ENSC 835-3: NETWORK PROTOCOLS AND PERFORMANCE

CMPT 885-3: SPECIAL TOPICS: HIGH-PERFORMANCE NETWORKS

Final Project

EVALUATION OF DIFFERENT TCP CONGESTION CONTROL ALGORITHMS USING NS-2

Spring 2002

Hui (Hilary) Zhang (hzhang@sfu.ca)

Zhengbing Bian (zbian@cs.sfu.ca)

http://www.sfu.ca/~zbian/courses/cmpt885/

TABLE OF CONTENT

Abstract 1
1. Introduction 1
2. TCP Congestion Control Algorithm 2
  2.1 TCP Tahoe 2
  2.2 TCP Reno 2
  2.3 TCP SACK 3
  2.4 TCP Vegas 3
3. Evaluation of TCP Congestion Control Algorithms 5
  3.1 Effective Resource Utilization 5
    3.1.1Bandwidth Utilization 5
    3.1.2 Congestion Window Size Variation 6
    3.1.3 Retransmit Rate 6
  3.2 Fairness between Connections with Different Delays 7
    3.2.1 Testify the Behavior on Long Delay connections 7
    3.2.1.1 Two Reno TCP connections 8
    3.2.1.2 Two Vegas TCP connections 9
    3.2.1.3 Two SACK TCP Connections 9
    3.2.2 Fairness Variation with Delay Changes 10
    3.2.3 Queue Algorithms Effects 10
    3.2.3.1 DropTail Queue 11
    3.2.3.2 RED Queue 11
  3.3 Fairness between Different Version TCPs when Competing on Same Link 12
    3.3.1 Competition when two connections running together 12
    3.3.2 Queue Algorithms Effects 13
    3.3.2.1 DropTail Queue 13
    3.3.2.2 RED Queue 14
    3.3.3 Other Improvement 15
  3.4 Reno-Sack1-Vegas 16
4. Conclusion 18
Reference 19
Appendix 19
  Source code list 19

Abstract

The algorithm for TCP congestion control is the main reason we can use the Internet successfully today despite resource bottlenecks and largely unpredictable user access patterns. There are different implementations among which are TCP Reno, SACK and Vegas. We use simulations to evaluate these TCP congestion control algorithms from many aspects. NS-2 is used for the simulation. Effective resource utilization, such as bandwidth utilization, retransmission rate and window size, is compared. We also concern fair resource allocation from two main categories, one is fairness between different delay links, and the other is competition between different TCP congestion control algorithms. Our simulation results show that bias exists in both categories, so factors may affect the fairness are also simulated. We focus on the effect of different queue algorithms, such as Drop Tail and RED.

1. Introduction

Early TCP (Transmission Control Protocol) implementation uses go-back-n model with cumulative positive acknowledgement and requires a retransmit time-out to retransmit the lost packet. These TCP did little to minimize network congestion. TCP congestion control was introduced into the Internet in the 1988 by Van Jacobson [5]. At that time, the Internet was suffering from congestion collapse. This so-called Tahoe TCP added a number of new algorithms such as *Slow-Start*, *Congestion Avoidance* and *Fast Retransmit*. Since then, many modifications have been made to TCP and several different versions of TCP have been implemented. Two years later than the release of Tahoe, Jacobson revised it as Reno TCP by modifying the *Fast Retransmit* operation to include *Fast Recovery* [7]. Later, Brakmo *et al*. have proposed a new version of TCP, which is named TCP Vegas, with fundamentally different congestion avoidance algorithm from TCP Reno [3]. Another conservative extension of Reno TCP is SACK TCP, which adds Selective Acknowledgement to TCP.

In our project, we will evaluate the congestion control algorithms in Reno, Vegas and SACK TCP from different aspects. First, we will compare the performance of these algorithms: how much of the available network bandwidth does it utilize? How frequently does it retransmit packets? How does it modify window size on congestion? These comparisons are based on each version TCP running separately on a congested network. The second evaluation is the fairness of sharing the network. This comparison is taken in two categories of experiment. One is the fairness between different delay connections running the same version TCP. Some algorithms may bias against long delay connection, such as Reno TCP and SACK. The other experiment is carried out between different versions TCP when they compete each other on the same connection. TCP Vegas does not receive a fair share of bandwidth when competing with other TCP Reno or SACK connections. Since bias exists in both categories, how different queue algorithms may affect the fairness is also studied.

All the evaluation data are gained from NS-2. Reno, Vegas and SACK TCP agents have been implemented in NS-2 by former researchers. We designed our simulation scenario and tuned the parameters for each purpose of comparison. We read some test-suites of NS2 and got a clear understanding of the object TCL programming. In this report, we will introduce the chief difference between each algorithm in section 2. In section 3, detail of our simulation and result discussion will be given. A conclusion will be given based on our simulation result in the section 4.

2. TCP Congestion Control Algorithm

The basis of TCP congestion control lies in Additive Increase Multiplicative Decrease (AIMD), halving the congestion window for every window containing a packet loss, and increasing the congestion window by roughly one segment per RTT otherwise. The second component of TCP congestion control is the Retransmit Timer, including the exponential bakeoffs of the retransmit timer when a retransmitted packet is itself dropped. The third fundamental component is the Slow-Start mechanism for the initial probing for available bandwidth. The fourth TCP congestion control mechanism is ACK-clocking, where the arrival of acknowledgements at the sender is used to clock out the transmission of new data.

The TCP variants discussed in this project, except TCP Vegas, all adhere to this underlying framework of Slow-Start, AIMD, Retransmit Timers, and ACK-clocking. None of these changes alter the fundamental underlying dynamics of TCP congestion control. Instead, these changes help to avoid unnecessary Retransmit Timeouts, correct unnecessary Fast Retransmits and Retransmit Timeouts resulting from disordered or delayed packets, and reduce unnecessary costs (in delay and unnecessary retransmits) associated with the mechanism of congestion notification.

2.1 TCP Tahoe

The Tahoe TCP implementation added a number of new algorithms and refinements to earlier TCP implementations. The new algorithms include *Slow-Start*, *Congestion Avoidance*, and *Fast Retransmit* [5]. The refinements include a modification to the round-trip time estimator used to set retransmission timeout values. The Fast Retransmit algorithm is of special interest because it is modified in subsequent versions of TCP. With Fast Retransmit, after receiving a small number of duplicate acknowledgments for the same TCP segment (*dup ACKs*), the data sender infers that a packet has been lost and retransmits the packet without waiting for a retransmission timer to expire, leading to higher channel utilization and connection throughput [1].

