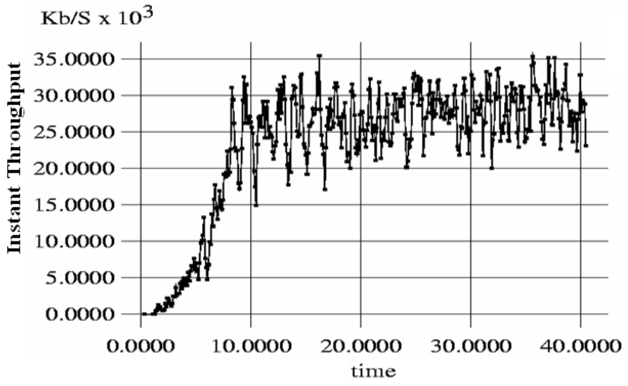


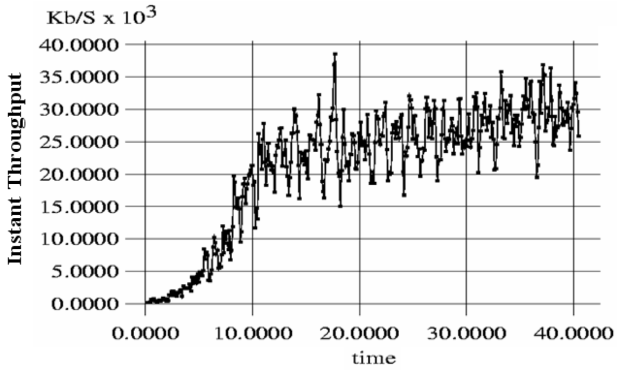
Figure 3: AT&T Network Topology [26].

5.1 Throughput

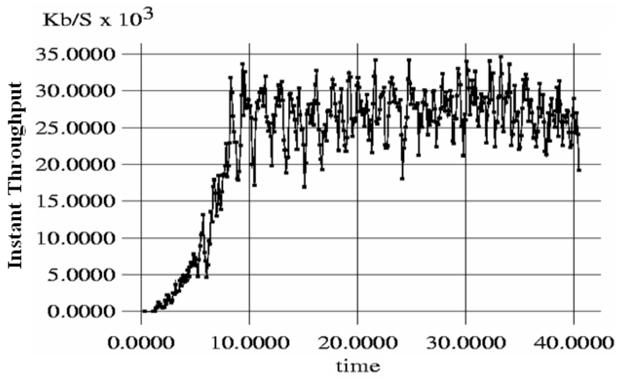
Figure 4 shows the instant throughput, calculated every 0.1 second, for the mechanisms: NewReno, Vegas, and EnewReno. Form the figure, the throughput achieved by the EnewReno is higher than that of Vegas and NewReno. This is because, during the fast recovery algorithm, NewReno waits to recover all lost packets and send few new packets, but with EnewReno the congestion window decreased by the degree the network load so it could send more new packets depend on the network status.



(a) TCP NewReno



(b) TCP Vegas



(c) TCP EnewReno

Figure 4: Instant Throughput for TCP NewReno, Vegas and EnewReno

Figure 5 shows the aggregate throughput for TCP EnewReno, NewReno and Vegas. The figure shows that, the TCP EnewReno provides higher throughput than the other mechanisms. This is because, the NewReno adapts congestion window according to blind rate adaptation mechanism regardless to the current load at the network, i.e., the ACK causes the congestion window to increase and packet losses cause the window to decrease. While, the TCP EnewReno adapts its sending rate by adjusting the congestion window based on the current network load. From the figure, before 10 second, the aggregate throughput for NewReno and EnewReno is similar. This is because they have the same behavior in the slow start phase. But, after 10 second, the TCP EnewReno provides higher throughput than the other mechanisms because of the behavior of EnewReno during the congestion avoidance and the fast recovery phases. In the congestion avoidance phase, EnewReno adapts window size according to the network status (2 packets per RTT or 1 packet per RTT or zero packets per RTT), while NewReno continuously increases its window size one packet per RTT regardless to the current load at the network, and TCP Vegas increases, stops, or decreases its window. During the fast recovery phase, the EnewReno reduces the sending rate and adjusts its congestion window according to the network status as soon as a packet loss is detected. While, TCP NewReno and Vegas reduce their congestion window to half the current one, regardless to the network load.

