ISSUES IN TCP VEGAS

by

Richard J. La, Jean Walrand and Venkat Anantharam

Memorandum No. UCB/ERL M99/3

10 February 1999

ISSUES IN TCP VEGAS

by

Richard J. La, Jean Walrand and Venkat Anantharam

Memorandum No. UCB/ERL M99/3

10 February 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Issues in TCP Vegas

Richard J. La, Jean Walrand, and Venkat Anantharam
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
{hyongla, wlr, ananth}@eecs.berkeley.edu

Abstract

TCP Vegas uses an estimate of the round trip time (RTT) as a decision variable. In some cases, this may cause a serious problem for its performance. We investigate these cases and suggest a possible solution to rerouting and persistent congestion problems caused by TCP Vegas.

1 Introduction

The Transmission Control Protocol (TCP) was first proposed and implemented to prevent the future congestion collapses after the congestion collapses in 1986. Since then, it has gone through several phases of improvement, and many new features such as fast retransmit and fast recovery have been added.

Recently Brakmo et al. [2] have proposed a new version of TCP, which is named TCP Vegas, with a fundamentally different congestion avoidance scheme from that of TCP Reno and claimed that TCP Vegas achieves 37 to 71 percent higher throughput than TCP Reno. Ahn et al. [1] have evaluated the performance of TCP Vegas on their wide area emulator and shown that TCP Vegas does achieve higher efficiency than TCP Reno and causes much less packet retransmissions. However, they have observed that TCP Vegas when competing with other TCP Reno connections, does not receive a fair share of bandwidth, i.e., TCP Reno connections receive about 50 percent higher bandwidth. This incompatibility property is analyzed also by Mo et al. [6]. They show that due to the aggressive nature of TCP Reno, when the buffer sizes are large, TCP Vegas loses to TCP Reno that fills up the available buffer space, forcing TCP Vegas to back off.

TCP Vegas has a few more problems that have not been discussed before, which could have a serious impact on the performance. One of the problems is the issue of rerouting. Since TCP Vegas uses an estimate of the propagation delay, *baseRTT*, to adjust its window size, it is very important for a TCP Vegas connection to be able to have an accurate estimation. Rerouting a path may change the propagation delay of the connection, and this could result in a substantial decrease in throughput. Another important issue is the stability of TCP Vegas. Since each TCP Vegas connection attempts to keep a few packets in the network, when their estimation of the propagation delay is off, this could lead the connections to inadvertently keep many more packets in the network, causing a persistent congestion. In this paper, we will discuss these problems and propose a solution to them. The remainder of the paper is organized as follows. In section 2 we describe TCP Vegas. Section 3 discusses the problem of rerouting and a proposed solution, which is followed by the problem of persistent congestion in section 4. Then, we finish with conclusions and future problems.

2 TCP Vegas

Following a series of congestion collapses starting in October of '86, Jacobson and Karels developed a congestion control mechanism, which was later named TCP Tahoe [5]. Since then, many modification have been made to TCP, and different versions have been implemented such as TCP Tahoe and Reno.

TCP Vegas was first introduced by Brakmo et al. in [2]. There are several changes made in TCP Vegas. First, the congestion avoidance mechanism that TCP Vegas uses is quite different from that of TCP Tahoe or Reno. TCP Reno uses the loss of packets as a signal that there is a congestion in the network and has no way of detecting any incipient congestion before packet losses occur. Thus, TCP Reno reacts to congestion rather than attempts to prevent the congestion.

TCP Vegas, on the other hand, uses the difference between the estimated throughput and the measured throughput as a way of estimating the congestion state of the network. We describe the algorithm briefly here. For more details on TCP Vegas, refer to [2]. First, Vegas sets *BaseRTT* to the smallest measured round trip time, and the expected throughput is computed according to

$$Expected = \frac{WindowSize}{BaseRTT},$$

where $WindowSize$ is the current window size. Second, Vegas calculates the current *Actual* throughput as follows. With each packet being sent, Vegas records the sending time of the packet by checking the system clock and computes the round trip time (RTT) by computing the elapsed time before the ACK comes back. It then computes *Actual* throughput using this estimated RTT according to

$$Actual = \frac{WindowSize}{RTT}.$$

Then, Vegas compares *Actual* to *Expected* and computes the difference

$$Diff = Expected - Actual,$$

which is used to adjust the window size. Note that *Diff* is non-negative by definition. Define two threshold values, $0 \leq \alpha < \beta$. If $Diff < \alpha$, Vegas increases the window size linearly during the next RTT. If $Diff > \beta$, then Vegas decreases the window size linearly during the next RTT. Otherwise, it leaves the window size unchanged. What TCP Vegas attempts to do is as follows. If the actual throughput is much smaller than the expected throughput, then it suggests that it is likely that the network is congested. Thus, the source should reduce the flow rate. On the other hand, if the actual throughput is too close to the expected throughput, then the connection may not be utilizing the available flow rate, and hence should increase the flow rate. Therefore, the goal of TCP Vegas is to keep a certain number of packets or bytes in the queues of the network [2, 9]. The threshold values, α and β, can be specified in terms of number of packets rather than flow rate.

