Analysis and Comparison of TCP Reno and Vegas

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

July 13, 1998

Abstract

We propose some improvements of TCP Vegas and compare its performance characteristics in comparison with TCP Reno. We demonstrate through analysis that TCP Vegas, with its better bandwidth estimation scheme, uses the network resources more efficiently and is more fair than TCP Reno. Simulation results are given and support the results of the analysis.

1 Introduction

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 [9]. Since then, many modifications have been made to the transmission control protocol (TCP) and several different versions of TCP have been implemented [9, 10].

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.

*address all correspondence to the second author

Floyd has observed that TCP Reno is biased against the connections with longer delays [5, 7] . The intuition behind 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. Obviously the connections with shorter delay can update their window sizes faster than those with longer delays, and thus steal higher bandwidths. Based on this observation, Floyd and Jacobson [6] have proposed a *constant rate adjustment* algorithm. Henderson et al. [8] have simulated a variation of this scheme and reported that if the rate of increase of the window size is not excessive, this scheme is not harmful to the other connections that use a different version of TCP. Moreover, as expected, this scheme results in better performance for the connections with longer delays. However, choosing the parameters for such algorithms is still an open problem.

In this paper, we explain the performance characteristics of TCP Vegas such as fairness and incompatibility with TCP Reno. We also discuss a few problems with the original TCP Vegas. The rest of the paper is organized as follows. In section 2 we describe the congestion detection and avoidance schemes of TCP Reno and TCP Vegas. Section 3 presents a few problems intrinsic in the original TCP Vegas and explains a possible solution to them. In section 4 we discuss the fairness of TCP Vegas using a simple closed fluid model. Section 5 compares TCP Vegas with TCP Reno and explains the incompatibility between them. We confirm our analysis with simulation results in section 6, and finish with some conclusion and future problems.

2 Congestion Avoidance Mechanism

In order to make an efficient use of network bandwidth, TCP controls its flow rate using the feedback from the network. In order to control its flow rate, TCP needs to estimate the available bandwidth in the network using a *bandwidth estimation scheme*. Among many new features implemented in TCP Vegas, the most important difference between TCP Vegas and TCP Reno lies in this bandwidth estimation scheme used to estimate the available bandwidth. As will be shown shortly, TCP Reno views packet losses as a signal of network congestion, while TCP Vegas uses the difference in the expected and actual rates to adjust its window size. This fundamental difference in the bandwidth estimation schemes enables TCP Vegas to utilize the available bandwidth more efficiently.

2.1 TCP Reno

TCP Reno induces packet losses to estimate the available bandwidth in the network. While there are no packet losses, TCP Reno continues to increase its window size by one during each round trip time. When it experiences a packet loss, it reduces its window size to one half of the current window size. This is called *additive increase and multiplicative decrease*. Additive increase and multiplicative decrease algorithm is based on the results given in [3]. They have shown that such an algorithm leads to a fair allocation of bandwidth. TCP Reno, however, fails to achieve such fairness because TCP is not a synchronized rate based control scheme, which is necessary for the convergence.

As can be easily seen, the congestion avoidance mechanism adopted by TCP Reno causes a periodic oscillation in the window size due to the constant update of the window size. This oscillation in the window size leads to an oscillation in the round trip delay of the packets. This oscillation results in larger delay jitter and an inefficient use of the available bandwidth due to many retransmissions of the same packets after packet drops occur.

The rate at which each connection updates its window size depends on the round trip delay of the connection. Hence, the connections with shorter delays can update their window sizes faster than other connections with longer delays, and thereby steal an unfair share of the bandwidth. As a result, TCP Reno exhibits an undesirable bias against the connections with longer delays [5, 7].

2.2 TCP Vegas

TCP Vegas adopts a more sophisticated bandwidth estimation scheme. It uses the difference between expected and actual flows 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. Note that 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

$$Diff = (Expected - Actual)\ BaseRTT, \tag{1}$$

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 follows:

1. First, the source 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 source 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 source, using the expected and actual flow rates, computes the estimated backlog in the queue from *Diff = (Expected-Actual) BaseRTT*.

