Performance Evaluation of End-to-End Congestion Control Protocols

***Hanaa A. Torkey*[*], *Gamal M. Attiya*[†], *and I. Z. Morsi*[‡]**

1 Abstract

Congestion control remains an important topic for today's Internet protocols. Congestion is generally bad for net users, applications and network. Several mechanisms were proposed by researchers to improve congestion control. These mechanisms include TCP Tahoe, Reno, Vegas, SACK, and NewReno. In this paper, the current congestion control protocols are evaluated considering throughput, losses, delay, and fairness issues that provided by each mechanism. This study is done using the well known network simulator NS-2 and a realistic topology generator called GT-ITM

2 Introduction

Over the past decade, the use of Internet services has experienced dramatic growth and recently Internet applications have evolved from standard document-retrieval functionality to advanced multimedia services. One of the main barriers to the continuing success of the Internet is Internet congestion, which in many cases leads to unacceptable long response times particularly for real-time applications.
Generally, the network may get congested for one of two reasons. First, a high speed

[*]Dept. of Computer Science and Engineering, Faculty of Electronic Engineering, Minufiya University
[†]Dept. of Computer Science and Engineering, Faculty of Electronic Engineering, Minufiya University
[‡]Dept. of Electrical Engineering Power, Faculty of Engineering, Minufiya University

computer source might generate traffic faster than the network can transfer. Second, when many sources with high sending rates put there data simultaneously on a limited bandwidth link. When more packets arrive too quickly than a receiving-host or router can process, they are temporarily stored in memory. If the traffic continues, the receiving-host or router eventually exhausts its memory, and then it discards additional packets that arrive afterward. When a packet encounters congestion, there is a high chance that the packet is dropped out, and this dropped packet actually wasting precious network bandwidth along the path from its sender to its untimely death. Therefore, the stability of the Internet depends on the congestion control at the network. This motivates the design of congestion control schemes, which allow the user to gain utility from the network either by reserving resources to prevent overload, or by reacting to the increased networks loads.
Congestion control is a form of distributed mechanisms that either prevent congestion before it happen or remove it if happened. There are two major objectives in congestion control mechanisms. The first is to avoid the occurrence of network congestion and dissolve the congestion if its occurrence cannot be avoided. The second is to provide a fair service to the different connections. Generally, there are two ways to implement congestion control:

- Network assisted congestion control mechanisms (approaches taken by routers).

- End-to-End congestion control mechanisms (approaches taken by Transmission Control Protocol).

The congestion control protocols considered and evaluated in this paper use the second way and are mostly achieved in transport layer. The basic idea of congestion control by TCP is that the TCP sender always attempts to fill the pipe between the sending and receiving hosts and also to adapt its transmission rate in the network. This adaptation prevents a source from exceeding the network capacity so as to avoid congestion in routers, links or at the destination host. Over time, much research has been conducted to achieve stable, efficient, and fair operation of TCP congestion control. A set of End-to-End mechanisms has been widely acknowledged for maintaining stability of the Internet. Among these mechanisms are TCP Tahoe [1], Reno [2], NewReno [3], Sack [4], and Vegas [5]. In this paper, TCP congestion control mechanisms have been evaluated considering throughput, losses, and fairness for each mechanism. This investigation is done using the well known network simulator NS-2 and a realistic topology generator called GT-ITM.
This paper is organized as follows. Section 2 provides a brief description of the Transmission Control Protocol (TCP) while Section 3 presents the basics of TCP congestion control algorithms. Section 4 briefly describes the current TCP congestion control mechanisms. In section 5, the simulation results are given; simulation environments and assumptions are first defined and then the simulation results for each TCP mechanism are discussed. Finally, conclusions and suggestions for farther research aspects are given in Section 6.

3 Transmission Control Protocol

The Transmission Control Protocol (TCP) is the most dominant protocol used to deliver data reliably over the Internet [6]. The TCP is often described as a byte stream, connection-oriented and reliable delivery transport layer protocol. It guarantees the delivery of data supplied from an application to the other end reliably, although the underlying layers (IP) offer only unreliable data delivery [7] [8]. This reliability is achieved by assigning a sequence number for each transmitted packet and expecting a positive acknowledge (ACK) to come back from the receiver [9] [10]. This acknowledges maintains the next data expected to be received by the receiver. The sequence number is used to reorder the packets which may arrive at the receiver either out of order, duplicated or corrupted. Indeed, the receiver handles the corrupted packets by discarding them, so the sender eventually times out and retransmits corrupted or lost packets. The role of TCP is then to provide End-to-End mechanisms to ensure that data is reliably delivered.
Originally, TCP implements a window based flow control mechanism [11]. A window based protocol means that a current window size defines a strict upper bound on the amount of unacknowledged packets that can be in transit between a given sender receiver pair. The early implementation of TCP follows a simple Go-Back-N model using cumulative positive acknowledgment and requiring a retransmit timer expiration to resend data lost during transportation without any powerful congestion control techniques. In Go-Back-N model, sender transmits up to a window size of data continuously without waiting ACK. If the packets are received in order, and free of error, the receiver acknowledges them by sending an ACK. On the other hand, if the transmitter times out waiting for an ACK, it resends all packets starting from the oldest packet which has not been acknowledged.