Now note that this mechanism used in Vegas to estimate the available bandwidth is fundamentally different from that of Reno, and does not purposely cause any packet loss. Consequently this mechanism removes the oscillatory behavior from Vegas, and Vegas achieves higher average throughput and efficiency. Moreover, since each connection keeps only a few packets in the switch buffers, the average delay and jitter tend to be much smaller.

Another improvement added to Vegas over Reno is the retransmission mechanism. In TCP Reno, a rather coarse grained timer is used to estimate the RTT and the variance, which results in poor estimates. Vegas extends Reno's retransmission mechanism as follows. As mentioned before, Vegas records the system clock each time a packet is sent. When an ACK is received, Vegas calculates the RTT and uses this more accurate estimate to decide to retransmit in the following two situations [2]:

- When it receives a duplicate ACK, Vegas checks to see if the RTT is greater than timeout. If it is, then without waiting for the third duplicate ACK, it immediately retransmits the packet.

- When a non-duplicate ACK is received, if it is the first or second ACK after a retransmission, Vegas again checks to see if the RTT is greater than timeout. If it is, then Vegas retransmits the packet.

3 Rerouting

TCP Vegas that was first suggested by Brakmo et el. [2] does not have any mechanism that handles the rerouting of connection. In this section, we will show that this could lead to strange behavior for TCP Vegas connections. Recall that in TCP Vegas *BaseRTT* is the smallest round trip delay, which is an estimate of the propagation delay of the path .

First, if the route of a connection is changed by a switch, then without an explicit signal from the switch, the end host cannot directly detect it. If the new route has a shorter propagation delay, this does not cause any serious problem for TCP Vegas because most likely some packets will experience shorter round trip delay and *BaseRTT* will be updated. On the other hand, if the new route for the connection has a longer propagation delay, the connection will not be able to tell whether the increase in the round trip delay is due to a congestion in the network or a change in the route. Without this knowledge the end host will interpret the increase in the round trip delay as a sign of congestion in the network and decrease the window size. This is, however, the opposite of what the source should do. When the propagation delay of connection i is d_i, the expected number of backlogged packets of the connection is $w_i - r_i d_i$, where w_i is connection i's window size and r_i is the flow rate. Since TCP Vegas attempts to keep between α and β packets in the switch buffers, if the propagation delay increases, then it should increase the window size to keep the same number of packets in the buffer. Because TCP Vegas relies upon delay estimation, this could impact the performance substantially. We have simulated a simple network in order to see how TCP Vegas handles changes in propagation delay.

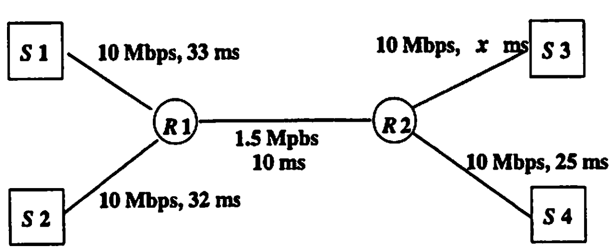

Figure 1: Simulation network

Figure 1 shows the network that was simulated. We have used real time network simulator (ns) developed at Lawrence Berkeley Laboratory for the simulation. In the simulation there are two connections. The propagation delay for connection 2 is 134 ms. We changed the delay of the link that connects $R2$ and $S3$ to simulate the change in propagation delay for connection 1, i.e., change in route. The propagation delay of connection 1 is changed from 134 ms to 88 ms at $t = 15$ sec, 88 ms to 484 ms at $t = 36$ sec, and again from 484 ms to 88 ms at $t = 60$ sec. Connection 1 starts at $t = 0$ sec and connection 2 starts at $t = 2$ sec. The packet size is set to 1,000 bytes.