4. Based on *Diff*, the source updates its window size as follows.

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

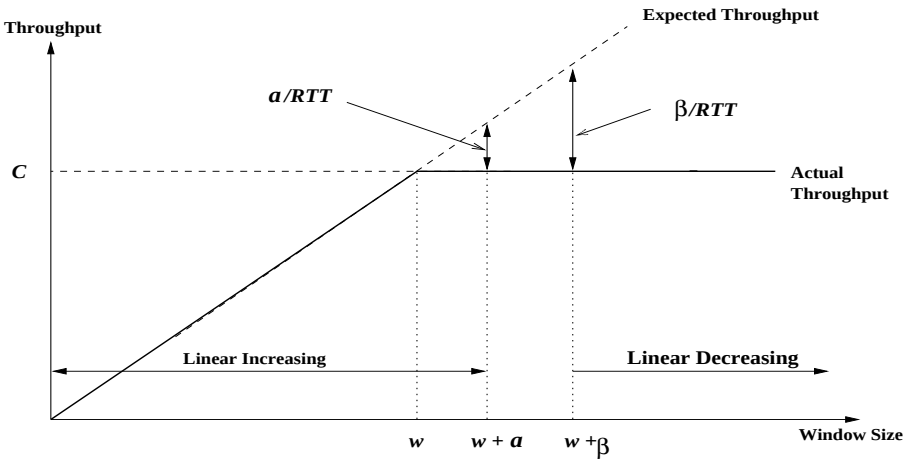

Figure 1: Window control of TCP Vegas

Figure 1 illustrates the behavior of TCP Vegas. Consider a simple network with one connection and one link with capacity C. Let $BaseRTT$ be the minimum round trip delay. The throughput of this connection is $\frac{window}{BaseRTT}$ when $window < C \times BaseRTT$. In the figure, w corresponds to the window size where $window = C \times BaseRTT$. When $window > w$, queue starts to build up and *(Expected - Actual)* > 0. TCP Vegas increases the window size by one during the next round trip time if $window < w + \alpha$ and decreases the window size by one if $window > w + \beta$. Otherwise, it leaves the window size unchanged. In the figure, *Diff* is user i's estimated backlogged queue size in the link. This will be explained in more details in section 3. 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. Thus, when there is only one connection, the window size of TCP Vegas converges to a point that lies between $w+\alpha$ and $w+\beta$. Note that 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.

3 TCP Vegas

TCP Vegas has a few 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. La et el. [12] have investigated these issues. We summarize the results here. For more details, refer to [12].

3.1 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.

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. La et el. [12] have suggested the following modification. For the first K packets, it works the same way as TCP Vegas, where K is a prefixed parameter. After the ACK for Kth packet arrives, the source keeps track of the minimum round trip delay of the N consecutive packets. If the minimum round trip time of the last $L \cdot N$ packets is much larger than the current *baseRTT*, the source updates *baseRTT* to the minimum round trip time of the last N packets and resets the congestion window size based on this new *baseRTT*.

The basic idea behind this mechanism is as follows. If the minimum round trip time computed for N packets is consistently much higher than *baseRTT*, 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 problem is briefly discussed in section 3.2. 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. The detailed description of the mechanism is given in [12] with simulation results.

3.2 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 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.

Once RED gateways are widely deployed, this problem can be handled by the same mechanism suggested in section 3.1. The intuition behind this solution is as follows. 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. For more details and simulation results, see [12].

4 Fairness of TCP Vegas

As we have mentioned before, TCP Reno is biased against the connections with long delays [5, 7, 13]. In this section we show that TCP Vegas does not suffer from this delay bias as TCP Reno does by analyzing a simple closed fluid model, and confirm the resulting analysis by simulation in section 6. We assume that the packets are infinitely divisible, i.e., we use a fluid model, and the propagation delay is deterministic.

Consider the simple network in Figure 2. There are two users with possibly different round trip delays, which share a single bottleneck link. User i has round trip delay d_i and exercises window-based flow control. Without loss of generality we assume that the capacity of the bottleneck link is normalized to one.

