TCP Vegas: New Techniques for Congestion Detection and Avoidance

Lawrence S. Brakmo Sean W. O'Malley Larry L. Peterson*

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

Vegas is a new implementation of TCP that achieves between 40 and 70% better throughput, with one-fifth to one-half the losses, as compared to the implementation of TCP in the Reno distribution of BSD Unix. This paper motivates and describes the three key techniques employed by Vegas, and presents the results of a comprehensive experimental performance study—using both simulations and measurements on the Internet—of the Vegas and Reno implementations of TCP.

1 Introduction

Few would argue that one of TCP's strengths lies in its adaptive retransmission and congestion control mechanism, with Jacobson's paper [4] providing the cornerstone of that mechanism. This paper attempts to go beyond this earlier work; to provide some new insights into congestion control, and to propose modifications to the implementation of TCP that exploit these insights.

The tangible result of this effort is a new implementation of TCP that we refer to as TCP *Vegas*. This name is a take-off of earlier implementations of TCP that were distributed in releases of 4.3 BSD Unix known as Tahoe and Reno; we use Tahoe and Reno to refer to the TCP implementation instead of the Unix release. Note that Vegas does not involve any changes to the TCP specification; it is merely an alternative implementation that interoperates with any other valid implementation of TCP. In fact, all the changes are confined to the sending side.

*This work supported in part by National Science Foundation Grant IRI-9015407 and ARPA Contract DABT63-91-C-0030.

The main result reported in this paper is that Vegas is able to achieve between 40 and 70% better throughput than Reno.[1] Moreover, this improvement in throughput is not achieved by an aggressive retransmission strategy that effectively steals bandwidth away from TCP connections that use the current algorithms. Rather, it is achieved by a more efficient use of the available bandwidth. Our experiments show that Vegas retransmits between one-fifth and one-half as much data as does Reno.

This paper is organized as follows. Section 2 outlines the tools we used to measure and analyze TCP. Section 3 then describes the techniques employed by TCP Vegas, coupled with the insights that led us to the techniques. Sections 4 and 5 present a comprehensive evaluation of Vegas' performance; the former reports simulation results and the latter gives preliminary numbers for measurements of TCP running over the Internet. Finally, Section 6 discusses relevant issues and Section 7 makes some concluding remarks.

2 Tools

This section briefly describes the tools used to implement and analyze the different versions of TCP. All of the protocols were developed and tested under the University of Arizona's x-kernel framework [3]. Our implementation of Reno was derived by retrofitting the BSD implementation into the x-kernel. Our implementation of Vegas was derived by modifying Reno.

2.1 Simulator

Many of the results reported in this paper were obtained from a network simulator. Even though several good simulators are available—e.g., REAL [9] and Netsim [2]—we decided to build our own simulator based on the x-kernel.

[1] We limit our discussion to Reno, which is both newer and better performing than Tahoe. Also note that in terms of the congestion-related algorithms, Reno is roughly equivalent to the Berkeley Network Release 2 (BNR2) implementation of TCP.