4 TCP's Congestion Control

The earlier implementations of TCP would start a connection with the sender injecting multiple segments into the network, up to the window size advertised by the receiver. While this is possible when the two hosts are on the same LAN, problems can arise with the presence of intermediate routers and slower links between the sender and the receiver. Some intermediate routers cannot handle such situation, so queuing of packet results, packets get dropped, retransmission should take place and consequently performance is degraded.
Modern TCP implementations contain a number of algorithms to control network congestion while maintaining good throughput. The TCP congestion control algorithms prevent a sender from overrunning the capacity of the network. Besides the receiver's advertised window, awnd, TCP's congestion control introduced two new variables for the connection: the congestion window (cwnd), and the slow start threshold (ssthresh). The window size of the sender, w, is defined to be w = min (cwnd, awnd), instead of being equal to awnd. The congestion window is a flow control imposed by the sender, while the receiver's advertised window is a flow control imposed by the receiver. The former is based on the sender's assessment of perceived network congestion; the latter is related to the amount of available buffer

space at the receiver for this connection [12]. The congestion window can be thought of as being a counterpart to advertised window. Whereas awnd is used to prevent the sender from overrunning the resources of the receiver, the purpose of cwnd is to prevent the sender from sending more data than the network can accommodate in the current load conditions.
As a congestion control mechanism, TCP sender dynamically increases/decreases its window size according to the degree of network congestion. The idea of congestion control is thus to adaptively modify cwnd to reflect the current load of the network. In practice, this is done through detection of lost packets which can basically be detected either via a time-out mechanism or via duplicate acknowledges (DACKs). Modern implementations of TCP contain four algorithms as basic Internet standards: Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery.

4.1 Slow Start

The slow start algorithm operates by observing that the rate at which new packets should be injected into the network is the same as (or less than) the rate at which the acknowledgments are returned by the other end. When a new connection is established, the congestion window (cwnd) is initialized to one segment (cwnd = 1). The sender then starts transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one segment to two, and hence two segments can be sent. When each of those two segments is acknowledged, the congestion window is then increased to four. That is, after each received ACK, the value of cwnd is updated to cwnd = cwnd + 1 implying doubling of cwnd for each RTT. This provides an exponential growth of the cwnd per Round Trip Time (RTT), although it is not exactly exponential because the receiver may delay its ACKs. This continues until the occurrence of congestion or reaching the value of ssthresh (slow start threshold). At this point, the connection goes into congestion avoidance phase. Conceptually, ssthresh indicates the right window size depending on current network load calculated based on first few packets transmitted. The earlier implementation of slow start was performed only if the other end in different network current always performs slow start [13]. The start slow behavior can be described as follows:
$Initial : cwnd = 1;$
$For(eachpacketAcked)$
$cwnd++;$
$Until(congestionevent, or, cwnd > ssthresh)$

4.2 Congestion Avoidance

As soon as congestion window size (cwnd) crosses ssthresh, TCP goes into congestion avoidance phase to ensure that cwnd does not exponentially increase without using Additive Increase Multiplicative Decrease (AIMD) system. For each received acknowledge, cwnd is increased by $1/cwnd$(Additive Increase) implying linear instead of exponential growth. This linear increase will continue until a packet loss is

detected.
$/* slowstartisover */$
$/* cwnd > ssthresh */$
$EveryAck:$
$cwnd = cwnd + (1/cwnd)$
$Until(timeoutor3Dacks)$
When congestion occurs, TCP sender must slow down its transmission rate of packets into the network, If the congestion is indicated by timeout expiration, one half of the current cwnd or at least two segments are saved into ssthresh ($ssthresh = max(cwnd/2, 2segment)$)and the cwnd is set to one segment and invoke slow start phase. Additionally, if the congestion is indicated by receiving of a threshold of duplicate acknowledges (DACKs), generally set to 3 DACKs, both ssthresh and cwnd are set to one half of the current one (Multiplicative Decrease) and invoke fast retransmit and fast recovery phase.