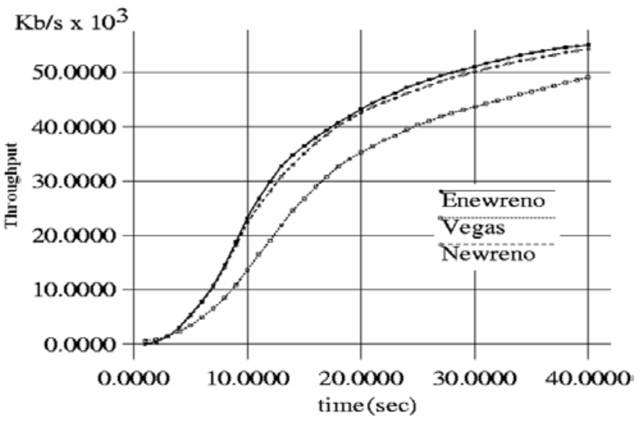


Figure 5: Aggregate Throughput of different algorithms

5.2 Delay

Propagation and queuing delays are the primary sources of communication delay in the network. The propagation delay is a link characteristic, while, the queuing delay is a control flow affected delay. Hence, one of the goals of a congestion control mechanisms is to adjust the sending rate in order to minimize the queuing delay. Figure 6 shows the packet delay verses time for EnewReno, NewReno, and Vegas. As shown in the figure, between 10 and 24 sec, the TCP Vegas provides less delay than the other variants. This is because, at start, Vegas increases the sending rate very carefully, where it sets the ssthresh to two packets only. While, TCP EnewReno tries to utilize the available network resources by increasing the sending rate. However, during the congestion avoidance and fast recovery phases, EnewReno adapts the transmission rate to get better performance behavior, and that is clear between time 25 and 36 sec, the time delay get less than Vegas.

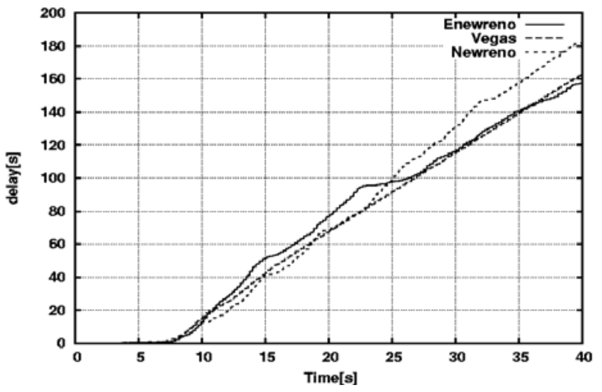


Figure 6: Delay versus Time

5.3 Packet Losses

Figure 7 shows the number packets losses versus time for the TCP EnewReno, NewReno, and Vegas. From the figure, between 6 and 10 sec, EnewReno and NewReno have the same behavior of losses, because they have the same slow start algorithm, but they provide more losses than Vegas. However, during congestion avoidance and fast recovery phases, EnewReno increases or decreases its congestion window size based on the change in the network load, so the probability of losses decreased. As shown in the figure, EnewReno get fewer packet losses than NewReno and Vegas.