2.2 TCP Reno

The Reno TCP implementation retained the enhancements incorporated into Tahoe TCP but modified the Fast Retransmit operation to include *Fast Recovery* [7]. The new algorithm prevents the communication channel from going empty after Fast Retransmit, thereby avoiding the need to Slow-Start to re-fill it after a single packet loss. Fast Recovery operates by assuming each dup ACK received represents a single packet having left the pipe. Thus, during Fast Recovery the TCP sender is able to make intelligent estimates of the amount of outstanding data. A TCP sender enters fast Recovery after receiving an initial threshold (*tcprexmtthresh*) of dup ACKs. Once the threshold (generally is 3) of dup ACKs is received, the sender retransmits one packet and reduces its congestion window by one half. After entering Fast Recovery and retransmit a single packet, the sender effectively waits until half of a window of dup ACKs have been received, and then sends a new packet for each additional dup ACK that is received. Upon receipt of an ACK for new data, the sender exits Fast Recovery. Reno significantly improves upon the behavior of Tahoe TCP when a single packet is dropped from a window of data, but can suffer from performance problems when multiple packets are dropped from a window of data.

2.3 TCP SACK

The congestion control algorithms implemented in SACK TCP are a conservative extension of Reno's congestion control, in that they use the same algorithms for increasing and decreasing the congestion window, and make minimal changes to the other congestion control algorithms. Adding SACK (Selective Acknowledgement) to TCP does not change the basic underlying congestion control algorithms. The SACK TCP implementation preserves the properties of Tahoe and Reno TCP of being robust in the presence of out-of-order packets, and uses retransmit timeouts as the recovery method of last resort. The main difference between the SACK TCP implementation and the Reno TCP implementation is in the behavior when multiple packets are dropped from one window of data. During Fast Recovery, SACK maintains a variable called *pipe* that represents the estimated number of packets outstanding in the path. (This differs from the mechanisms in the Reno implementation.) The sender only retransmits data when estimated number of packets in the path is less than the congestion window. Use of the *pipe* variable decouples the decision of when to send a packet from the decision of which packet to send. . The sender maintains a data structure (*scoreboard*) that remembers acknowledgments from previous SACK options. When the sender is allowed to send a packet, it retransmits the next packet from the list of packets inferred to be missing at the receiver. The SACK sender has a special handling for partial ACKs (ACKs received during Fast Recovery that advance the Acknowledgment Number field of TCP header, but do not take the sender out of fast Recovery). The sender decrements *pipe* by two rather than one for partial ACKs, the SACK sender never recovers more slowly than a Slow-Start. Detailed description of SACK TCP can be found in [1].

2.4 TCP Vegas

TCP Vegas adopts a more sophisticated bandwidth estimation scheme. It uses the difference between expected and actual flow rates to estimate the available bandwidth in the network. The idea is that when the network is not congested, the actual flow rate will be close to the expected flow rate. Otherwise, the actual flow rate will be smaller than the expected flow rate. TCP Vegas, using this difference in flow rates, estimates the congestion level in the network and updates the window size accordingly. This difference in the flow rates can be easily translated into the difference between the window size and the number of acknowledged packets during the round trip time, using the equation

$$Diff = (Expected - Actual)\ BaseRTT,$$

Where *Expected* is the expected rate, *Actual* is the actual rate, and *BaseRTT* is the minimum round trip time. The details of the algorithm are as follow:

1. First, the sender computes the expected flow rate $Expected = \frac{CWND}{BaseRTT}$,
   where *CWND* is the current window size and *BaseRTT* is the minimum round

trip time.

2. Second, the sender estimates the current flow rate by using the actual round trip time according to $Actual = \frac{CWND}{RTT}$, where *RTT* is the actual round trip time of a packet.
3. The sender, using the expected and actual flow rates, computes the estimated backlog in the queue from *diff*=(*Expected* – *Actual*)*BaseRTT*.
4. Based on *diff*, the sender updates its window size as follows:

$$CWND = \begin{cases} CWND + 1 & \text{if } diff < \alpha \\ CWND - 1 & \text{if } diff > \beta \\ CWND & otherwise \end{cases}$$

TCP Vegas tries to keep at least α packets but no more than β packets in the queues. The reason behind this is that TCP Vegas attempts to detect and utilize the extra bandwidth whenever it becomes available without congesting the network. This mechanism is fundamentally different from that used by TCP Reno. TCP Reno always updates its window size to guarantee full utilization of available bandwidth, leading to constant packet losses, whereas TCP Vegas does not cause any oscillation in window size once it converges to an equilibrium point [6].

Our project is focused on Reno, SACK and Vegas TCP since Tahoe is replaced by Reno in most of today’s applications.

3. Evaluation of TCP Congestion Control Algorithms

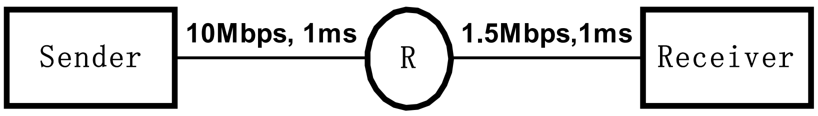
We use simulations to compare TCP Reno, SACK and Vegas from different aspects. All the simulations use FTP traffic, which corresponds to bulk data transfer. The packet size is fixed at 1,000 bytes. The buffer and window sizes are both in the unit of packet. The receiver advised window is large enough so no packet will be dropped at the receiver. We also assume that the ACKs are never lost. All the simulations are run for 120 seconds, unless specified.. We use ACK number ratio to indicate the bandwidth occupancy ratio, since we use the same packet size throughout our simulation (ACK number * packet size / time = bandwidth). We will explain and discuss our simulation results in three main categories as follows.

3.1 Effective Resource Utilization

We compare the effective resource utilization of these three algorithms: how much of the available network bandwidth does it utilize? How frequently does it retransmit packets? How does it modify window size on congestion? These comparisons are based on each version TCP running separately on a congested network.

3.1.1Bandwidth Utilization

Simulation Design



Topology 1

Topology 1 shows the topology of the simple simulation network. The circle indicates a finite-buffer DropTail gateway, and the squares indicate sender and receiver hosts. The links are labeled with their bandwidth capacity and delay.

Simulation Results

It has been reported that Vegas TCP can achieve 37 to 71 percent higher throughput than Reno TCP [3]. In this part of the project, we measure the performance of Vegas, Reno and Sack on link of different loss rates. We add error model on the slow link between the gateway and the receiver. The buffer size is 6. Table 1 shows the results of 1% and 5% uniform loss. We found that the bandwidth utilization ratio of these TCP flavors do not show much difference for link of small loss rate. However, TCP Vegas achieves higher throughput than Reno and Sack for large loss rate.