4.3 Fast Retransmit

The purpose of fast retransmit mechanism is to speed up the retransmission process by allowing the sender to retransmit a packet as soon as it has enough evidence that a packet has been lost. Fast retransmit avoids having TCP to wait for a timeout to resend lost segments. This means that instead of waiting for the retransmit timer to expire, the sender can retransmit a packet immediately after receiving three duplicate ACKs.
$Fastretransmit:$
$Afterreceiving3DACKs$
$Sendthatpacket;$
$/* avoidwaitingtimeout */$
Since TCP sender does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is enough indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire.

4.4 Fast Recovery

The fast recovery algorithm is an improvement that allows high throughput under moderate congestion, especially for large windows. After fast retransmit sends what appears to be the missing segment, congestion avoidance, rather than a slow start, is performed, the reason for not performing slow start in this case is that the receipt of the duplicate ACKs tells TCP that more than just a packet has been lost. Since the receiver can only generate the duplicate ACK when another segment is received, that segment has left the network and it should be in the receiver's buffer. That is, there is still data flowing between the two ends [14], and TCP does not want to

reduce the flow abruptly by going into slow start.
$Afterfastretransmit$;
$/*donotenterslowstart*/$
$ssthresh = Cwnd/2$;
$Cwnd = ssthresh + 3$;
$EachDACKreceived$;
$Cwnd++$;
$Sendnewpacketifallowed$;
$AfternewAck$:
$Cwnd = ssthresh$;
$Returntocongestionavoidance$;

5 TCP congestion control Mechanisms

Over the years, many enhancements are made to TCP congestion control by many researches leading to addition of new algorithms [15]. The rest of this section presents brief description to the different TCP mechanisms, which are Tahoe, Reno, Vegas, Sack and NewReno.

5.1 TCP Tahoe

The TCP Tahoe (4.3 BSD Tahoe) is the first implementation that handles congestion control. The Tahoe TCP adds the congestion control algorithms to refine the earlier implementations of TCP. These algorithms are Slow-Start, Congestion Avoidance and Fast Retransmit. The refinements include a modification to the round-trip time estimator used to set retransmission timeout values. With Fast Retransmit, after receiving a small number of duplicate acknowledgments (DACKs) to the same TCP segment, the data sender infers that a packet has been lost and retransmits the packet without waiting for the retransmission time to expire. Then the sender sets the ssthresh to half of the current cwnd and sets the cwnd to one, and starts the slow start algorithm [16]. So timeout is used as the last resort to recover lost packets [17].

5.2 TCP Reno

The Reno TCP retains the enhancements incorporated into Tahoe, but modifies the Fast Retransmit operation and includes Fast Recovery [18]. Fast Recovery operates by assuming that each DACK received represents a single packet having left the pipe (communication path). Thus during Fast Recovery the TCP sender is able to make intelligent estimation of the amount of outstanding data. TCP enters the Fast Recovery algorithm after receiving 3 DACKs. Once the threshold of 3 DACK is received, the sender transmits one packet and reduces its cwnd and ssthresh by one half of the current cwnd, continues with Fast Recovery until the receiving of new ACK Then TCP invokes Congestion Avoidance phase Instead of slow-starting as performed by a Tahoe TCP sender. The Reno sender uses additional incoming DACK to clock subsequent outgoing packets.

5.3 TCP NewReno

The TCP NewReno enhances the TCP Reno to include multiple packets dropped from a single window [19]. The change concerns the sender's behavior during Fast Recovery when a partial ACK, that acknowledges some but not all of the packets, is received. In Reno, partial ACKs take TCP out of Fast Recovery by deflating the usable window back to the size of the congestion window. In NewReno, partial ACKs do not take TCP out of Fast Recovery. Instead, partial ACKs received during Fast Recovery are treated as an indication that the packet immediately following the acknowledged packet in the sequence space has been lost and should be retransmitted. Thus, TCP NewReno can recover multiple packets losses from a single window of data, without retransmission timeout. NewReno remains in Fast Recovery until all of the data outstanding at the point when Fast Recovery was initiated to be acknowledged. In other words, the TCP NewReno distinguishes between a partial Ack and a full Ack, where the later acknowledge all the packets that were outstanding at the start of fast recovery period and the former acknowledge only some but not all of the outstanding data.
$If(Ack = partialAck)$
$Stayinfastrecovery$;
$RetransmitonepacketperRTT$;
$If(Ack = fullAck)$
$Exitfastrecovery$;