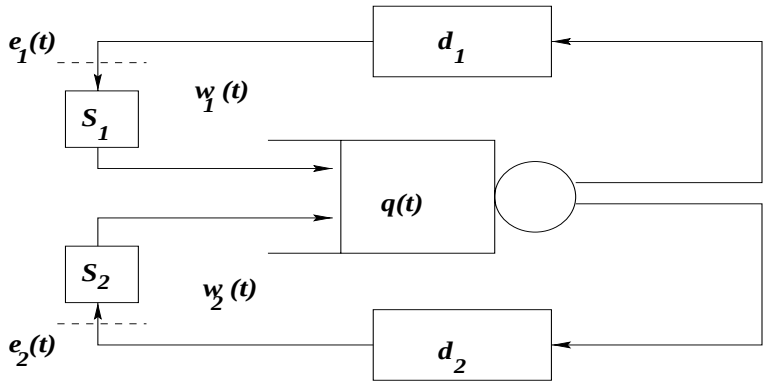

Figure 2: Network with two users

Let $q_i(t)$ denote the user i's queue size at time t, $q(t) = (q_1(t), q_2(t))$, and $z(t) = \sum_{i=1}^{2} q_i(t)$, which is the total backlog at time t. Define $u_i(t) = z(t - u_i(t)) + d_i$, which denotes the round trip delay experienced by user i's packet whose ACK arrives at the source at time t. We assume that the switch adopts the $FIFO$ service discipline and has infinite buffer size. Suppose $e_i(t)$ denotes the rate at which connection i receives ACK as shown in Figure 2 and the capacity is fully utilized, i.e., $e_1(t - d_1) + e_2(t - d_2) = 1$. Users update their window sizes once during each round trip time.

Since the window size is equal to the sum of the amount of fluid in queue and in transit, at time t we have

$$w_i(t) = q_i(t) + \int_t^{t+d_i} e_i(s)ds. \tag{2}$$

Note that the second term on the right hand side is the amount of fluid in transit at time t. Each source computes the expected backlog using the product of the propagation delay and the difference between the expected flow rate and the actual flow rate. Thus, the estimated amount of backlogged fluid computed by connection i is

$$Diff_i = w_i(t) - \frac{d_i}{u_i(t)} \int_{t-u_i(t)}^{t} e_i(s)ds. \tag{3}$$

A heuristic argument for the convergence of window sizes after some time t_0 to a point such that

$$\alpha \leq w_i(t) - \frac{d_i}{u_i(t)} \int_{t-u_i(t)}^{t} e_i(s)ds \leq \beta \tag{4}$$

is given in Appendix. Now suppose that window sizes satisfy equation (4). If we substitute (2) in (4), we have

$$\alpha \leq q_i(t) + \int_t^{t+d_i} e_i(s)ds - \frac{d_i}{u_i(t)} \int_{t-u_i(t)}^{t} e_i(s)ds \leq \beta. \tag{5}$$

Once the window sizes satisfy (4), the sources do not change the window size. Suppose that the system is quasi-static and $e_i(t)$ is relatively constant. Then, the second and third terms in (5) cancel each other out, and it yields

$$\alpha \leq q_i(t) \leq \beta. \tag{6}$$

Note that under the quasi-static assumption the queue size and the estimated queue size in (3) of connection i do not depend on d_i. Since the server exercises the FIFO service discipline, the flow rate of each connection is proportional to its queue size. Therefore, the throughput of a connection does not depend on its propagation delay, and there is no bias in favor of connections with short delays.

If a connection with a larger delay wants to keep the same amount of fluid in the queue and similar throughput as a connection with a much shorter delay, it needs to have a bigger window size than the connection with a shorter delay because it needs more fluid in transit. This can be easily seen from (2). If both connections have same throughput, then the second term in (2) is directly proportional to the delay of the connection. Therefore, when all connections are receiving similar throughput, the window sizes of the connections should be roughly proportional to their propagation delays.

If we assume that the throughput and the queue size of each connection is relatively constant, then from (2) we get

$$w_i(t) = q_i(t) + e_i(t) \cdot d_i \tag{7}$$

and

$$e_i(t) = \frac{window\ size}{round\ trip\ delay} = \frac{w_i(t)}{z(t) + d_i}. \tag{8}$$

Since we have assumed that the capacity is fully utilized, from (8) we have

$$\frac{w_1(t)}{z(t) + d_1} + \frac{w_2(t)}{z(t) + d_2} = 1. \tag{9}$$

Hence, given the window sizes $w_i(t)$, we can compute $z(t)$ by solving (9). By substituting the solution of (9) in (8) we can compute the throughput of each connection. Plugging in the solution of (8) in (7) yields the queue size of the connection. Thus, under the quasi-static assumption, we can compute the throughput and queue size of the connections, given the window sizes.