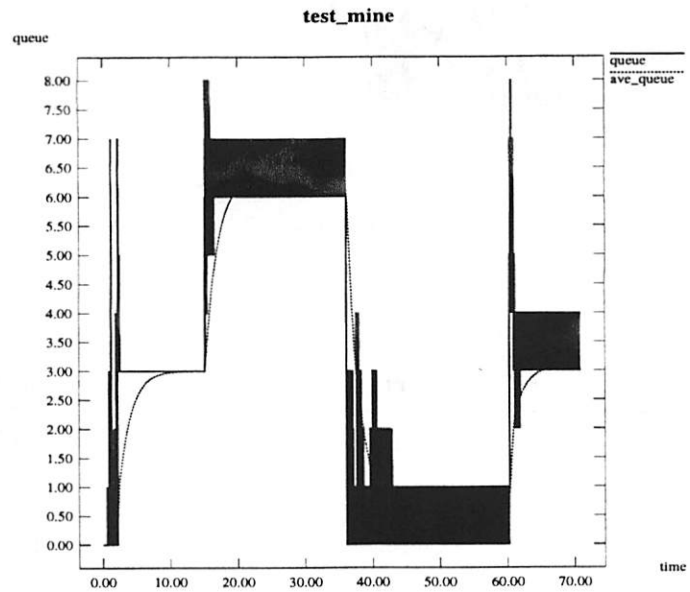

Figure 2: Queue size under TCP Vegas

Time (sec)	Vegas		Modified Vegas	
	connection1	connection 2	connection 1	connection 2
14	16.87	14	16.87	14
34	17.82	14	17.82	14
60	6.82	25.56	24.64	19.63
70	7.70	22.56	15.33	14.63

Table 1: Window sizes.

Figure 2 and 3 represent the queue length and the packet sequences. In Figure 3 connections are arranged in the increasing order from bottom to top. Each segment of a connection represents

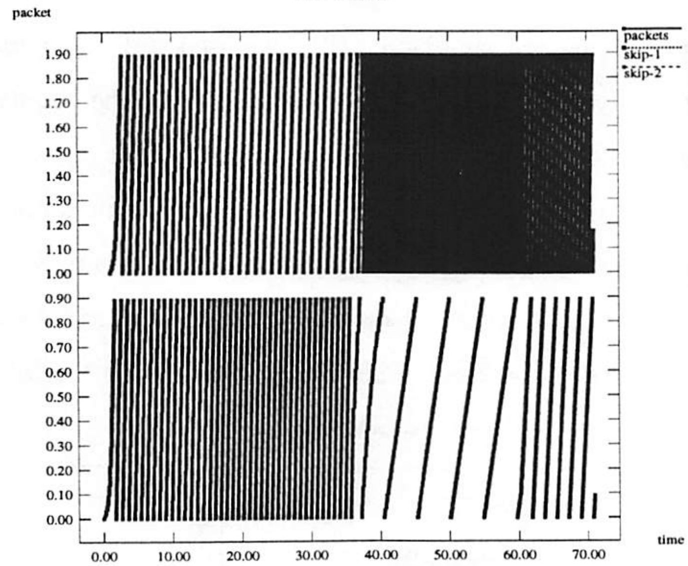

Figure 3: Packet sequence under TCP Vegas

Time (sec)	Vegas		Modified Vegas	
	connection1	connection 2	connection 1	connection 2
14	1,307	1,056	1,307	1,056
34	3,490	2,626	3,490	2,626
60	4,229	6,667	5,324	5,577
70	4,723	8,053	6,364	6,425

Table 2: Number of packets acknowledged.

90 consecutively numbered packets from the connection, and the distance between two segments tells you how fast the connection is transmitting packets. For more details on how to read the charts, refer to [4].

Figure 2 and 3 clearly show the impact of an increase in the propagation delay. The average queue size drops considerably after $t = 36$ sec. This is due to that as soon as the propagation delay increases, connection 1 interprets this as a sign of congestion and reduces its window size, which in turn decreases the backlog at the queue. This decrease in window size manifests itself in Figure 3 between $t = 36$ and $t = 40$ as an increase in distance between successive segments. The window sizes at various times are given in Table 1. Connection 1 window size drops from 17.82 at $t = 34$ sec to 6.82 at $t = 60$ sec. This results in a low flow rate as shown in Figure 3.

Since the switches in the network do not notify the connections of change in routes, this requires that the source be able to detect any such change in the route. We propose to use any lasting increase in the round trip delays as a signal of rerouting. Let us describe our mechanism first.