5.4 TCP SACK

The TCP SACK (TCP with Selective Acknowledgement (SACK)) is a conservative extension for TCP Reno's congestion control mechanism by supporting SACK option by both ends [20]. In TCP SACK, if a packet is dropped, it creates a hole in the receiving window when the receiver receives a packet right after the hole and then the receiver sends a DACK for the packet before the hole as TCP Reno does. The SACK option field adds to the ACK containing a pair of sequence number describing the blocks of data that were received out of order with a maximum size 40 bytes [21]. So one ACK can only carry at most 4 blocks and because of the increasing use of timestamp option the number of SACK block arrives 3. The first block reports the most recently received ordered sequence of packets. Adding SACK does not change the underlying congestion control algorithms implementation. The main difference when multiple packets are dropped from one window, is that the sender can selectively resend lost packets depending on SACK option field in ACK. The SACK adds a variable pipe to fast recovery algorithm representing the estimated number of the packets that stand in the path, the sender only retransmits or sends new packet when the variable pipe value is less than cwnd.The pipe changes as follows:
$pipe++$
When packet is transmitted or;
$pipe--$
when DACK is received. Succeeding development of SACK gives it special han-

dling for partial ACK that it decreases the pipe by two rather than one. A solution to prevent the unwanted retransmission is proposed in [22]. It suggested reducing the amount of data used to represent SACK block to 32 bit sequence number for the first block right edge, but the rest of the edges could be represented in either way, one way using the offset from the first block right edge, or as an offset from the previous edge value, which will lead to increase the number of block in SACK option.

5.5 TCP Vegas

TCP Vegas extends the Reno's retransmission mechanisms. It dynamically increases/decreases the congestion window size, based on the available bandwidth in the network. TCP Vegas adopt more sophisticated bandwidth estimation scheme using the difference between the expected and actual flow rates [23] [24]. When the network is congested, the difference between the Expected and the Actual flow rates is used to adjust the window size [25] [26]. The Expected and the Actual values can be formulated as:

$$Expected = W/RTTb;$$

$$Actual = windowsize(W)/RTT;$$

$$Diff = [Expected - Actual] * RTTb;$$

Where, the RTTb (base RTT) is the minimum round trip time. The cwnd continuously updates to:

$$Cwnd = \begin{cases} Cwnd + 1; & \text{if } Diff < \alpha \\ Cwnd - 1; & \text{if } Diff > \beta \\ Cwnd; & \text{otherwise} \end{cases}$$

where α and β are two threshold higher than zero. As network congestion in the backward path should not affect the flow in forward path. An enhancement to TCP Vegas is done in [27] to overcome the reduction in Cwnd size. The reduction is caused by the increasing in backward queuing time that affect in turn the actual throughput and enlarging the Diff. The authors suggest to divide the measured RTT into forward delay time, forward queuing time, backward delay time, and backward queuing time using of tcp timestamp option to obtain backward queuing time and estimating the Actual as follow:

$$Actual = windowsize/(RTT - QDbackward),$$

where QDbackward is the backward queuing delay.

6 Simulation Results

Three performance metrics are used in evaluating TCP congestion control mechanisms; these are throughput, losses and fairness. The throughput refers to the

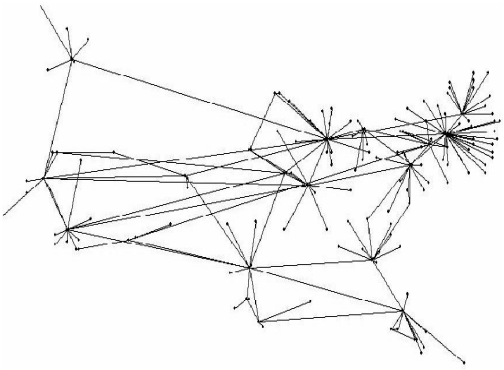

Figure 1. *AT&T network topology [34]*

number of packets sent from the source to the destination which are correctly received. The losses are the packets that are retransmitted from the sender whatever is corrupted or lost. The fairness issue of each mechanism is evaluated based on the presence of non congestion control flow protocol, like UDP.

6.1 Simulation Environments and Assumptions

In investigating the TCP mechanisms by simulation, the selected topology often influences simulations outcomes; hence, realistic topologies are needed to produce realistic results. In this study, the well known network simulator (NS-2) [28] was used to evaluate the different mechanisms of TCP congestion control. The NS-2 is an event driven, object oriented network simulator enabling the simulation of different network (LAN, WAN) considering different protocols, traffic sources, and router queue managements. Also from the wide topology generators [29] [30]; GT-ITM [31], BRITE [32], and TIERS [33], GT-ITM topology generator was used to produce realistic topology. In [34], it have been proved that the level of similarity for the created topology using GT-ITM generator is high and indicate that the hierarchical topology generators are able to produce realistic router level topology, with a similarity to the real network reaching the order of 96.6%.
Figure 1 shows the network topology that is generated by GT-ITM generator and used for evaluating the TCP mechanisms in this study.