The simulation results show that the window sizes converge to a point such that the queue sizes computed from (7) - (9) are between α and β for both connections. Figure 4 shows simulation results and Figure 3 describes how to read the plots in Figure 4. In Figure 3, α_i and β_i lines for

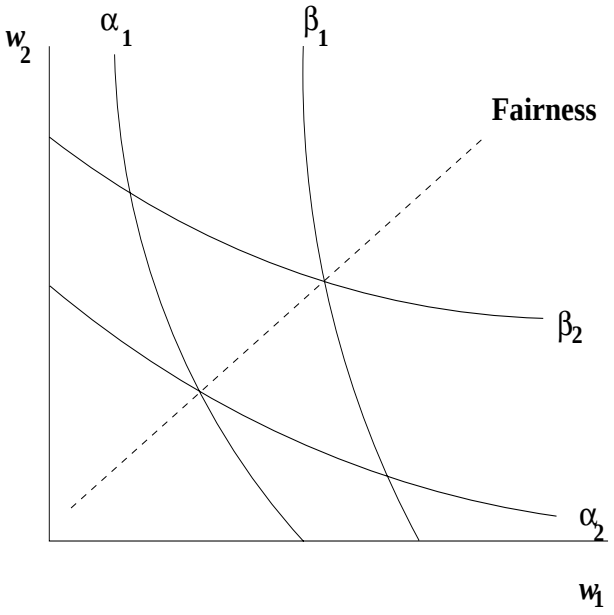

Figure 3: Convergence region

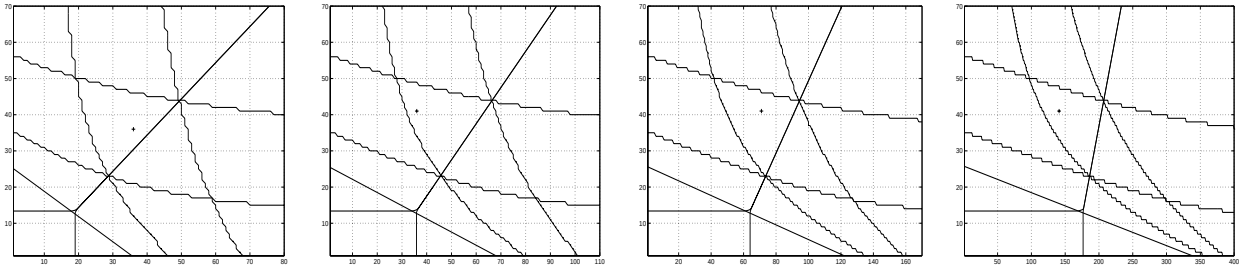

Figure 4: Converging point of TCP-Vegas with different delays.

user i denote the sets of window size pairs $\{(w_1, w_2)|w_i - e_i d_i = \alpha\}$ and $\{(w_1, w_2)|w_i - e_i d_i = \beta\}$, respectively, where e_i is connection i's throughput computed from (7) - (9). The fairness line is the set of window size pairs such that the throughputs of the users are the same, i.e.,$\{(w_1, w_2)|e_1 = e_2\}$. In each plot of Figure 4 '+' denotes the converging point of TCP Vegas. As we can see, in each case it converges to a point (w_1, w_2) that satisfies

$$\alpha \leq q_i(w_1, w_2) \leq \beta, \ i = 1, 2, \tag{10}$$

where $q_i(w_1, w_2)$ is the queue size of connection i computed using (7) - (9). The numbers on the X-axis and Y-axis are the window sizes $\times$ 10. The delay of connection 1 (Y-axis) is 12.9 ms in all plots and that of connection 2 (X-axis) is 18.9 ms, 36.9 ms, 66.9 ms, and 186.9 ms, respectively, from left to right.

5 Incompatibility of TCP Vegas with TCP Reno

There has been some previous work that demonstrates that in many cases TCP Vegas performs better than other implementations of TCP [2, 1]. However, Ahn et al. have observed that when a TCP Vegas user competes with other users that use TCP Reno, it does not receive a fair share of bandwidth due to the conservative congestion avoidance mechanism used by TCP Vegas.

TCP Reno continues to increase the window size until a packet is lost. Packet losses occur mainly due to buffer overflows. This bandwidth estimation mechanism results in a periodic oscillation of window size and buffer-filling behavior of TCP Reno. Thus, while TCP Vegas tries to maintain a smaller queue size, TCP Reno keeps many more packets in the buffer on the average, stealing higher bandwidth.