Table 1: Effective Bandwidth Utilization Ratio

		Reno	Sack	Vegas
Bandwidth utilization	1% loss	89.4%	97.7%	95.5%
	5% loss	45.5%	53.7%	63.7%

3.1.2 Congestion Window Size Variation

One main difference in congestion control algorithms of TCP SACK and TCP Reno is how they deal with more than one packet loss in one congestion window. We simulate the case when four packets are dropped in one congestion window to see the window size variation. Simulation topology is the same as in 3.1.1. We use the source code from [1] and get consistent simulation results with [1]. The result graphs are as follows.

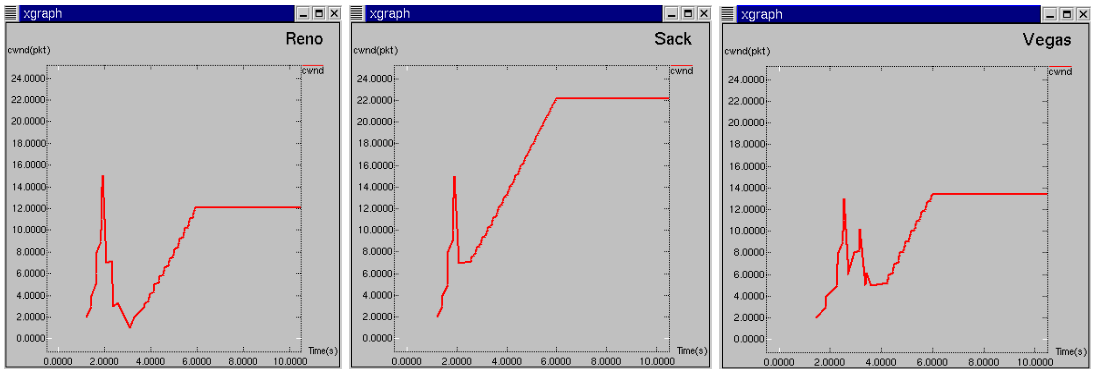


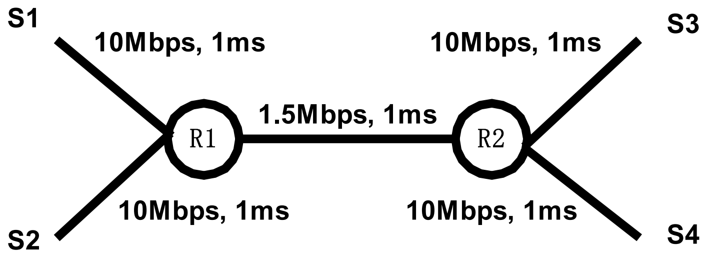
Figure 1 Congestion Window of Four Packets Loss

From *Figure 1*, we see that congestion window of TCP Reno drops to 0 and slow-start when more than one packet are dropped in one window. Congestion window of TCP Vegas oscillates when more packets are dropped, but never goes back from slow-start. TCP SACK maintains the same window size as the value after the first packet drop and returns to a higher window size than both Reno and Vegas. We can say that the algorithm of TCP SACK performs better in the case of more than one packet is dropped in one window. This is consistent with the algorithm explanations in section 2.

3.1.3 Retransmit Rate

Simulation Design

The network configuration is shown in *Topology 2*. In the figure, R1 and R2 are finite-buffer switches and S1~S4 are the end hosts. Connection 1 transmits packets from S1 to S3, and connection 2 from S2 to S4. The links are labeled with their capacities and propagation delays.



Topology 2

Simulation Results

We vary the S1 – S3 and S2 – S4 connections to different TCP flavors and get six possible combinations. These one-on-one experiments are to test the retransmit rate of Reno, SACK and Vegas. The results from simulation are shown in the following table.

Table 2: One-on-One Transfer

	Vegas/Reno	Reno/Vegas	Reno/SACK	SACK/Reno	SACK/Vegas	Vegas/SACK
Transmit packets	6536/13337	13339/6534	9765/10108	10113/9760	13427/6446	6443/13430
Retransmit Packets	0/273	272/0	348/346	348/346	272/0	0/266
Retransmit Ratio (%)	0/2.05	2.04/0	3.56/3.42	3.44/3.55	2.03/0	0/1.98

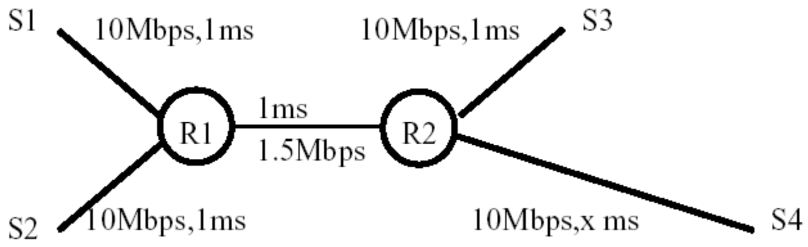
From the above table, we can see that the performance of Vegas is perfect in the sense of loss control. It never loses a packet. When running together with Vegas, the retransmit ratio of Reno and SACK is about 2%, which is smaller than Reno and SACK compete with each other (about 3.5%). The reason for this is as follows. Both Reno and SACK use aggressive algorithm to increase their window size until drop happens. They occupy more bandwidth when compete with Vegas. Therefore, fewer packets are dropped in this case. When Reno and SACK compete, they share the bandwidth almost in half and half, but with the expense of higher loss rate. We will explain this unfair behavior in later subsections.

3.2 Fairness between Connections with Different Delays

In this part, we want to test the fairness between different delay connections. When the same version TCP run together on one bottleneck link, whether they can share the bandwidth fairly if their connection delays are different? We want to testify that TCP Reno is biased against the connections with longer delays and see the behaviors of SACK and Vegas in the same situation. We are also interested in the factors that may affect this fairness, such as different queue algorithms and buffer size.

Simulation Design

Topology 3 shows the topology of the network that was used in the simulation. In the figure, R1 and R2 are finite-buffer switches and S1~S4 are the end hosts. Connection 1 transmits packets from S1 to S3, and connection 2 from S2 to S4. The links are labeled with their capacities and propagation delays, and they will be changed during different simulations. The propagation delay of the link that connects R2 and S4, which is denoted by X in *Topology 3*, will be varied in order to see the effect of different delays on the fairness.



Topology 3: Network Topology

3.2.1 Testify the Behavior on Long Delay connections

In this part, we want to testify the observation in [6] that TCP Reno is biased against the connections with longer delays. The reason for this behavior is as follows. While a source does not detect any congestion, it continues to increase its window size by one during one round trip time (RTT). Obviously, connections with a shorter delay can update their window sizes faster than those with longer delays, and thus capture higher bandwidths. To our understanding, TCP SACK does not change this window increasing mechanism, so we expect the same unfair behavior with TCP SACK.