(1) It uses same mechanism as TCP Vegas for the first K packets. When it receives the ACK for the Kth packet, it computes the difference *diff_estimate* between *baseRTT* and RTT_K, where RTT_K is the round trip delay of the Kth packet. This difference provides a rough estimate for the increase in the round trip delay due to its own packets queued at the switch buffers. Note that an average of several differences could be used instead of one value. For simplicity we will assume that we use one value for *diff_estimate.*

(2) After the ACK for the Kth packet is received, the source checks the smallest round trip delay of every N packets, which is denoted by *baseRTTestimate.* If the difference between the minimum round trip time of N packets and *baseRTT* is larger than *diff_estimate* + $\min\{\delta \cdot \textit{baseRTT}, \gamma\}$, where $0 < \delta < 1$ and $\gamma > 0$ is a prefixed parameter, for L consecutive times, then the source interprets this as a change in propagation delay, i.e., change in route, and

(a) sets the *baseRTT* equal to the minimum round trip time of the last N packets, and

(b) sets the window size *cwnd* to

$$cwnd = cwnd \cdot \frac{baseRTTestimate}{baseRTT} + 1. \tag{1}$$

The basic idea behind this mechanism is as follows. If the minimum round trip time computed for N packets is consistently much higher than the sum of *baseRTT* and *diff_estimate*, then it is likely that the actual propagation delay is larger than the measured *baseRTT*, and it makes sense to increase *baseRTT*. However, it is possible that the increase is due to a persistent congestion in the network. This is discussed in more details in the following section. Since the increase in delay forces the source to decrease its window size, the round trip delay comes mostly from the propagation delay of the new route. Thus, the minimum round trip delay of the previous N packets is a good estimate of the new propagation delay, as is *baseRTT* for the previous route. At first glance, the new *cwnd* in (1) may seem more aggressive than it is. However, this is not true in general. Since it takes a few round trip times before the source detects an increase in propagation delay, the window size is reduced by the source before it increases it again. Thus, the window size before the update is actually smaller than what it was before the rerouting took place, and the new window size is still smaller than what it would be at an equilibrium where no connection changes its window size.

In order to compare the performance of this modified TCP Vegas to that of original TCP Vegas, we have simulated the same network in Figure 1 with the modified TCP Vegas. We have used the values of $K = 100$, $N = 20$, $\delta = 0.2$, $L = 4$, and $\gamma = 100$ ms.

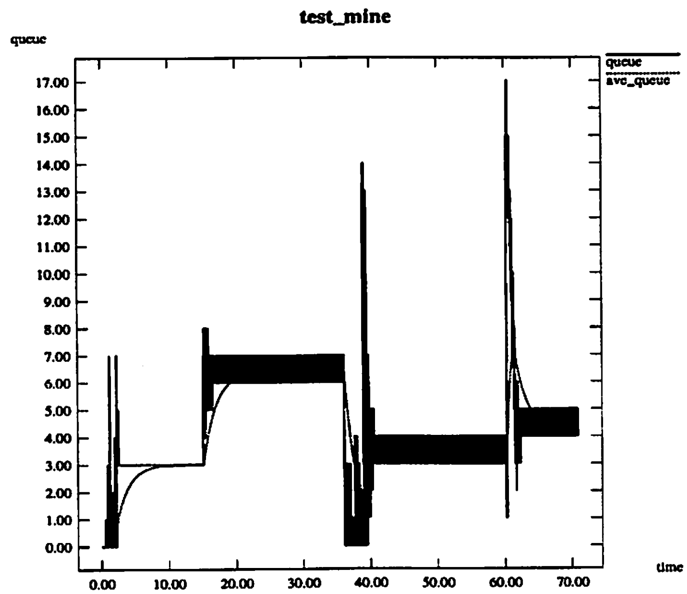

Figure 4: Queue size under modified TCP Vegas

As shown in Figure 4 and 5 under modified TCP Vegas, connection 1 quickly detects the change in delay and updates its *baseRTT* properly and resets the window size based on the new *baseRTT* as shown in Table 1. This is very easy to see in Figure 5. As in Figure 3, from $t = 36$ to $t = 38$ before connection 1 detects the increase in delay, the distance between successive segments grows. However, once connection 1 detects the increase in the propagation delay, the gap starts to close as the window size is properly reset and increases linearly for a few seconds afterwards. This results in much better performance for connection 1 as shown in Table 2. Between $t = 34$ sec and $t = 60$ sec, connection 1 successfully transmits 739 packets in TCP Vegas, whereas it transmits 1,834 packets in the modified TCP Vegas.