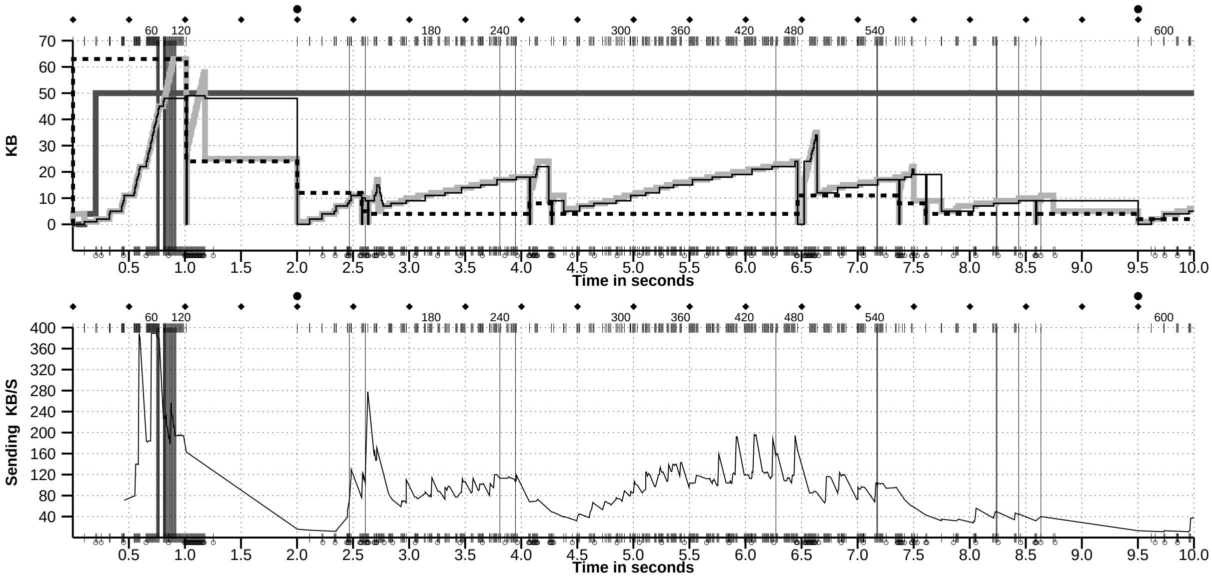

Figure 1: TCP Reno Trace Examples.

In this environment, actual x-kernel protocol implementations run on a simulated network. Specifically, the simulator supports multiple hosts, each running a full protocol stack (TEST/TCP/IP/ETH), and several abstract link behaviors (point-to-point connections and ethernets). Routers can be modeled either as a network node running the actual IP protocol code, or as an abstract entity that supports a particular queuing discipline (e.g., FIFO).

The x-kernel-based simulator provides a realistic setting for evaluating protocols—each protocol is modeled by the actual C code that implements it rather than some more abstract specification. It is also trivial to move protocols between the simulator and the real world, thereby providing a comprehensive protocol design, implementation, and testing environment.

One of the most important protocols available in the simulator is called TRAFFIC—it implements TCP Internet traffic based on *tcplib* [1]. TRAFFIC starts conversations with interarrival times given by an exponential distribution. Each conversation can be of type TELNET, FTP, NNTP, or SMTP, each of which expects a set of parameters. For example, FTP expects the following parameters: number of items to transmit, control segment size, and the item sizes. All of these parameters are based on probability distributions obtained from traffic traces. Finally, each of these conversations runs on top of its own TCP connection.

2.2 Trace Facility

Early in this effort it became clear that we needed good facilities to analyze the behavior of TCP. We therefore added code to the x-kernel to trace the relevant changes in the connection state. We paid particular attention to keeping the overhead of this tracing facility as low as possible, so as to minimize the effects on the behavior of the protocol. Specifically, the facility writes trace data to memory, dumps it to a file only when the test is over, and keeps the amount of data associated with each trace entry small (8 bytes).

We then developed various tools to analyze and display the tracing information. The rest of this section describes one such tool that graphically represents relevant features of the state of the TCP connection as a function of time. This tool outputs multiple graphs, each focusing on a specific set of characteristics of the connection state. Figure 1 gives an example. Since we use graphs like this throughout the paper, we now explain how to read the graph in some detail.

First, all TCP trace graphs have certain features in common, as illustrated in Figure 2. The circled numbers in this figure are keyed to the following explanations:

1. Hash marks on the x-axis indicate when an ACK was received.
2. Hash marks at the top of the graph indicate when a segment was sent.
3. The numbers on the top of the graph indicate when the n^{th} kilobyte (KB) was sent.
4. Diamonds on top of the graph indicate when the periodic coarse-grained timer fires. This does not imply a TCP timeout, just that TCP checked to see if any timeouts should happen.
5. Circles on top of the graph indicate that a coarse-grained timeout occurred, causing a segment to be

retransmitted.

6. Solid vertical lines running the whole height of the graph indicate when a segment that is eventually retransmitted was originally sent, presumably because it was lost.[2] Notice that several consecutive segments are retransmitted in the example.

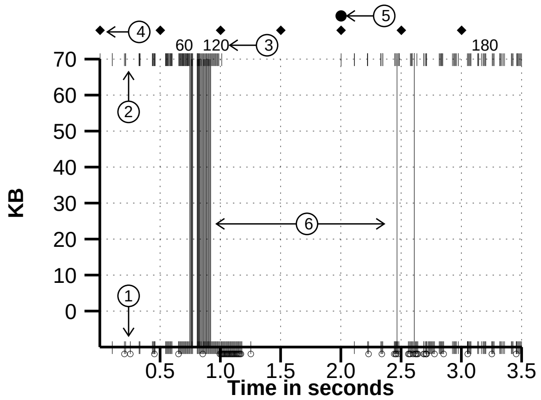

Figure 2: Common Elements in TCP Trace Graphs.