We designed the simulation scenarios as follows. The network topology is shown in *Topology 3*. S1 and S2 will be set to be the same TCP agents, such as two Reno, two Vegas or two SACK TCP agents, respectively. Results of X=1ms (the same propagation delay as comparison baseline) and X=22ms (the RTT of longer delay connection is 8 times of the shorter one) will be collected to show the fairness between different delay connections.

We use bandwidth occupancy graphs to show the fairness between different delay connections. The vertical axis is bandwidth (in Mbps). The horizontal axis is simulation time (in second). Connections 1 and 2 start at time 0 and 0.5 seconds, respectively. We start to collect simulation data after 15 seconds to eliminate the transient effect. The top red line is the total bandwidth, which is set to 1.5Mbps. The blue line represents the bandwidth of the shorter delay connection. The green one is the bandwidth of longer delay connection. The graph on the left-hand side is bandwidth occupancy of two connections with the same delays, which is used as comparison baseline. The right-hand side graph is bandwidth occupancy of connections with different delays.

3.2.1.1 Two Reno TCP connections

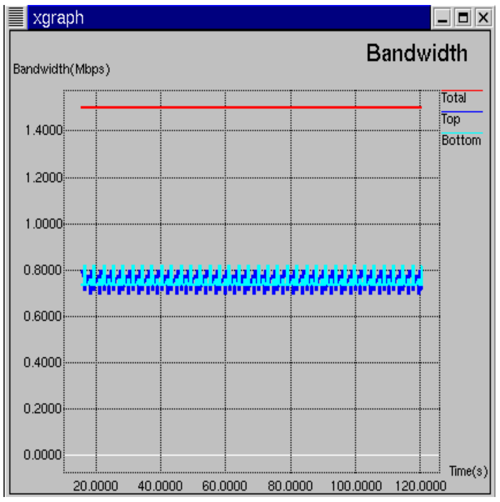


Figure 2a (X=1ms)

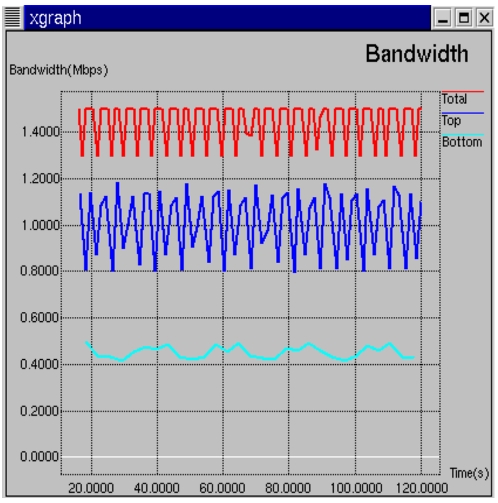


Figure 2b (X=22ms)

Figure 2a and *2b* show the bandwidth occupancy of two Reno TCP connections. From *Figure 2a*, we can see that bandwidth occupancy of two same delay Reno TCP connections is fair. The results of *Figure 2b* show that the delay bias of TCP Reno becomes very noticeable when the delay difference is large. The shorter delay connection (blue line) occupies almost twice bandwidth as the longer one. One can also notices the large oscillation of the total bandwidth Reno TCP. This result is consistent with what we want to testify.

3.2.1.2 Two Vegas TCP connections

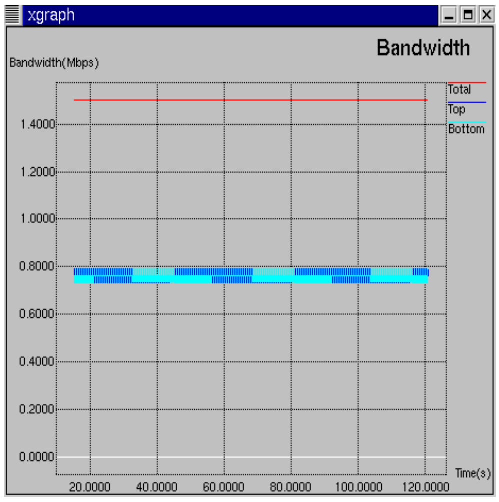


Figure 3a (X=1ms)

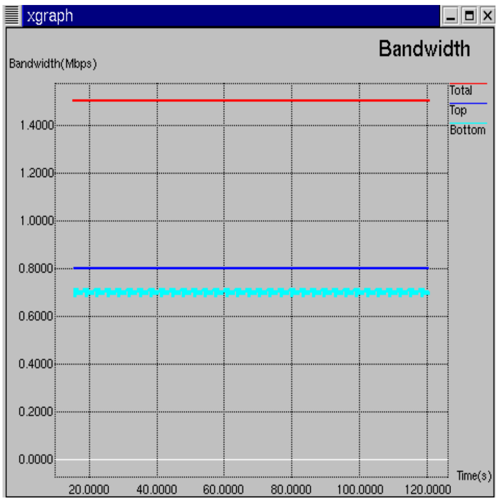


Figure 3b (X=22ms)

Figure 3a and *3b* show the bandwidth of two Vegas TCP connections. We can see that the two connections share the bandwidth in almost half and half (X=1ms, 0.75Mbps; X=22ms, 0.7Mbps and 0.8Mbps). The results show that the bandwidth occupancy does not change significantly with the link delay, which means Vegas TCP is not biased against connection with longer delay. This result is consistent with what we want to testify.

3.2.1.3 Two SACK TCP Connections

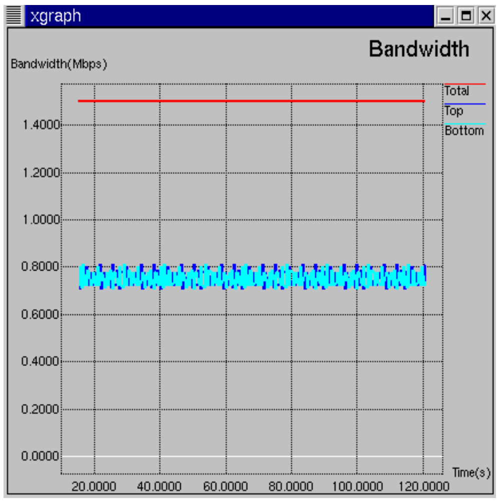


Figure 4a (X=1ms)