Moreover, in TCP Vegas, after the delay of connection 1 is reduced at $t = 60$, even though its window size increases slightly, connection 1 still does not get a fair share of bandwidth due to continuing aggressive behavior of connection 2. In the modified TCP Vegas, not only connection 1 picks up the increase in the propagation delay within a few seconds, it detects the decrease in the propagation delay at $t = 60$ better and quickly decreases its window size. Table 2 shows that the

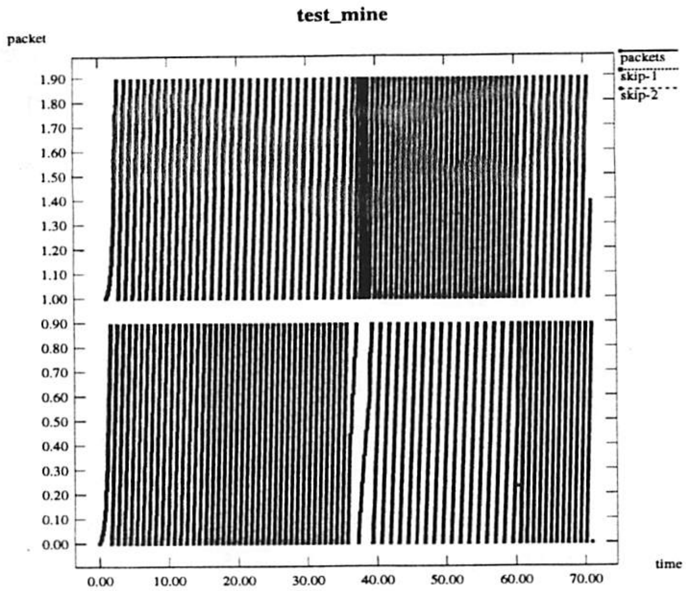

Figure 5: Packet sequence under modified TCP Vegas

window sizes at $t = 34$ sec and $t = 70$ sec are comparable as they should be. Note that the average queue size under the modified TCP Vegas is much more stable than under TCP Vegas. This is obvious from that the TCP Vegas is designed to keep between α and β packets in the network queues when working properly.

4 Persistent Congestion

Since TCP Vegas uses *baseRTT* as an estimate of the propagation delay of route, its performance is sensitive to the accuracy of *baseRTT*. Therefore, if the connections overestimate the propagation delay due to incorrect *baseRTT*, it can have a substantial impact on the performance of TCP Vegas. We will first consider a scenario where the connections overestimate the propagation delays and possibly drive the system to a persistently congested state.

Suppose that a connection starts when there are many other existing connections, the network is congested and the queues are backed up. Then, due to the queuing delay from other backlogged packets, the packets from the new connection may experience round trip delays that are considerably larger than the actual propagation delay of the path. Hence, the new connection will set the window size to a value such that it believes that its expected number of backlogged packets lies between α and β, when in fact it has many more backlogged packets due to the inaccurate estimation of the propagation delay of the path. This scenario will repeat for each new connection, and it is possible that this causes the system to remain in a persistent congestion. This is exactly the opposite of a desirable scenario. When the network is congested, we do not want the new connections to make the congestion level worse. This same situation may arise with TCP Reno or TCP Tahoe. However, it is more likely to happen with TCP Vegas due to its fine-tuned congestion avoidance mechanism.

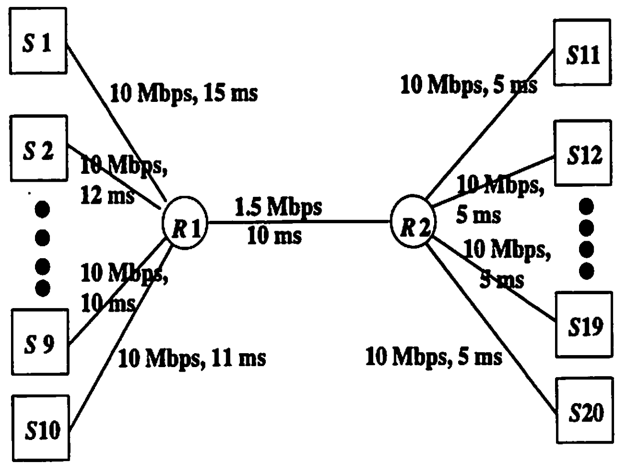

Figure 6: Simulation network

In order to demonstrate this, we have simulated a simple network shown in Figure 6. In the figure ten connections share a bottleneck. The connections have different propagation delays and start at different times. The propagation delays and the starting times are given in Table 3. The buffer sizes are set to 90. Every link has a capacity of 10 Mbps except for the link that connects $R1$ and $R2$ and has a capacity of 1.5 Mbps. We purposely picked large buffer sizes in order to allow congested network states.