In addition to this common information, each graph depicts more specific information. The bottom graph in Figure 1 is the simplest—it shows the average sending rate, calculated from the last 12 segments. The top graph in Figure 1 is more complicated—it gives the size of the different windows TCP uses for flow and congestion control. Figure 3 shows these in more detail, again keyed by the following explanations:

1. The dashed line gives the threshold window. It is used during slow-start, and marks the point at which the congestion window growth changes from exponential to linear.
2. The dark gray line gives the send window. It is the minimum of the sender's buffer size and receiver's available buffer space.
3. The light gray line gives the congestion window. It is used for congestion control, and is an upper limit to the number of bytes sent but not yet acknowledged.
4. The thin line gives the actual number of bytes in transit at any given time, where by in transit we mean sent but not yet acknowledged.

The graphs just described are obtained from tracing information saved by the protocol, and are, thus, available whether the protocol is running in the simulator or over a real network. The simulator itself also reports certain information, such as the rate, in KB/s, at which data is entering or leaving a host or a router. For a router, the traces also save

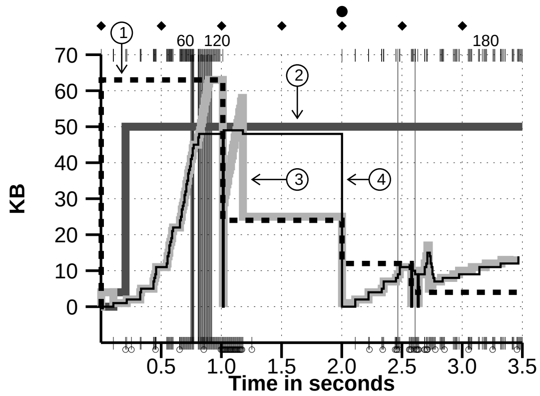

Figure 3: TCP Windows Graph.

the size of the queues as a function of time, and the time and size of segments that are dropped due to insufficient queue space.

3 Techniques

This section motivates and describes three techniques that Vegas employs to increase throughput and decrease losses. The first technique results in a more timely decision to retransmit a dropped segment. The second technique gives TCP the ability to anticipate congestion, and adjust its transmission rate accordingly. The final technique modifies TCP's slow-start mechanism so as to avoid packet losses while trying to find the available bandwidth. The relationship between our techniques and those proposed elsewhere are also discussed in this section.

3.1 New Retransmission Mechanism

In Reno, round trip time (RTT) and variance estimates are computed using a coarse-grained timer (around 500 ms), meaning that the RTT estimate is not very accurate. This coarse granularity influences both the accuracy of the calculation itself, and how often TCP checks to see if it should time out on a segment. For example, during a series of tests on the Internet, we found that for losses that resulted in a timeout—usually due to two or more dropped segments in a RTT—it took Reno an average of 1100ms from the time it sent a segment that was lost until it timed out and resent the segment, whereas less than 300ms would have been the correct timeout interval had a more accurate clock been used.

Reno not only retransmits when a coarse-grained timeout occurs, but also when it receives n duplicate ACKs (n is usually 3). Reno sends a duplicate ACK whenever it receives new data that it cannot acknowledge because it has

[2] For simplicity, we sometimes say a segment was lost, even though all we know for sure is that the sender retransmitted it.

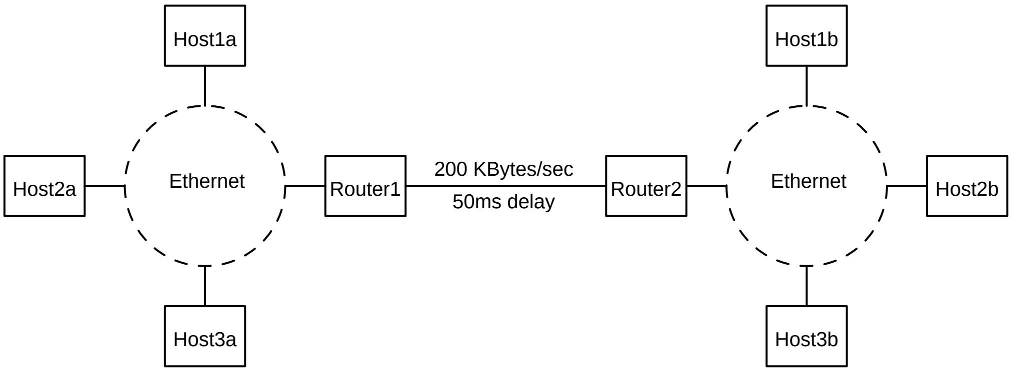

Figure 5: Network Configuration for Simulations.