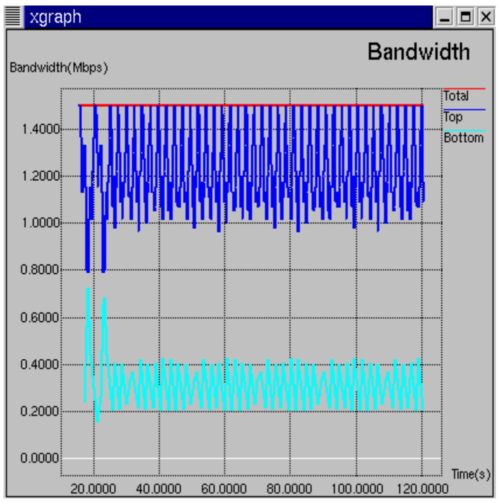


Figure 4b (X=22ms)

Figure 4a and *4b* show the bandwidth of two Sack TCP connections. From the figures, we can see that the delay bias of TCP Sack becomes very noticeable when the delay difference is large. This result is similar to that of Reno TCP, which is quite straightforward, since the window increasing mechanism of Sack is the same as Reno.

3.2.2 Fairness Variation with Delay Changes

As we find out from previous tests, both Reno and SACK TCP are biased against the connection with longer delays. We want to see how this bias changes while the delay differences increase. We vary the number of X in our simulation topology to change the delay of connection 2 (from S3 to S4) as follows.

Table 3: Fairness with RTT Changes

X (ms)	RTT2/RTT1	ACK1/ACK2		
		Reno	SACK	Vegas
1	1	1	1	1
13	5	1.71	1.77	1.17
22	8	2.2	4.31	1.18
58	20	12.42	30.69	1.03
208	70	46.62	70.3	2.15

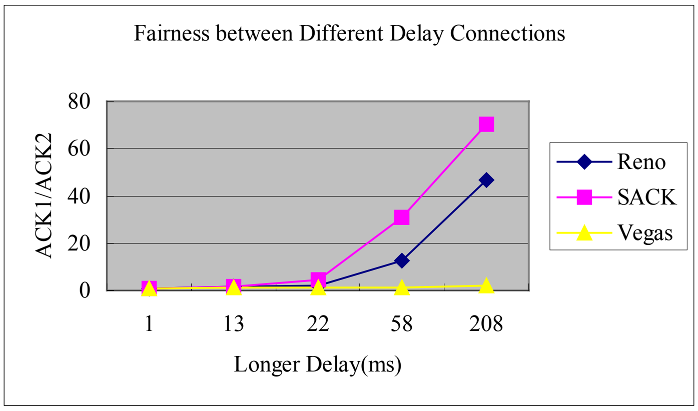


Figure 5

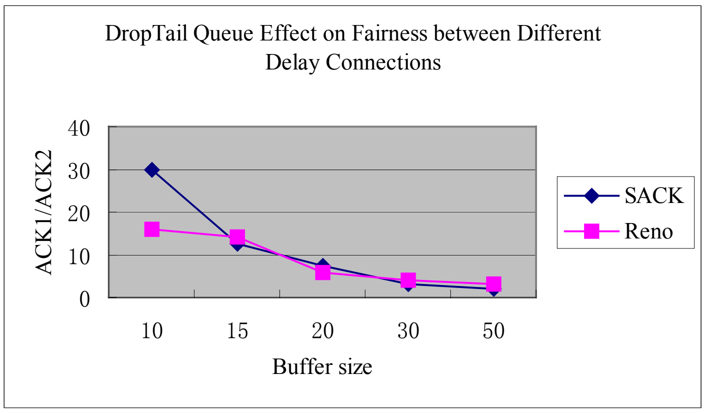
ACK1 and ACK2 are the number of packets that have been acknowledged by the corresponding receivers for connection 1 and 2. We use ACK1/ACK2 to represents the bias (vertical axis), which is the same as bandwidth occupancy ratio. If there's no bias against the longer delay connection, ACK1/ACK2 should be near 1. From the data above, we can see that the delay bias of TCP Reno and SACK becomes very noticeable when the delay difference becomes large, while TCP Vegas does not show such delay bias. The ACK ratio of the two connections under TCP Reno and SACK increases almost linearly with RTT ratio. When the delay difference is quite large (X=58ms and 208ms), the delay bias of SACK TCP is even greater than Reno.

3.2.3 Queue Algorithms Effects

There are many suggestions on how to improve the fairness between different delay connections. Here we focus on the effect of different queuing algorithms, such as DropTail (First come, first service), RED (Random Early Detection). We use simulations to see how the buffer size parameter will affect the fairness in this case.

The simulation topology is the same as shown in *Figure 2*. Queue is set at R1, the bottleneck of the network. Buffer sizes are given in packets. We set the longer delay to be 58ms (so the RTT of the longer delay is 30 times of the shorter one) to see queuing algorithms effect clearly.

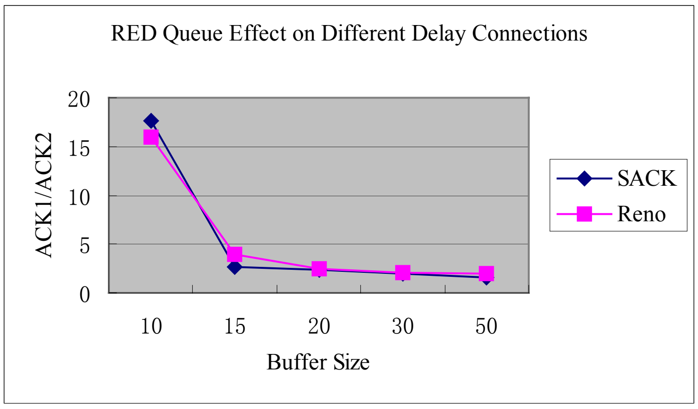
3.2.3.1 DropTail Queue



We use ACK1/ACK2 to denote the bias on longer delay connections. From the chart, we can see as the buffer size of DropTail queue increases, both Reno and SACK ACK1/ACK2 drop down to close 1, which means longer delay connection can receive more bandwidth. Increasing the buffer size in the router can make the two connections share the link more fairly. The reason is that Reno and Sack TCP need buffer space to increase window. When the buffer size is small, the fast link occupies most of the buffer. When the buffer size is large, the slow link can get some buffer space because the window of Reno and TCP oscillates due to an AIMD algorithm.

3.2.3.2 RED Queue

RED is designed for congestion avoidance. The two main parameters are *threshold* and *maxthreshold*. When the average queue size is above *threshold* but smaller than *maxthreshold*, it starts dropping packets with certain probability that is proportional to the queue size. If the average queue size exceeds *maxthreshold*, all incoming packets are dropped. We set *maxthreshold=buffer_size* and *threshold*=0.5**maxthreshold*. In this setting, we tune the buffer size as we did with DropTail queue to see the effect of buffer size changes on the fairness. We got the chart as follows.