Figure 7 and 8 show the queue sizes and the packet sequences. Note in Figure 7 that the jumps in the average queue size after an entrance of a new connection increase for the first 15 seconds. This is due to the inaccurate measurement of the propagation delay along the path by the connections as the network becomes more congested. The queue size increases with the number of active connections. Even though the existing connections reduce their window sizes, the incorrect measurement of the true propagation delay causes the new connections to set the window sizes to larger values than they should be. Recall that in TCP Vegas each connection is supposed to keep no more than β packets, which is usually three. However, due to the overestimation of *baseRTT* by the new connections, the average queue size increases by considerably more than three after $t = 10$ sec each time a new connection starts, and the network remains in a congested state. This becomes very obvious once the queue size settles down after tenth connection starts its transmission. At

Connection #	Propagation Delay (ms)	Starting Time (sec)
1	60	0.0
2	54	2.0
3	48	5.0
4	64	7.0
5	56	10.0
6	42	15.0
7	74	20.0
8	44	22.0
9	50	25.0
10	52	40.0

Table 3: Propagation delays and starting times.

$t = 50$ sec, the average queue size is almost 60 while there are only ten active connections. If each connection had an accurate measurement for *baseRTT*, the average queue size should not exceed 30.

Another interesting thing to note from Figure 3 and Table 4 is that the later a connection starts, the higher its flow rate is. This can be easily explained using the inaccurate *baseRTT*. The window size of a connection is determined from *baseRTT* and the flow rate. Hence, if the *baseRTT* is much larger than what it should be, then its window size will be much larger than what it should. This implies that the connection will have many more backlogged packets in the switch buffers. Since the flow rate of each connection is roughly proportional to its average queue size, the connections with larger queue sizes receive higher flow rates.

These problems are intrinsic in any scheme that uses an estimation of the propagation delay as a decision variable. For instance, the window-based scheme proposed by Mo et al. [7] adjusts the window size in such a way that it leads to a fair distribution of available bandwidth. Since this scheme uses a similar estimation of the propagation delay, the same problems will arises in implementability issues.

One might suspect that the problem of persistent congestion could be fixed once RED gateways are widely deployed over the network. RED gateways are characterized by a set of parameters,

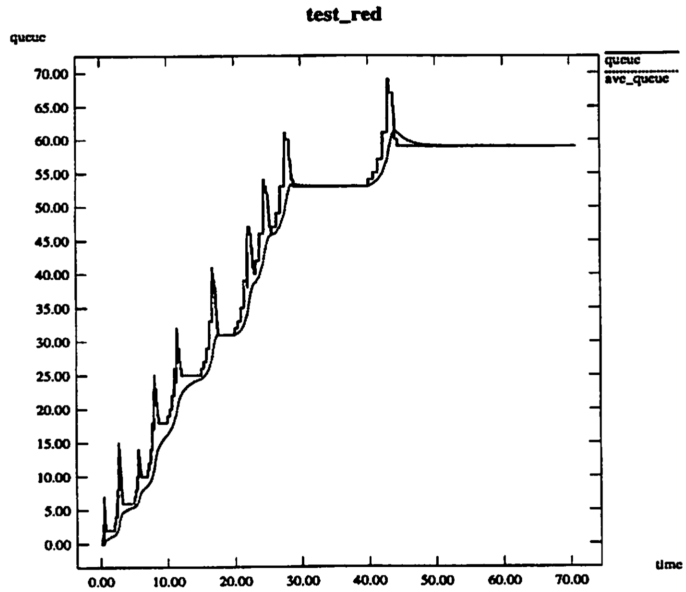

Figure 7: Queue sizes (Vegas with drop-tail gateways)

two of which are *thresh* and *maxthresh* [4]. When the average queue size is larger than *thresh* but smaller than *maxthresh*, RED gateways drop packets with certain probability that is proportional to the average queue size. If the average queue size is higher than *maxthresh*, they drop all arriving packets. The main purpose of RED gateways is to maintain a certain level of congestion. When the network becomes congested, RED gateways start dropping arriving packets. Since the packets are dropped independently, it is likely that more packets from connections with higher flow rates are dropped, forcing them to reduce their window sizes. This is simulated using the same network in Figure 6 with *thresh* = 40 and *maxthresh* = 80. The simulation is run for 140 seconds.

Figure 9 and 10 show the queue sizes and the packet sequences of the simulation results. In Figure 9 the average queue size stays around 45 when all ten connections are active which is much lower than with drop-tail gateways. Therefore, RED gateways alleviate the problem of persistent congestion. However, they do not take care of the second problem discussed above, which is the discrepancy in flow rate tied with starting times. In fact, Table 4 shows that there is very little difference between with and without RED gateways.