Let $q_v(t)$ and $q_r(t)$ denote the number of Vegas and Reno packets in the buffer at time t, and B be the buffer size. If we assume $q_v(t) \approx k \leq \beta$, $q_r(t)$ ranges from $[0, B-k]$. Assume that $q_r(t)$ is distributed uniformly on the interval $[0, B-k]$. Then the average queue size of the Reno user $\bar{q}_r(t)$ is approximately $\frac{B-k}{2}$. The ratio of TCP Vegas throughput to TCP Reno throughput is roughly $\frac{2k}{B-k}$. When B is relatively large, which is usually the case, then it is obvious that TCP Reno gets much higher bandwidth than TCP Vegas. We will confirm this analysis with simulation results in the next section.

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. This may be one of reasons why TCP Vegas is not widely deployed despite its many desirable properties.

6 Simulation Results

In order to verify the observations we have made through the analysis of simple networks, we have run simulations, using real network simulator (ns) developed at Lawrence Berkeley Laboratory [4]. We will discuss the simulation results in this section.

6.1 Fairness of TCP Vegas

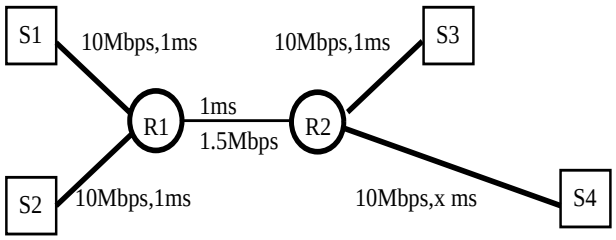

Figure 5: Network Topology

Figure 5 shows the topology of the network that was simulated to demonstrate the fairness of TCP Vegas. In the figure, the circles denote finite-buffer switches and the squares denote 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. We vary the propagation delay of the link that connects R2 and S4, which is denoted by x in Figure 5, in order to see the effect of different delays on the throughputs.

x	w_1	w_2	ACK_1	ACK_2	$\frac{max(ACK_1,ACK_2)}{min(ACK_1,ACK_2)}$
4	3.5	3.5	21,425	16,068	1.33
13	4	7	17,522	19,965	1.14
22	4	7	20,061	17,427	1.15
58	4	13	19,507	17,973	1.09
148	4	30	21,068	16,398	1.29

Table 1: TCP Vegas with varying propagation delays.

x	ACK_1	ACK_2	ACK_1/ACK_2
4	21,100	15,637	1.35
13	25,460	11,785	2.16
22	25,684	11,672	2.20
58	34,429	2,627	13.11
148	35,598	959	37.12

Table 2: TCP Reno with varying propagation delays.

Connections 1 and 2 start at time 0 second and 0.5 seconds, respectively, and the simulation is

run for 200 seconds. Tables 1 and 2 show the simulation results. In each table ACK_1 and ACK_2 are the number of packets that have been acknowledged by the corresponding receiver for connections 1 and 2. The last column shows the ratio of $\max(ACK_1, ACK_2)$ to $\min(ACK_1, ACK_2)$.

Tables 1 and 2 clearly show that the delay bias of TCP Reno becomes very noticeable when the delay difference becomes large, while TCP Vegas does not show any such delay bias. The ratio of throughputs under TCP Vegas ranges from 1.09 to 1.33, whereas throughput ratio under TCP Reno increases almost linearly with delay ratio. This is consistent with our analysis given in previous sections and confirms that TCP Vegas does not suffer from the delay bias as TCP Reno does. We can also verify that the expected backlog for TCP Vegas connections at the converging point is between α and β from (7) - (9).

6.2 Incompatibility of Vegas with Reno

In the second part of simulation, we have simulated a network with two connections with same round trip delay of 24 ms, where one is a TCP Reno connection and the other is a TCP Vegas connection, in an attempt to verify the incompatibility of TCP Vegas with TCP Reno. We vary the buffer size at the switches to see how the relative performance of the TCP Vegas connection changes with the buffer size. The simulation results are given in Table 3. The fourth column of the table is the ratio of throughputs, and the last column is the interval computed from $\frac{B-k}{2k}$ as described in section 4, with $k = 1$ and 3, since α and β in TCP Vegas are usually set to 1 and 3, respectively, and k can be any number between α and β.