From the chart, we got the same result as in DropTail Queue that fairness between different delay connections is improved while the buffer size grows large. It can also be seen from the figures that RED gateway needs smaller buffer size than DropTail gateway to maintain fair connections.

3.3 Fairness between Different Version TCPs when Competing on Same Link

There are different versions TCP running together on the Internet. Therefore, we are also interested in the fairness when they compete on the same connection. We carried out several simulations to test Reno, SACK and Vegas TCP from this aspect.

3.3.1 Competition when two connections running together

Simulation Design
The simulation topology is quite similar to *Topology 2*. The two connections have the same round trip time of 6ms. We varied S1 and S2 to be different TCP agents. We put DropTail Queue in R1 and set the buffer size to be 15 (in packets).

The following three graphs show the bandwidth occupancy of the two TCP connections running on the same link (R1-R2). The total bandwidth is 1.5Mbps, shown in the top red line. The horizontal axis is the simulation time (in second). The total simulation time is 120 seconds. We start to collect simulation data after 15 seconds to eliminate the transient effect.

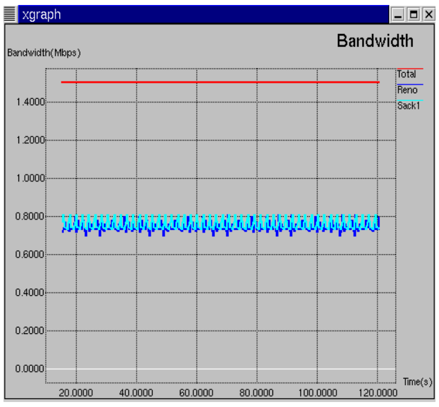


Figure 6a
Blue line- Reno, Green line-SACK

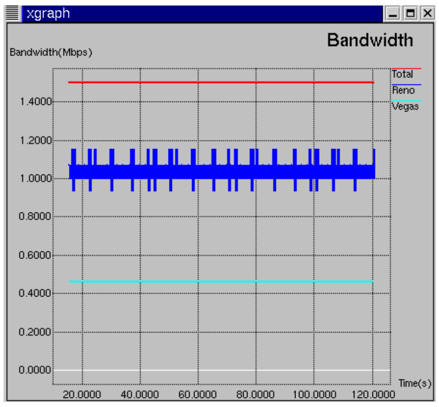


Figure 6b
Blue line – Reno, Green line – Vegas

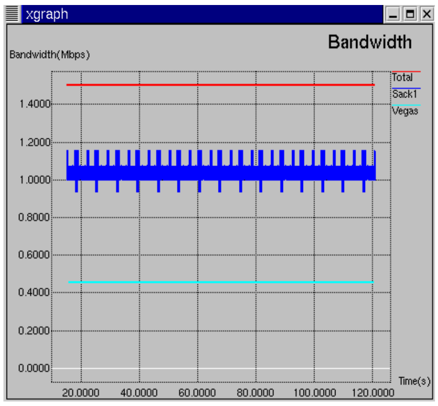


Figure 6c
Blue line – SACK, Green line - Vegas

Figure 6a shows that Reno and SACK can share the link quite fairly, almost half and half (0.75Mbps each). From *figure 7b* and *figure 7c*, we can see that TCP Vegas connection does not receive a fair share bandwidth in the presence of TCP Reno or TCP SACK.

The TCP Reno congestion avoidance scheme is aggressive in the sense that it leaves little room in the buffer for other connections, while TCP Vegas is conservative and tries to occupy little buffer space. When a TCP Vegas connection shares a link with a TCP Reno connection, the TCP Reno connection uses most of the buffer space and the TCP Vegas connection backs off, interpreting this as a sign of network congestion. That's the reason why TCP Vegas receives a small portion of the bandwidth when competing with TCP Reno. TCP SACK uses the same aggressive mechanism as TCP Reno. Therefore the behavior of TCP SACK is the same as TCP Reno when competes with TCP Vegas [6].

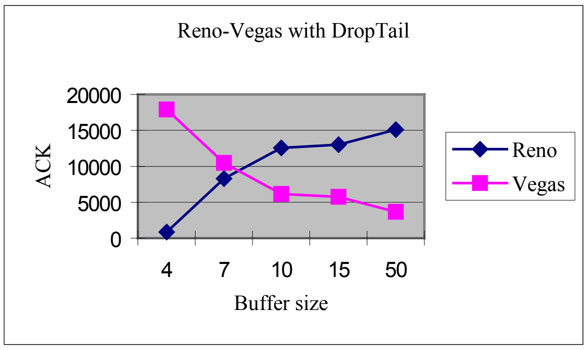
3.3.2 Queue Algorithms Effects

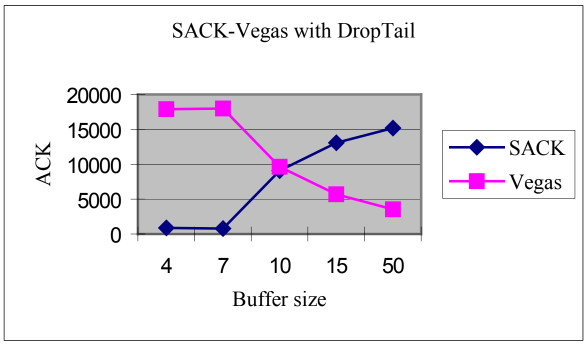
We are also interested in the effects of different queue algorithms on this unfairness. We carried out simulations to see how the buffer size and different queue algorithm will affect the behavior of the two connections.

3.3.2.1 DropTail Queue

The simulation setting is the same as section 3.3.1. We vary the buffer size at the switches to see how the bandwidth occupancy ratio changes with the buffer size. The simulation results are given as follows.