One of reasons why RED gateways fail to achieve a more fair distribution of the bandwidth is the following. Even though the connections with higher flow rates experience more packet drops, since the packet drops happen sparsely, TCP Vegas connections that adopt fast retransmit recover quickly without slowing down much. One way of resolving this problem is initially getting all the

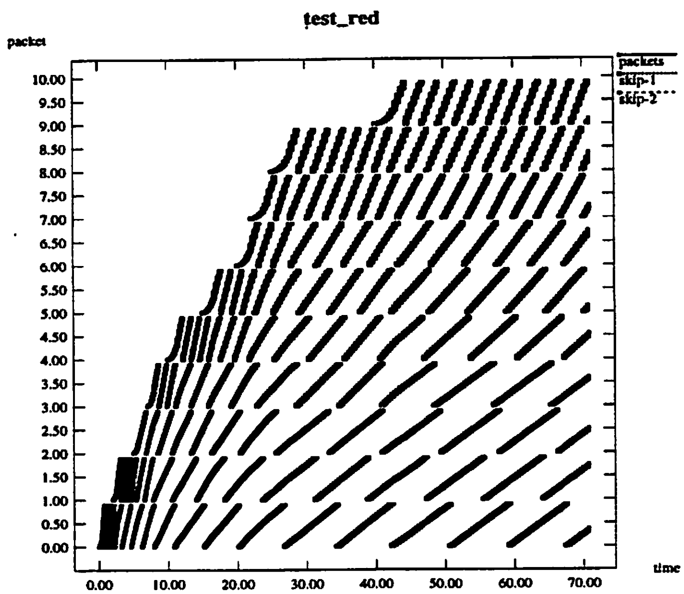

Figure 8: Packet sequences (Vegas with drop-tail gateways)

connections to overestimate propagation delays and let the connections back off almost simultaneously, without causing synchronization of the connections. This can be achieved by the same mechanism that was proposed in the previous section. When the network stays in a persistently congested state, the connections interpret the lasting increase in the round trip delay as an increase in the propagation delay and updates their *baseRTT*. This creates a temporary increase in congestion level in the network, and most connections, if not all, back off as they detect the congestion. As connections reduce their window sizes, the congestion level will come down, which allows the connections to estimate the correct *baseRTT*. Once most connections have an accurate measurement of the propagation delay, the congestion level will remain low. Since RED gateways drop packets independently, the synchronization effect will not be a big problem.

We have tested this idea with the same network to compare the performance of the connections with the modified TCP Vegas. Figure 11 and 12 show the queue sizes and the packet sequences of the connections. It is easy to see from Figure 12 and Table 4 that the available bandwidth is more evenly distributed than in the previous two cases. Especially it is obvious that the last three connections in Figure 10 receive much higher throughput than the others, whereas the same connections receive about the same throughput as others in Figure 12. The connections that start early lose some of the throughput they had as new connections start. They, however, do not continue to reduce the window size as their *baseRTT*s get updated. This enables the existing connection to

Connection #	Reno	Vegas	Vegas w/ RED	Modified Vegas w/ RED
1	3,829	2,460	2,552	4,154
2	3,278	2,037	1,904	3,190
3	2,758	1,877	1,472	2,034
4	2,520	2,229	1,725	3,032
5	1,825	2,252	2,337	3,010
6	2,563	2,366	2,510	1,957
7	2,234	2,342	2,192	2,200
8	2,680	3,094	3,386	2,340
9	2,245	3,814	3,926	2,039
10	2,179	3,687	4,141	2,179
Total	26,111	26,158	26,145	26,132

Table 4: The number of packets transmitted.

effectively compete with the new connections as expected.

Note that when the congestion level is low, the connections can get a pretty good estimate of the propagation delay without the help from the RED gateways and RED gateways do not need to drop any packets. The problem arises only when the congestion level becomes high, and this is when RED gateways are actively involved. Moreover, the temporary increase in congestion level happens only when the network is already congested and a new connection starts. Therefore, the number of packet drops should be minimal. Table 4 shows that less than 0.2 percent of the packets are dropped by RED gateways in the simulation. This clearly demonstrates that when TCP Vegas is used, RED gateways are an effective means of controlling the congestion level without sacrificing much of available bandwidth, while maintaining a fairness behavior of TCP Vegas.

5 Conclusion and Future Problems

In this paper we have discussed a few issues of TCP Vegas. We have shown that TCP Vegas could cause a strange behavior when there is a rerouting in the network and connections do not detect such change. We have demonstrated that a simple scheme that updates *baseRTT* when the round trip delay is consistently much larger than *baseRTT* results in a much better performance for the