Buffer Size	ACK of Reno	ACK of Vegas	Reno/Vegas	Interval
4	13,010	24,308	0.535	[0.167, 1.5]
7	16,434	20,903	0.786	[0.667, 3.0]
10	22,091	15,365	1.438	[1.167, 4.5]
15	25,397	12,051	2.107	[2.0, 7.0]
25	30,798	6,621	4.652	[3.667, 12.0]
50	34,443	2,936	11.73	[7.833, 24.5]

Table 3: Throughput of Vegas vs. Reno with drop-tail gateways.

It is clear from Table 3 that TCP Vegas connection does not receive a fair share of bandwidth in the presence of a TCP Reno connection unless the buffer sizes are extremely small. Moreover, the

ratios of the throughputs fall well within the region computed from the equation $\frac{B-k}{2k}$, confirming the validity of our analysis.

6.3 RED Gateways

The simulation results in the previous subsection suggest a way of making TCP Vegas more competitive in the presence of TCP Reno connections. Note in Table 3 that when the buffer sizes are small, TCP Vegas outperforms TCP Reno. The intuition behind this is as follows. TCP Reno needs 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, on the other hand, quickly adapts to the small buffer size since it requires enough space for only a few packets.

This key observation tells us that Random Early Detection (RED) gateways could be used to make TCP Vegas more compatible with TCP Reno in a competitive environment [7]. RED gateways are characterized by a set of parameters, two of which are *threshold* and *maxthreshold*. RED gateways maintain an estimate of average queue size. When the average queue size exceeds *threshold* but is smaller than *maxthreshold*, they start dropping packets with certain probability that is proportional to the average queue size. If the average queue size exceeds *maxthreshold*, they drop all packets. Hence, if the threshold values are set low enough, they give the connections an impression that the switch buffer size is smaller than it really is. This is discussed in more details in [11].

Threshold	Maxthreshold	ACK of Vegas	ACK of Reno
3	6	61,069	57,774
4	8	54,259	64,305
5	10	50,823	70,020
7	14	44,469	77,202
10	20	34,081	87,379

Table 4: Throughput of Vegas vs. Reno with RED gateways.

The same network used in section 6.2 is simulated again with RED gateways instead of drop-tail gateways. Each simulation is run for 100 seconds, and the buffer size is fixed at 25. The buffer size, however, plays only a minor role if maxthreshold value is smaller than the buffer size. As

summarized in Table 4, it is easy to see how the threshold values affect the relative performance of TCP Vegas against TCP Reno. As the threshold values increase, TCP Reno throughput increases at the cost of a decrease in TCP Vegas throughput. Also note that the total throughput increases slightly with the threshold values. This is obviously due to the decrease in the number of dropped packets.

7 Conclusions and Further Problems

In this paper we have analyzed the performance of TCP Vegas in comparison with TCP Reno. We have shown that TCP Vegas does lead to a fair allocation of bandwidth and explained some of other characteristics, using the congestion avoidance mechanism adopted by TCP Vegas. We have demonstrated that TCP Vegas does not suffer from the delay bias as TCP Reno does through both analysis and simulations. TCP Vegas achieves better performance than TCP Reno since its bandwidth estimation does not rely on packet losses in order to estimate the available bandwidth in the network. However, when competing with other TCP Reno connections, TCP Vegas gets penalized due to the aggressive nature of TCP Reno.

We are currently investigating the possibility of deploying TCP Vegas, adopting gateway control such as RED gateways. We model the problem both as a single shot game and as an infinitely repeated game. We suspect that we can provide the TCP Reno users with a proper incentive to switch to TCP Vegas by using RED gateways to penalize the users when the average queue sizes fluctuate at the gateways in the presence of TCP Reno users. This problem will be discussed in another paper.

References

[1] J.S. Ahn, P.B. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: emulation and experiment. *IEEE Transactions on Communications*, 25(4):185–95, Oct 1995.

[2] 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*, 13(8):1465–80, October 1995.

[3] D. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

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

[5] S. Floyd and V. Jacobson. Connection with multiple Congested Gateways in packet-Switched Networks,Part1: One-way Traffic". *ACM Computer Communication Review*, 21(5):30–47, August 1991.