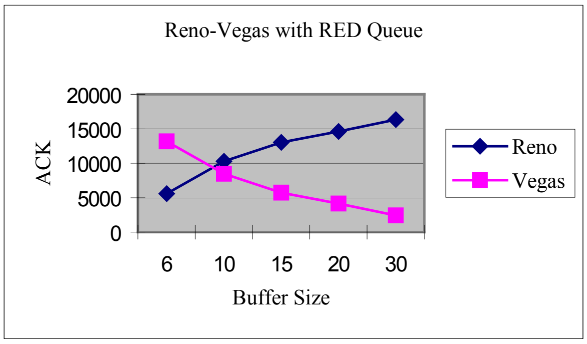
From the charts below, we can see that when the buffer sizes are small, TCP Vegas outperforms TCP Reno and TCP SACK. The reason for this is as follows. Both TCP Reno and SACK need some room in the switch buffer for oscillation in order to estimate the available bandwidth. Without the necessary room in the buffer, its performance degrades noticeably. TCP Vegas can quickly adapt to the small buffer size since it requires enough space for only a few packets [6]. When the buffer size is large enough, TCP Reno and SACK occupy much more bandwidth and leave little for TCP Vegas, which is consistent with our previous simulation results. The intersection points in the chart (about 8 for Reno-Vegas and 10 for SACK-Vegas) suggest the appropriate buffer size with which the two connections can share the link fairly. This optimal buffer size is only valid for this simulation scenario.

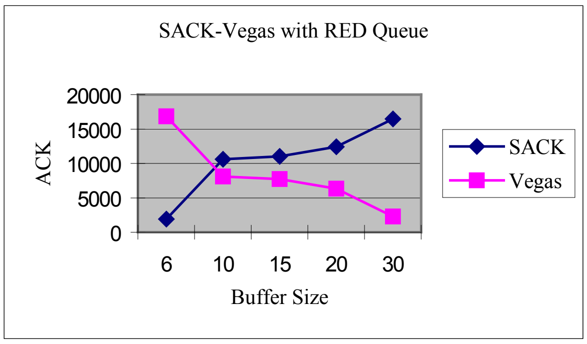




3.3.2.2 RED Queue

We also test RED queue to see the effect of parameters changes. If we set the main two parameters *threshold*=0.5**maxthresh* and *maxthresh=buffer_size* as previous section, we got the results shown in following charts when we tune the buffer size. The conclusion is similar to DropTail queue. If the buffer size is small, Vegas outperforms than Reno and SACK. As the buffer size grows up, Reno and SACK occupy more bandwidth due to their aggressive algorithms.





To our understanding, *threshold* plays a more important role in RED queuing algorithm, since when the average queue size exceeds *threshold* but smaller than the *maxthresh*, they start dropping incoming packets with certain probability that is proportional to the average queue size. Therefore, we fixed the threshold at 3 and vary the *maxthresh* to see the effects. The buffer size is set to be the same as *maxthresh*. We got the results as show in the following table.

Table 4a: Reno/Vegas Competition with RED *maxthresh* changes

Maxthresh	ACK of Reno	ACK of Vegas	Reno/Vegas
10	11105	8858	1.3
20	13158	6512	2.02
40	12211	7476	1.63
80	13767	5919	2.33

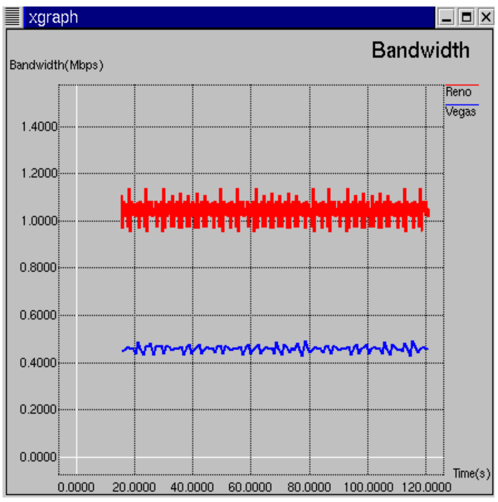
Table 4b: SACK/Vegas Competition with RED *maxthresh* changes

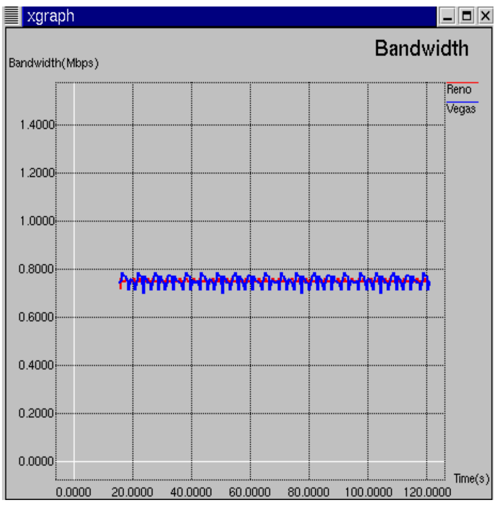
Maxthresh	ACK of SACK	ACK of Vegas	SACK/Vegas
10	11180	8484	1.32
20	11854	7829	1.51
40	13519	6150	2.2
80	12478	7209	1.73

From the above tables, we can see that if the *threshold* is fixed, changes of the *maxthresh* do not affect the fairness much. If the *threshold* is small enough (3, in our simulation), TCP Vegas can receive a relative fair share of the bandwidth. The reason behind this is that, when the threshold values are low enough, they give the connections the impression that the buffer size is smaller than it really is.

3.3.3 Other Improvement

As stated before, Vegas does not receive a fair share of the bandwidth when competing with Reno or Sack. Vegas uses a conservative algorithm to increase the congestion window, while Reno and Sack use aggressive ones. If we change the window increasing mechanism of Vegas, we can somehow enlarge its bandwidth ratio. From the description of section 2, we know that Vegas tries to maintain a buffer size between $w+\alpha$ and $w+\beta$. The α and β values are usually set to 1 and 3 (default values in NS2). We set the buffer size to be 15. α and β values are set to 3 and 6. We compare the result of that of the default values (1 and 3). It can be seen from the figure that the modified one is not biased against Vegas. We got the same conclusion for Sack1-Vegas case (result not shown here). However, the result may only be valid for this simulation scenario. What we try to point out is that it is possible to modify the algorithm of Vegas to make it more competitive in the presence of TCP Reno and SACK, while maintaining its fairness on connections with different delays. On the other hand, one could also change the algorithm of Reno or Sack to reduce the aggressive behavior, and reduce their unfairness on connections with different delays, such as the Constant-Rate (CR) algorithm.