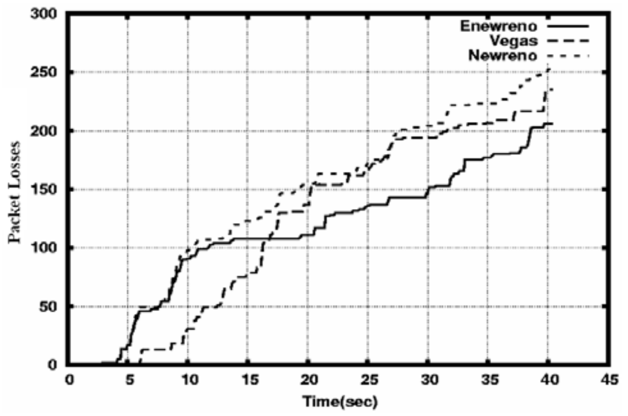


Figure 7: Packet Losses versus Time

5.4 Resource Utilization

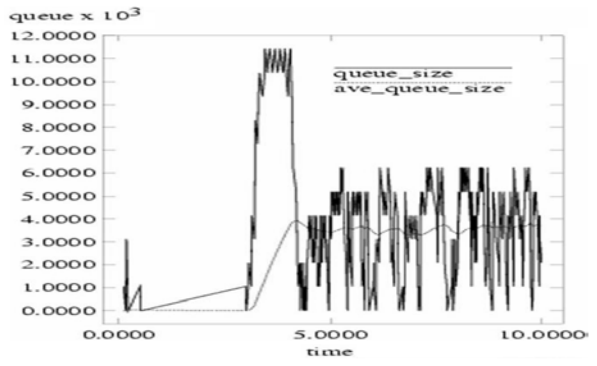
In this section, the resource utilization is studied in terms of how the congestion control mechanisms impact the queue size at the intermediate routers. The objective is to achieve high utilization of the link as well as maintain the queue size close to a reasonably reference value via controlling the sending rate at the sender. Figure 8 shows the behavior of queue size for TCP variants, EnewReno, Vegas, and NewReno considering the Random Early Detection (RED) discipline as a buffer management mechanism. The RED management discipline is designed to cooperate with the TCP congestion control mechanism. It tries to manage the average queue size by monitoring the average buffer occupancy at the router, and setting its packet drop probability as a function of buffer occupancy. Keeping the average queue size low, burst dropping can be avoided even when packets from the same connection continuously arrive.

The simulation results in Figure 8 are taken over a portion of time (10 sec) to have a range in the graph showing exactly how the queue size is changed. From the figure, EnewReno provides higher resource utilization, in form of maintaining effective queue size value, than the other TCP variants. This is because EnewReno tries to fully utilize the network resources by adapting the sending rate based on the network load. This behavior clearly explains the results in Figure 5 for the throughput and why Vegas has lower throughput than EnewReno and NewReno. Also, this behavior explains the results in Figure 6 for the delay and why EnewReno provides higher delay at the start of the connection transmission.

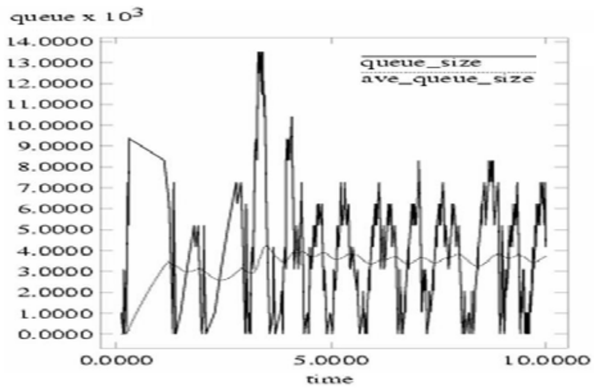
6. Conclusions and Future Work

In this paper a new mechanism, called EnewReno, is developed for congestion avoidance. The proposed mechanism is based on the enhancement of both the congestion avoidance and the fast recovery algorithms of the current TCP NewReno. The additional modifications are generally developed by adjusting the congestion window dynamically based on the network status. During congestion avoidance, the congestion window is not increased linearly as the other TCP variants but increased by an incremental value based on the degree of the network congestion in order to fully utilize the network resources efficiently. Also, during fast recovery, the congestion window is not decreased to half of its value as the other TCP variants but decreased by a value based on the change in round trip time so as to allow transferring more packets to the destination. The simulation results show that, the proposed mechanism provides better performance than previous TCP variants in terms of throughput, efficient utilization of the network resources, less packet losses and delay.

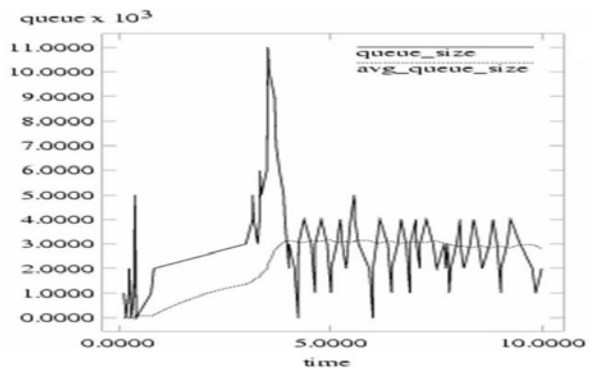
The EnewReno mechanism has been verified to work well in wired networks. So, the further work should focus on whether EnewReno performs well in wireless networks. Also, in the future studies, it is intended to evaluate the proposed mechanism against fairness when interact with the UDP.