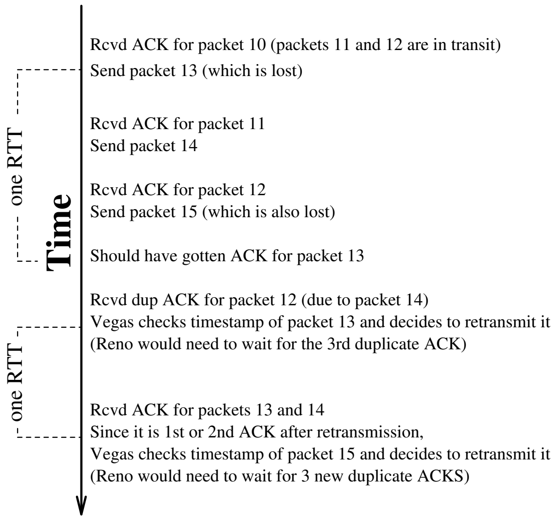

Figure 4: Example of Retransmit Mechanism.

not yet received all the previous data. For example, if Reno receives packet 2 but packet 3 is dropped, it will send a duplicate ACK for packet 2 when packet 4 arrives, again when packet 5 arrives, and so on. When the sender sees the third duplicate ACK for packet 2 (the one sent because the receiver had gotten packet 6) it retransmit packet 3.

Vegas extends Reno's retransmission mechanisms as follows. First, Vegas reads and records the system clock each time a segment is sent. When an ACK arrives, Vegas reads the clock again and does the RTT calculation using this time and the timestamp recorded for the relevant segment. Vegas then uses this more accurate RTT estimate to decide to retransmit in the following two situations (a simple example is given in Figure 4):

- When a duplicate ACK is received, Vegas checks to see if the difference between the current time and the timestamp recorded for the relevant segment is greater than the timeout value. If it is, then Vegas retransmits the segment without having to wait for n (3) duplicate ACKs. In many cases, losses are either so great or the window so small that the sender will never receive three duplicate ACKs, and therefore, Reno would have to rely on the coarse-grained timeout mentioned above.
- When a non-duplicate ACK is received, if it is the first or second one after a retransmission, Vegas again checks to see if the time interval since the segment was sent is larger than the timeout value. If it is, then Vegas retransmits the segment. This will catch any other segment that may have been lost previous to the retransmission without having to wait for a duplicate ACK.

In other words, Vegas treats the receipt of certain ACKs as a trigger to check if a timeout should happen. It still contains Reno's coarse-grained timeout code in case these mechanisms fail to recognize a lost segment.

Notice that the congestion window should only be reduced due to losses that happened at the current sending rate, and not due to losses that happened at an earlier, higher rate. In Reno, it is possible to decrease the congestion window more than once for losses that occurred during one RTT interval. In contrast, Vegas only decreases the congestion window if the retransmitted segment was previously sent *after* the last decrease. Any losses that happened before the last window decrease do not imply that the network is congested for the *current* congestion window size, and therefore, do not imply that it should be decreased again. This change is needed because Vegas detects losses much sooner than Reno.

3.2 Congestion Avoidance Mechanism

TCP Reno's congestion detection and control mechanism uses the loss of segments as a signal that there is congestion in the network. It has no mechanism to detect the incipient