[6] S. Floyd and V. Jacobson. On Traffic Phase Effects in packet Switched Gateways. *Internetworking:Research and Experience*, 3(3):115–156, September 1993.

[7] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[8] T. Henderson, E. Sahouria, S McCanne, and R. Katz. Improving Fairness of TCP Congestion Avoidance . In *EECS, UC Berkeley, May '98 (submitted to GLOBECOM '98)*, August 1998.

[9] V. Jacobson. Congestion avoidance and control . *Computer Communication Review*, 18(4):314–29, August 1988.

[10] V. Jacobson. Modified TCP Congestion Avoidance Algorithm. Technical report, April 1990.

[11] Richard J. La, Jeonghoon Mo, Jean Walrand, and Venkat Anantharam. A Case for TCP Vegas and Gateways using Game Theoretic Approach. Available at http://www.path.berkeley.edu/~ hyongla (submitted to INFOCOM '99), July 1998.

[12] Richard J. La, Jean Walrand, and Venkat Anantharam. Issues in TCP Vegas. Available at http://www.path.berkeley.edu/~ hyongla, June 1998.

[13] A. Mankin. Random Drop Congestion Control. In *Proc. SIGCOMM'90*, pages 1–7, Philadelphia, PA, September 1990.

A Convergence of Window Sizes

In this appendix, we give a heuristic argument for the convergence of window sizes to a point (w_1, w_2) such that

$$\alpha \leq Diff_i = w_i(t) - \frac{d_i}{u_i(t)} \int_{t-u_i(t)}^{t} e_i(s)ds \leq \beta \tag{11}$$

During each round trip time source i computes the estimated backlog

$$Diff_i = w_i(t) - \frac{d_i}{u_i(t)} \int_{t-u_i(t)}^{t} e_i(s)ds, \tag{12}$$

and uses this estimation of the queue size to adjust its window size. We assume that $e_i(t)$ and $q_i(t)$ change slowly and remain steady, and the average converges. This is not an unreasonable assumption because TCP Vegas updates its window size at most by one during one round trip time. Therefore, $e_i(t)$ and $q_i(t)$ do not change too fast. Now suppose that the sources use the average throughput to adjust their window sizes. For the simplicity of notation, we omit the dependency on time t. We have the following equations from (12) and the assumption.

$$Diff_i = w_{i,avg} - e_{i,avg} \cdot d_i \tag{13}$$

$$e_{i,avg} \approx \frac{w_{i,avg}}{z_{avg} + d_i} \tag{14}$$

If we assume that the capacity is fully utilized, we get

$$1 = e_{1,avg} + e_{2,avg} \approx \frac{w_{1,avg}}{z_{avg} + d_1} + \frac{w_{2,avg}}{z_{avg} + d_2}. \tag{15}$$

Hence, given the window sizes $w_{i,avg}$ we can compute an approximate value for z_{avg} from (15). The solution of (15) can be used to compute $e_{i,avg}$ and $Diff_i$ from (13) and (14). The sources update their window size based on $Diff_i$.

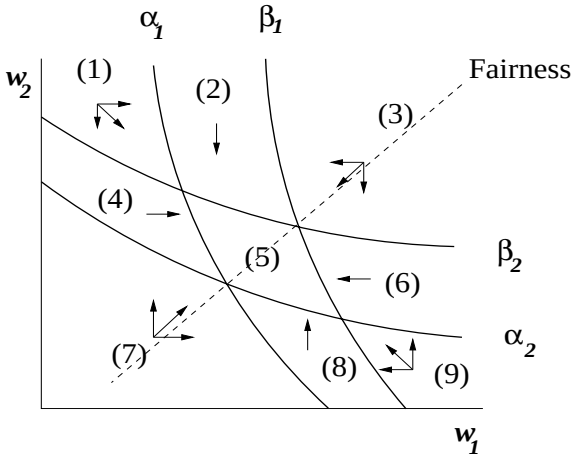

Figure 6: Convergence region of TCP Vegas

Figure 6 illustrates the convergence region of TCP Vegas. In the figure, α_i and β_i lines for user i denote the sets of window size pairs $\{(w_1, w_2)|Diff_i = \alpha\}$ and $\{(w_1, w_2)|Diff_i = \beta\}$, respectively,

from equations (13) - (15). The fairness line is the set of window size pairs such that the throughputs of the users are the same, i.e.,$\{(w_1, w_2)|e_1 = e_2\}$. User 1 increases its window size in regions (1), (4), and (7), and decreases its window size in regions (3), (6), and (9). Similarly, user 2 increases its window size in regions (7), (8), and (9), and decreases its window size in regions (1), (2), and (3).