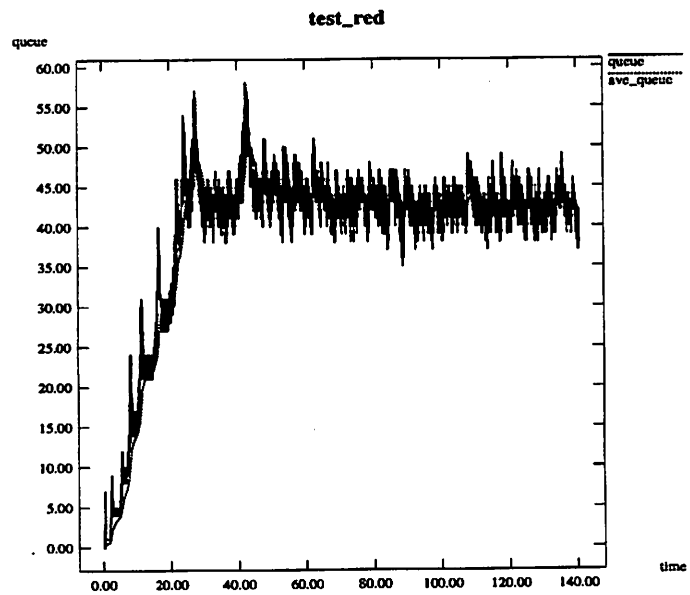

Figure 9: Queue sizes (Vegas with RED gateways)

connections that experience change in propagation delays. We have also shown that TCP Vegas could lead the network to a *persistent* congestion if connections start at different times when the network is congested. This problem could be solved using the combination of the RED gateways and the same modification proposed for the first problem of rerouting. This brings about a more even distribution of the bandwidth regardless of the starting time and guarantees that the congestion level stays around the desired congestion level by the RED gateways.

Finding the appropriate threshold values for the RED gateways, however, is still an open problem. If the threshold values are set too low, it may cause too many packet drops initially before the connections settle or the window sizes may never settle, which is undesirable. Another open question is finding the appropriate values for K, N, δ, L, and γ for the scheme that updates *baseRTT*. If *baseRTT* is updated too often, it may result in an oscillation of the window size. On the other hand, if it is not updated often enough, connections may take too long before they detect any change in route. Further studies are required on these issues.

References

[1] J. Ahn, P. Danzig, Z. Liu, and L. Yan, "Evaluation of TCP Vegas: emulation and experimen", *Computer Communication Review*, Vol. 25, No. 4, pp. 185-95, Oct. 1995.

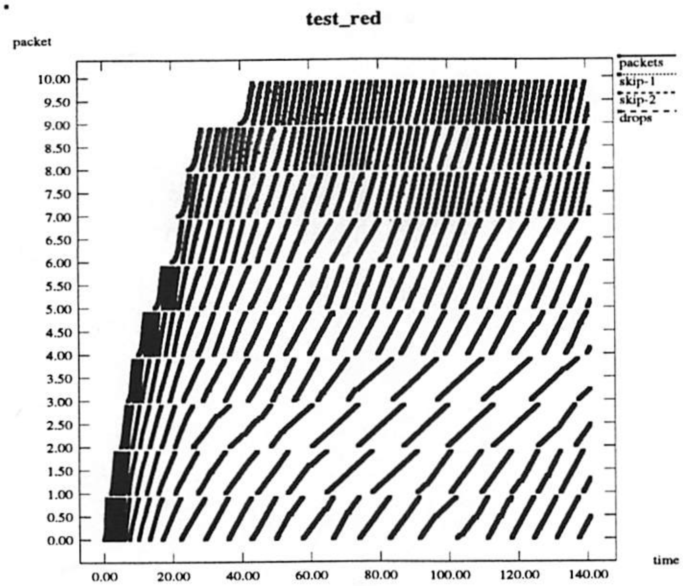

Figure 10: Packet sequences (Vegas with RED gateways)

[2] L.S. Brakmo, S. O'Malley, and L.L. Peterson. "TCP Vegas: New techniques for congestion detection and avoidance", *Computer Communication Review*, Vol. 24, No. 4, pp. 24-35, Oct. 1994.

[3] 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, No. 8, pp. 1465-80, Oct. 1995.

[4] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, Vol. 1, No, 4, pp. 397-413, August 1993.

[5] V. Jacobson, "Congestion avoidance and control.", *Computer Communication Review*, Vol. 18, No. 4, pp. 314-29, August 1988.

[6] J. Mo, R.J. La, V. Anantharam, and J. Walrand, "Analysis and Comparison of TCP Vegas.", Available at *http://www.path.berkeley.edu/ hyongla*, June 1998.

[7] J. Mo and J. Walrand, "Fair End-to-end Window-based Congestion Control", *SPIE '98 International Symposium on Voice, Video, and Data Communications*, Nov. 1998.

[8] W.R. Stevens, *TCP/IP Illustrated*, Vol. 1, Addison-Wesley Pub. Co., 1994.

[9] J. Walrand *Communication networks : a first course.* WCB/McGraw-Hill, Boston, MA, 1998