6.2 Throughput

Figure 2 shows throughput for each TCP mechanism. The aggregate or cumulative throughput is used, it is calculated as the collection of the received data over the time. The throughput is determined in kilo-bit per second (Kb/s), where the amount of the receiving is calculated every Time Interval length (TIL) and divided by the time. The results are given using TIL=1 second. As shown from the Figure

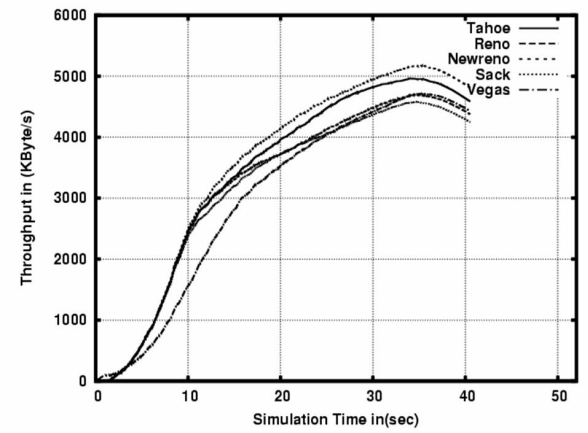

Figure 2. *Throughput VS. simulation Time*

for all TCP mechanisms in slow start phase the throughput increased exponentially, where in slow start algorithm, it duplicates the congestion window every RTT. As entering in congestion avoidance phase depends on ssthresh value which is determined from the first flow packets in the transmission. It is found that, with TCP Tahoe, the sender may retransmit packet which have been received correctly and this decreases its throughput. By entering the congestion avoidance phase, the congestion window size increases linearly until packet lost that is translated as congestion indication. Adding of fast retransmit algorithm improve the performance of TCP over its earlier implementation.

In Tahoe TCP the connection always goes to slow start after a packet loss. However, if the window size is large and packet losses are rare, it would be better for the connection to continue from the congestion avoidance phase, since it will take a while to increase the window size from 1 to ssthresh value. The purpose of the fast recovery algorithm in Reno TCP is to achieve this behavior. So as one can see, Reno's Fast Recovery algorithm is optimized for the case when a single packet is dropped from a window of data. Reno can suffer from performance problems when multiple packets are dropped from a data window. With multiple packet losses TCP Reno is times out and triggers slow start phase. As shown from the figure Reno throughput gets less than Tahoe and that is because with multiple packet losses from single window Reno will decrease its cwnd more than one time.

TCP NewReno is improved over Reno when multiple packet losses from single window. The NewReno increases the overall throughput than Reno. As shown from the figure, TCP NewReno has the highest throughput. It should be noted that TCP NewReno retransmitting one lost packet per round-trip time until all of the lost packets from that window have been retransmitted. TCP SACK tries to eliminate the problem of multiple packets lost from one window, where TCP SACK add selective acknowledgement and selective retransmit that the TCP sender can detect single window multiple packet losses using the SACK option within the acknowledg-

ment, and selectively retransmit the loss packets. Even that adding SACK option to the acknowledgement increases the load in the network and needs improvement in TCP sender and receiver to enable SACK option. From the figure, it could be noted that TCP Sack throughput higher than this of Reno. TCP Vegas attempts to increase its cwnd according to the bandwidth available. It is important to note that the TCP Vegas tries to avoid congestion more than just treating with it. From the figure, Vegas TCP get the lowest throughput value. The reasoning for this is that when the actual throughput of a connection approaches the value of the expected maximum throughput, it may not be utilizing the intermediate routers buffer space efficiently and hence, it should increase the flow rate. On the other hand, when the actual throughput is much less than the expected throughput, the network is likely to be congested and hence the sender should reduce the flow rate (cwnd).

6.3 Losses

In a network, most of the packet losses are due to the transmission delay that is undergone in the network path and the waiting time in routers queues between the two ends. The delay in router queue however is the largest factor that affects the total delay. With increasing the delay, the probability of packet dropping increases. Figure 3 shows the cumulative sum of all dropped packets plotted against time, where the number of dropped packets is summed cumulatively and the summation is determined for each time interval (one second in the graph).
The figure shows the dropped packets by TCP Tahoe. It has a high dropping rate. There is a similar loss value for all the mechanism, except Vegas, at the start of the simulation until simulation time of 10 sec, the increasing in the losses is because of the behavior of slow start and congestion avoidance phases, where the cwnd increasing cause delay on the router queue. With TCP Reno, the overall cumulative sum of packet losses decreased to approximately 35.7% of that of Tahoe, as fast recovery decrease the unnecessarily packet retransmission. In Vegas TCP, the overall sum of dropped packet is approximately 69.0% of that of Tahoe due to Vegas cwnd increasing/decreasing mechanism.