Figure 6 is a typical plot of convergence region for two users, and the same geometric picture can be easily extended to a case with more than two users. The only region in the figure where neither user updates its window size is region (5). The arrows in other regions indicate the directions in which the window sizes may get updated.

We will show in Theorem 1 that the window sizes converges to a point in region (5) within a finite amount of time, assuming that the distance between α_i and β_i lines is sufficiently large compared to $\delta > 0$, that is the amount by which users update their window sizes. Suppose $(w_1, w_2) = (w_1(0), w_2(0))$. Let

$$A = \{(w_1 + \delta n_1, w_2 + \delta n_2) \mid 0 \leq w_1 + \delta n_1 \leq w_{1,max}, 0 \leq w_2 + \delta n_2 \leq w_{2,max}, n_1, n_2 \in Z\},$$

where $w_{1,max}$ and $w_{2,max}$ are the maximum congestion window sizes declared by the receivers. Then, A is the set of all possible window size pairs. If $w_{i,max}$ and $w_{2,max}$ are finite, then $N = \mid A \mid$ is finite. We will first state a lemma that will be used in the proof of Theorem 1. Suppose that the distance between α_i and β_i lines is at least $\sqrt{2}\delta$.

Lemma 1 *No window size pair outside region (5) can be visited twice.*

Proof: Suppose that there exists a window size pair that could be visited twice outside region (5). Then, this implies that there exists a loop generated by window size updates as shown in Figure 7. We will now show that no such loop can exist.

In regions (1) through (6), user 2 never increases its window size. Hence, if user 2 updates in regions (1) through (3), then it is not possible to return to the same windows size pair since we have assumed that the distance between α_2 line and β_2 line is larger than δ. Moreover, if user 1 updates its window size in either region (1) or (3), then the previous window size pair cannot be revisited without reaching either region (2) or (5) because user 1 does not update its window size in opposite direction in the same region. However, once the window sizes reach a point in region (2) or (5), then for the same reason given above, it is not possible to return to the previous window size pair because at most only user 2 will update its window size and decrease it. Therefore, it

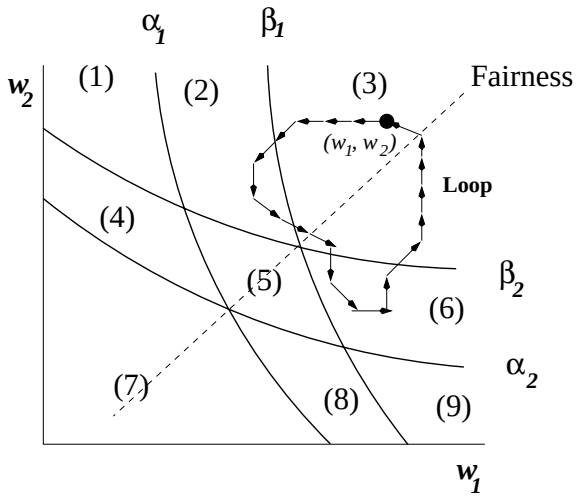

Figure 7: An example of a loop

is not possible for a window size pair in region (1) through (3) to be visited more than once. By symmetry between users 1 and 2, it is not possible to have such a loop for a window size pair in region (3), (6), and (9).

Similarly, since user 2 does not decrease its window size in region (4) through (9), none of window size pairs in regions (7) through (9) can be visited twice. Again by symmetry, the same is true for regions (1), (4), and (7). Therefore, none of the window size pairs outside region (5) can be visited more than once. ■

Theorem 1 *The window sizes converge to a point in region (5) within a finite amount of time, starting from a point in the region bounded by* $w_{1,max}$ *and* $w_{2,max}$.

Proof: From Lemma 1 we know that no window size pair outside region (5) can be visited more than once. Therefore, if $(w_1, w_2) = (w_1(0), w_2(0))$ and $N = \mid A \mid$, then assuming that time between window size updates of each connection is finite, the window sizes converge to a point in (5) because there are at most N-1 other pairs the window size pair could take on before it reaches a point in region (5), and once the window sizes are in region (5), they do not change. ■