(a) TCP EnewReno



(b) TCP NewReno



(c) TCP Vegas

Figure 8: RED Queue Size Behavior for TCP variants

References

[1] J. Postel, “Transmission Control Protocol,” IETF RFC 793, September 1981.
[2] J. Nagle, “Congestion control in IP/TCP Internetworks,” Request for Comments (RFC) 896, Internet Engineering Task Force, January 1984.
[3] V. Jacobson, “Congestion Avoidance and Control,” ACM SIGCOMM Computer Communication Review, Vol. 18, No. 4, pp. 314-329, August 1988.
[4] 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.
[5] A. Veres, M. Boda, "The Chaotic Nature of TCP Congestion Control," Proceedings of IEEE INFOCOM, pp.1715–1723, 2000.
[6] J. Hoe, “Start-up Dynamics of TCP’s Congestion Control and Avoidance Schemes,” Master Theses, Massachusetts Institute of Technology, 1995.
[7] M. Mathis, J. Mahdavi, "Forward acknowledgement: refining TCP congestion control," Proceedings of ACM SIGCOMM, pp. 181-191, 1996.
[8] M. Mathis, J. Mahdavi, S. Floyd and A. Romanow, “TCP Selective Acknowledgment Options,” RFC 2018, Internet Engineering Task Force, October 1996.
[9] H. Wang, C.L. Williamson, “A new scheme for TCP congestion control: smooth-start and dynamic recovery,” Proceedings of IEEE MASCOTS, pp. 69–76, 1998.
[10] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, “An extension to the selective acknowledgement (SACK) option for TCP,” IETF RFC 2883, July 2000.
[11] C. Casetti, M. Geria, S.S. Lee, S. Mascolo, M. Sanadidi, “TCP with faster recovery,” Proceedings of IEEE MILCOM, vol. 1, pp. 320–324, 2000.
[12] M. Zhang, B. Karp, S. Floyd, L.L. Peterson, “RR–TCP: a reordering–robust TCP with DSACK,” Proceedings of IEEE ICNP, pp. 95–106, 2003.
[13] B. Kim, J. Lee, “Retransmission loss recovery by duplicate acknowledgment counting,” IEEE Communications Letters, vol. 8, No. 1, pp. 69–71, 2004.
[14] B. Kim, D. Kim, J. Lee, “Lost retransmission detection for TCP SACK,” IEEE Communications Letters, vol. 8, No. 9, pp. 600–602, 2004.
[15] L. S. Brakmo, S. W. O’Malley and L. L. Peterson, “TCP Vegas: New Techniques for Congestion Detection and Avoidance,” Proceedings of ACM SIGCOMM, London, August 31-September 2, pp. 24–35, 1994.
[16] L. Brakmo and L. Peterson, “TCP Vegas: End-to-End Congestion Avoidance on Global Internet,” IEEE Journal on Selected Areas in Communications, Vol. 13, No. 8, pp. 1465-1480, 1995.
[17] K. Fall and S. Floyd, “Simulation Based Comparisons of Tahoe, Reno and SACK TCP,” ACM SIGCOMM Computer Communication Review, Vol. 26, No. 3, July 1996.
[18] J. Mo, R.J. La, V. Anantharam, J. Walrand, "Analysis and comparison of TCP Reno and Vegas," Proceedings of IEEE INFOCOM, pp. 1556–1563, 1999.
[19] Hanaa A. Torkey, Gamal M. Attiya and I. Z. Morsi, "Performance Evaluation of End-to-End Congestion Control Protocols", Minufiya Journal of Electronic Engineering Research (MJEER), Vol. 18, No. 2, pp. 99-118, July 2008.
[20] 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.
[21] M. Niels, B. Chadi, A. Konstantin, and A. Eitan, ”Inter-protocol fairness between TCP NewReno and TCP Westwood”, 3rd EuroNGI Conference on Next Generation Internet Networks, Vol.1, pp. 21-23, May 2007.
[22] 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.
[23] 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.
[24] K. Fall, and K. Varadhan, “The ns Manual (formerly ns Notes and Documentation)”, UC Berkeley, LBL, USC/ISI, and Xerox PARC, December 2006.
[25] GT-ITM “Georgia Tech Internetwork Topology”, http://www.cc.gatech.edu/project/gtitm.
[26] O. Heckmann, M. Piringer, J. Schmitt, and R. Steinmetz, “How to use Topology Generators to create realistic Topologies”, Technical Report, KOM Darmstadt University Germany, December 2002.

Gamal Attiya graduated in 1993 and obtained his MSc degree in computer science and engineering from the Menufiya University, Egypt, in 1999. He received PhD degree in computer engineering from the University of Marne-La-Vallée, Paris-France, in 2004. He is currently Lecturer at the department of Computer Science and Engineering, Faculty of Electronic Engineering, Minoufiya University, Egypt. His main research interests include distributed computing, task allocation and scheduling, computer networks and protocols, congestion control, QoS, and multimedia networking.