In TCP SACK, the overall cumulative sum of dropped packets equals that of NewReno but the change in the losses differs from that of Reno where it can recover single window multiple packet losses faster than the other mechanism. With TCP NewReno, the dropped packet is slightly increased over that of SACK where in fast recovery NewReno send new packet with received of duplicate ACK in order to fill the pipe which lead to increase the probability of packet dropping. Figure 4 show the delay vs. the time for the all mechanism, it's noted that Vegas has lower delay than all mechanisms in contrary to Tahoe and Reno which have high delay. The congestion window behavior is shown in figure 5, the change of the congestion window illustrates the behavior of the algorithms in each mechanism.

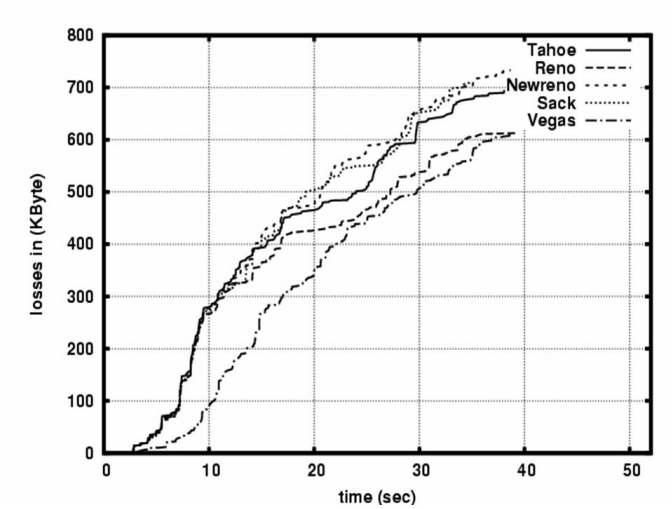

Figure 3. *losses VS. time*

6.4 Fairness

Keeping fairness between multiple connections is an essential factor for the network. Fairness is achieved when having link bandwidth equally distributed among the shared connections. In this section, the behavior of each TCP congestion control mechanism will be studied when sharing the same link with UDP (User Datagram Protocol) protocol, which does not offer congestion control mechanism. The throughput is used to show the bandwidth sharing, and how each TCP mechanism behavior in presence of UDP. Figure 6 shows the network topology that used in fairness study. The simulation results for each TCP mechanism are plotted in figure 7, but the throughput of UDP does not plotted at the same graph to give a chance for the graph showing the small differences.

The Figure shows that TCP suffers unfair bandwidth sharing when transmitting data in the same link with non congestion control offered protocol. As shown from the Figure each TCP congestion control mechanism has a different level of bandwidth sharing with UDP. TCP Tahoe and NewReno (over SACK) have the highest sharing level unlike Vegas where the presence of UDP gets a bad effect on its performance. Vegas detects the network status and change its window size by decreasing or increasing and that causes a problem for it. Where if Vegas share with other protocol when network load increased Vegas start to decreased its transmission rate which would not happen for the other protocol that is leading to unfair bandwidth

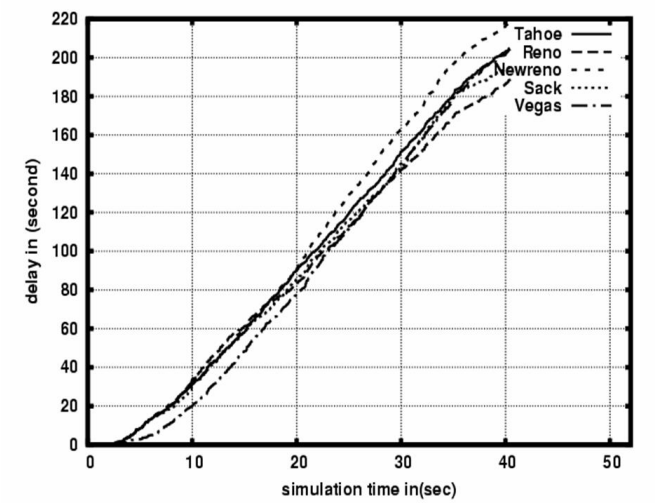

Figure 4. *delay VS. simulation Time*

share.