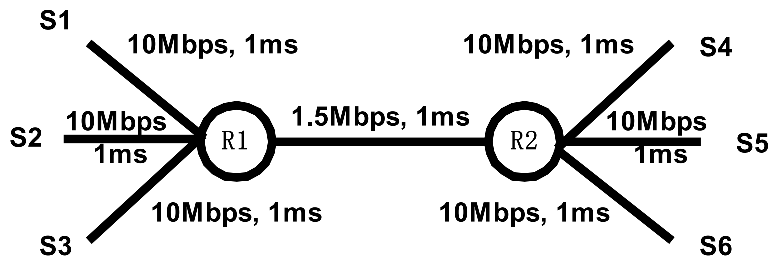
Alpha = 1, Beta = 3 Alpha = 3, Beta = 6

Figure 7. Bandwidth occupancy ratios of default and modified Vegas implementation

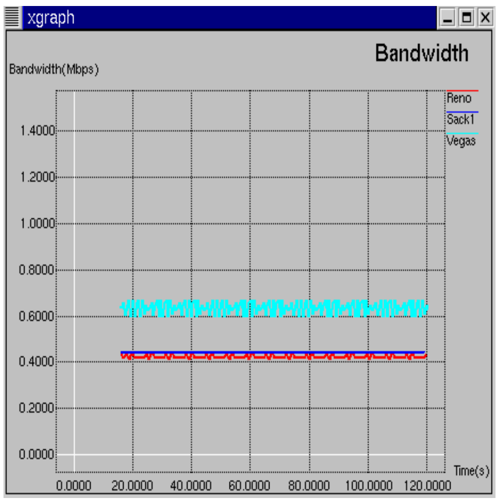
3.4 Reno-Sack1-Vegas

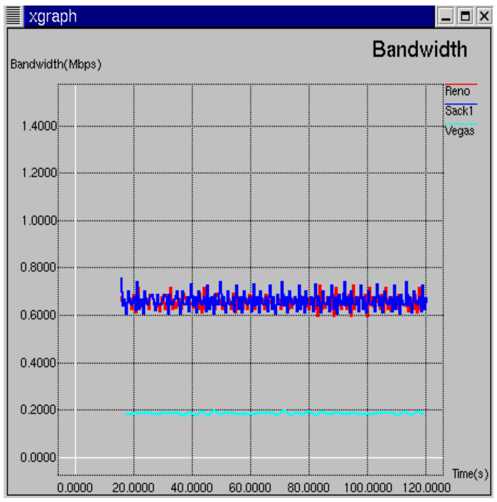
As a final part of our project, we test the case in which Reno, Sack and Vegas compete together on a bottleneck link. The simulation topology is shown in *Topology 4*, which is quite similar to *Topology 2* except that there are three senders instead of two. The total bandwidth is again set to be 1.5Mbps. Other parameters are the same as section 3.3. Results of buffer size of 15 and 30 are shown as follows. From the figures, we can see that when the buffer size is small, Vegas occupies more bandwidth than Reno and Sack. While the buffer size is large, Reno and Sack receive larger share of bandwidth than Vegas. In both cases, Reno and Sack receive about the same share of total bandwidth. This again confirms our previous conclusions. The importance of this part is that, Reno, Sack and Vegas may run simultaneously on today's Internet. Our simulation, though not complete, gives a more realistic result and conclusion.

Simulation Topology



Topology 4





Buffer Size =15

Buffer Size = 30

Figure 8. Bandwidth occupancy ratio of Reno, Sack and Vegas

4. Conclusion

In this project, we evaluate the performance of TCP Reno, TCP Vegas and TCP SACK from many aspects.

From our first subsection simulation results, we find that both TCP Vegas and TCP SACK make some performance improvements to TCP Reno. TCP Vegas achieves higher throughput than Reno and SACK for large loss rate. TCP SACK is better when more than one packets are dropped in one window. TCP Vegas causes much fewer packets retransmissions than TCP Reno and SACK.

We have also shown that TCP Vegas does lead to a fair allocation of bandwidth for different delay connections. Both TCP Reno and SACK bias against long delay connections. We also test different queuing algorithms, trying to find a method to improve the fairness. We find that with both DropTail and RED gateway, as the buffer size grows up, the longer delay connections can receive a fair share of the total bandwidth.

When competing with TCP Reno or SACK connections, TCP Vegas is penalized due to the aggressive nature of TCP Reno and SACK. We also investigated the effect of different queuing algorithms and found that when the buffer sizes are small, TCP Vegas performs better than TCP Reno and SACK, since it does not require much space in switch buffer. As the buffer sizes increase, TCP Reno and TCP SACK throughput increase at the cost of a decrease in TCP Vegas throughput. Our simulation results suggest that, for a certain link, there might be an appropriate value of buffer size (the intersection of the two ACK lines) for DropTail and RED queue to make different connections share the bandwidth in a relative fair way. The exact values may depend on the link speed and propagation delay. However, RED queue may have more parameters other than buffer size to tune for a fair allocation of bandwidth. We also suggest a change in Vegas algorithm (tune α and β values) to make Vegas more aggressive in the competition. This may be worthy of further investigation in the future work.

However, all the efforts in analysis of queuing algorithms effects lie in the gateway side of the network. There are many suggestions of modification that lie on the host side to improve the fairness. We do not have enough time to implement them in this project.

Many research efforts have been devoted to the comparison of different TCP implementations, such as Reno and Vegas, or Reno and SACK. To our knowledge, no research has been carried out for the comparison of all these three algorithms. The simulation result is consistent with our understanding of the algorithms and the literature work. Although we carry out many experiments, there are still many comparisons of TCP congestion control algorithms that can be taken. We hope to do them in future work.

Reference

[1] Kevin Fall, Sally Floyd, Simulation-based comparisons of Tahoe, Reno and SACK TCP, ACM SIGCOMM Computer Communication Review, v.26 n.3, p.5-21, July 1996. http://www.sfu.ca/~zbian/courses/cmpt885/fall96simulationbased.pdf

[2] S. Floyd, Congestion Control Principles, RFC2914, September 2000, http://www.ietf.org/rfc/rfc2914.txt

[3] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In Proceedings of the SIGCOMM '94 Symposium (Aug. 1994) pages 24-35

[4] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC2883, July 2000, http://www.ietf.org/rfc/rfc2883.txt

[5] V. Jacobson, Congestion avoidance and control, ACM SIGCOMM Computer Communication Review, v.18 n.4, p.314-329, August 1988. http://www.sfu.ca/~zbian/courses/cmpt885/congavoid.ps

[6] Jeonghoon Mo, Richard J. La, Venkat Anantharam, and Jean Walrand, Analysis and Comparison of TCP Reno and Vegas. http://www.sfu.ca/~zbian/courses/cmpt885/mo-tcp-reno-vegus.pdf

[7] V. Jacobson. "Modified TCP Congestion Avoidance Algorithm", Technical report, 30 Apr. 1990. ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

[8] Thomas R. Henderson, Emile Sahouria, Steven McCanne, Randy H. Katz. "On Improving the fairness of TCP Congestion Avoidance", Proceeding of IEEE Globecom. 98, November 1998.

Appendix

Source code list

tcpInit.tcl	---	class definition, pre-processing.
tcpTest.tcl	---	simulation scenario, result analysis.