7 Conclusions

In this paper, the performance of TCP congestion control mechanisms including Tahoe, Reno, NewReno, SACK and Vegas are evaluated using the network simulator NS-2 and a realistic topology generator called GT-ITM. From the simulation results it have been noted that, TCP NewReno has a slightly better performance over other mechanisms, even TCP Vegas have the better delay and losses performance, but it gets unfair bandwidth share when shared with other protocols like UDP. TCP NewReno with some modifications to fast recovery algorithm could improve its performance without adding SACK option and lead to an easy deployment for TCP NewReno over TCP Sack.
Transmission control protocol has several implementation versions which intend to improve congestion control mechanisms. All of these mechanisms have a problem with the fast recovery algorithm when there are multiple packet losses within a single window, and have anther problem with small window size. The comming work in this research will be:

- Addressing the improved of the performance of TCP congestion control and

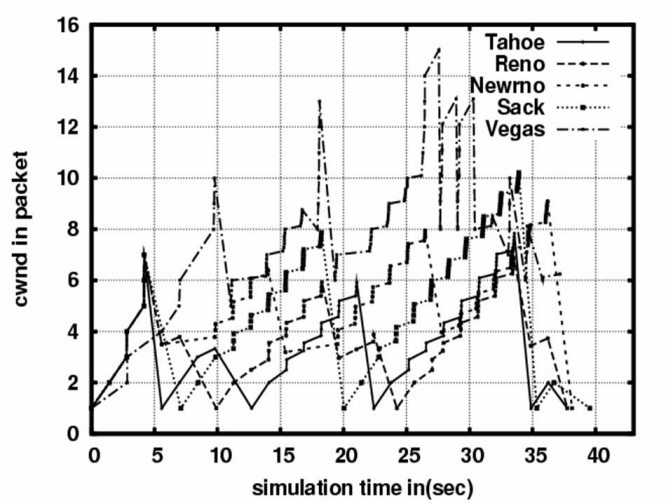

Figure 5. *congestion window Behavire*

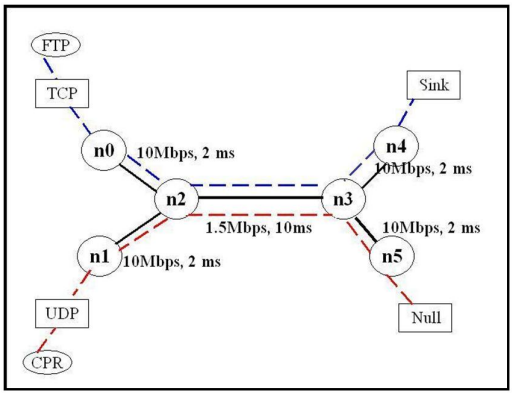

Figure 6. *fairness study topology*

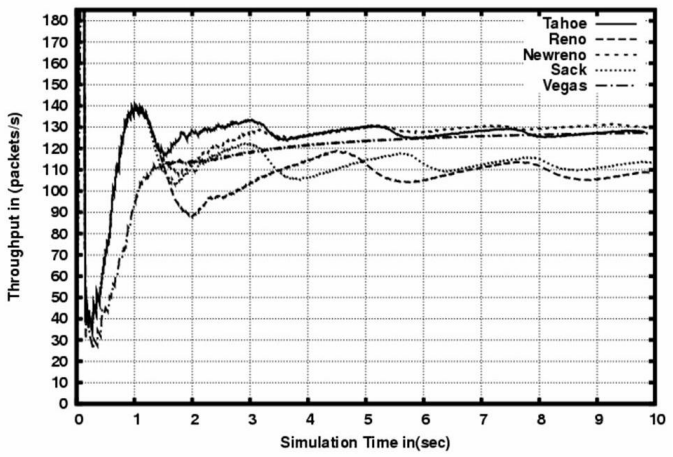

Figure 7. *TCP Fairness*

avoiding congestion into network by modifying an algorithm to implement congestion related values for window size after fast retransmit

- Measuring the performance of that new mechanism compared with existing mechanisms.

Bibliography

[1] V. Jacobson, and M. J. Karels, "Congestion Avoidance and Control", Sigcomm '88 Symposium Stanford, CA, August 1988, vol.18 (.4), pp.314-329.

[2] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose, "Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation", IEEE/ACM Transactions on Networking, vol. 8(2), April 2000, pp: 133-145.

[3] S. Floyd, and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 2582, April 1999.

[4] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.

[5] L. S. Brakmo, S. W. O'Malley, and L. Peterson," TCP Vegas: New Techniques for Congestion Detection and Avoidance", ACM SIGCOMM Computer Communication Review, vol. 24(4), October 1994, pp: 24-35.

[6] W. Boulevard, and A. Way," Transmission Control Protocol", RFC 793, September 1981.

[7] W. R. Stevens, "TCP/IP Illustrated, Volume 1 the Protocols", Addison Wesley Professional, Oct. 1993.

[8] S. J. Golestani, and S. Bhattacharyya, "A Class of End-to-End Congestion Control Algorithms for the Internet", the Sixth International Conference on Network Protocols, October 1998, pp:137-151.

[9] S. Fahmy, and T. P. Karwa, "Tcp congestion control: Overview and Survey of Ongoing Research", Technical report, Purdue University, 2000.

[10] C. Barakat, "TCP/IP Modeling and Validation", IEEE Network, vol.15 Issue 3, May 2001, pp: 38-47.

[11] S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet", IEEE/ACM Transactions On Networking, vol. 7(4), August 1999, pp: 458-472.

[12] S. Kunniyur, and R. Srikant, "End-to-End Congestion Control Schemes: Utility Functions, Random Losses and ECN Marks", IEEE/ACM Transactions on Networking, vol. 7(5), October 2003, pp: 689-702.

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

[14] M. Allman, V. Paxson, and W. Stevens,"TCP Congestion Control", RFC 2581, April 1999.

[15] B. Kim, D. Kim, and J. Lee," Lost Retransmission Detection for TCP SACK", IEEE Communications Letters, vol. 8(9), September 2004, pp: 600 - 602.

[16] K. Fall, and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", Computer Communication Review, vol. 26 (3) , July 1996, pp:5-21

[17] B. Sikdar, S. Kalyanaraman, and K. S. Vastola, "Analytic Models for the Latency and Steady-State Throughput of TCP Tahoe, Reno, and SACK", IEEE/ACM Transactions On Networking, vol. 11(6), December 2003, pp: 959 - 971.

[18] M. Jeonghoon, R. J. La, V. Anantharam, and J. Walrand, "Analysis and comparison of TCP Reno and Vegas", IEEE Eighteenth Annual Joint Conference of Computer and Communications Societies INFOCOM apos'99, vol. 3, Mar 1999, pp:1556-1563.

[19] N. Parvez, A. Mahanti, and C. Williamson, "TCP NewReno: Slow-but-Steady or Impatient?", IEEE International Communications Conference, ICC '06, vol. 2, June 2006, pp: 716-722.

[20] R. Bruyeron, B. Hemon, and L. Zhang, "Experimentations with TCP Selective Acknowledgment", ACM SIGCOMM Computer Communication Review, vol.28(2), April 1998, pp: 54-77

[21] K.N. Srijith, L. Jacob, and A.L. Ananda, "Worst-case performance limitation of TCP SACK and a feasible solution", IEEE International Conference on Communication Systems, vol. 2, November 2002, pp: 1152- 1156.

[22] J. Jacobson, R. Braderand, and D. Borman, "TCP Extension for high performance", RFC 1323, 1992.

[23] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: new techniques for congestion detection and avoidance", ACM SIGCOMM Computer Communication Review, vol.24(4), October 1994, pp:24-35.

[24] O. A. Hellal, and E. Altman, "Analysis of TCP Vegas and Reno", INFOCOM Annual Joint Conference of the IEEE Computer and Communications Societies, vol.3, March 1999, pp: 21-25.

[25] S. H. Low, L. L. Peterson, and L. Wang,, "Understanding TCP Vegas: A Duality Model", Journal of the ACM, Vol.49 (2), March 2002, pp:207235.

[26] L. S. Brakmo, and L. L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet", IEEE Journal on Selected Areas in Communications, vol.13(.8) , October 1995, pp: 1465-1480.

[27] Y. Chan, C. T. Chan, and Y. C. Chen, "An Enhanced Congestion Avoidance Mechanism for TCP Vegas", Communications Letters, vol.7(7), July 2003, pp:343-345.

[28] K. Fall, and K. Varadhan, "The ns Manual (formerly ns Notes and Documentation)", UC Berkeley, LBL, USC/ISI, and Xerox PARC, December 2006.

[29] O. Heckmann, M. Piringer, J. Schmitt, and R. Steinmetz, "On Realistic Network Topologies for Simulation", ACM SIGCOMM workshop on Models, methods and tools for reproducible network ACM, August 2003, pp:28-32.

[30] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee, "How to Model an Internetwork", IEEE Annual Joint Conference of the Computer Networking INFOCOM '96, vol. 2, March 1996, pp:594-602.

[31] GT-ITM "Georgia Tech Internetwork Topology", http://www.cc.gatech.edu/project/gtitm.

[32] BRITE "Bosten University Representative Internet Topology Generator", http://www.cs.bu.edu/brite.

[33] TIERS "Tiers Topology Generator", http://www.isi.edu/nsnam/ns/ns-topogen.htm#tiers.

[34] 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.