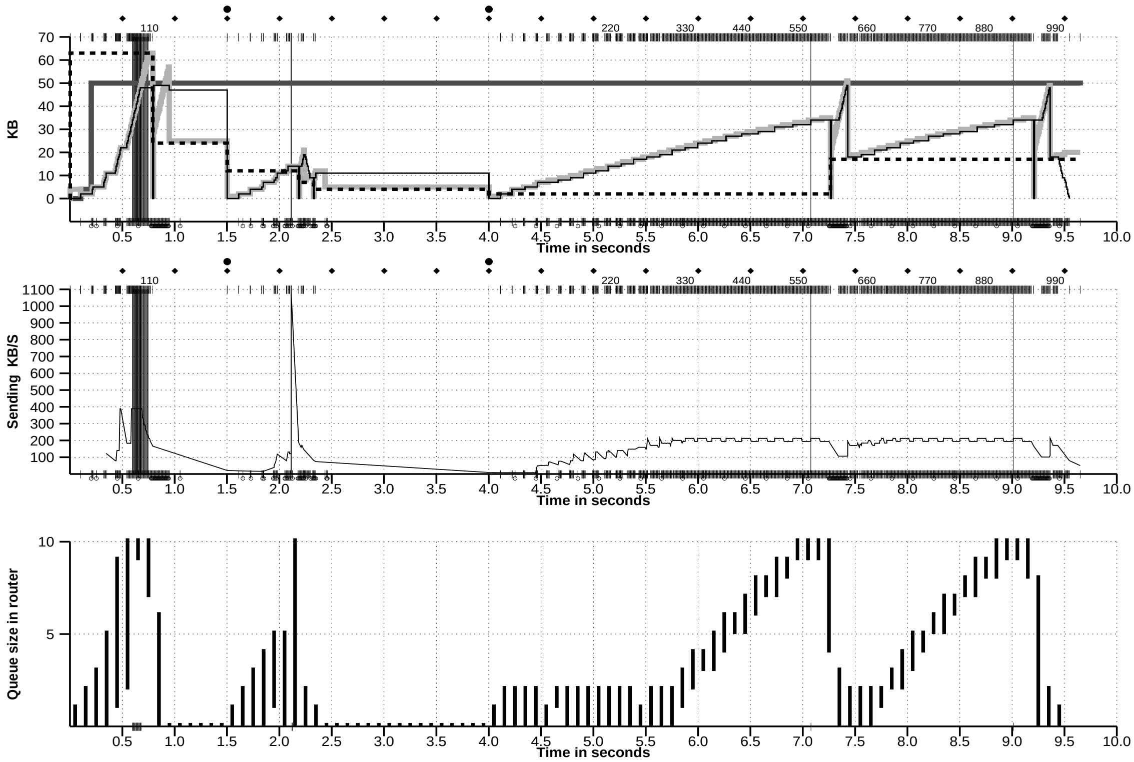

Figure 6: TCP Reno with No Other Traffic (Throughput: 105 KB/s).

stages of congestion—before losses occur—so they can be prevented. Reno is reactive, rather than proactive, in this respect. As a result, Reno *needs* to create losses to find the available bandwidth of the connection. This can be seen in Figure 6, which shows the trace of a Reno connection sending 1MB of data over the network configuration seen in Figure 5, with no other traffic sources; i.e., only Host1a sending to Host1b. In this case, the router queue size is ten and the queuing discipline is FIFO.

There are several previously proposed approaches for proactive congestion detection based on a common understanding of the network changes as it approaches congestion (an excellent development is given in [7]). These changes can be seen in Figure 6 in the time interval from 4.5 to 7.5 seconds. One change is the increased queue size in the intermediate nodes of the connection, resulting in an increase of the RTT for each successive segment. Wang and Crowcroft's DUAL algorithm [11] is based on this increase of the round-trip delay. The congestion window normally increases as in Reno, but every two round-trip delays the algorithm checks to see if the current RTT is greater than the average of the minimum and maximum RTTs seen so far. If it is, then the algorithm decreases the congestion window by one-eighth.

Jain's CARD (Congestion Avoidance using Round-trip Delay) approach [7] is based on an analytic derivation of a socially optimum window size for a deterministic network. The decision as to whether or not to change the current window size is based on changes to both the RTT and the window size. The window is adjusted once every two round-trip delays based on the product ($WindowSize_{current}$ - $WindowSize_{old}$) $\times$ ($RTT_{current}$ - RTT_{old}) as follows: if the result is positive, decrease the window size by one-eighth; if the result is negative or zero, increase the window size by one maximum segment size. Note that the window changes during every adjustment, that is, it oscillates around its optimal point.

Another change seen as the network approaches congestion is the flattening of the sending rate. Wang and Crowcroft's Tri-S scheme [10] takes advantage of this fact. Every RTT, they increase the window size by one segment and compare the throughput achieved to the throughput when the window was one segment smaller. If the difference is less than one-half the throughput achieved when only one segment was in transit—as was the case at the beginning of the connection—they decrease the window by one segment. Tri-S calculates the throughput by dividing the number of bytes outstanding in the network by the RTT.

Vegas' approach is most similar to Tri-S, in that it looks at changes in the throughput rate. However, it differs from Tri-S in that it calculates throughputs differently, and instead of looking for a change in the throughput slope, it compares

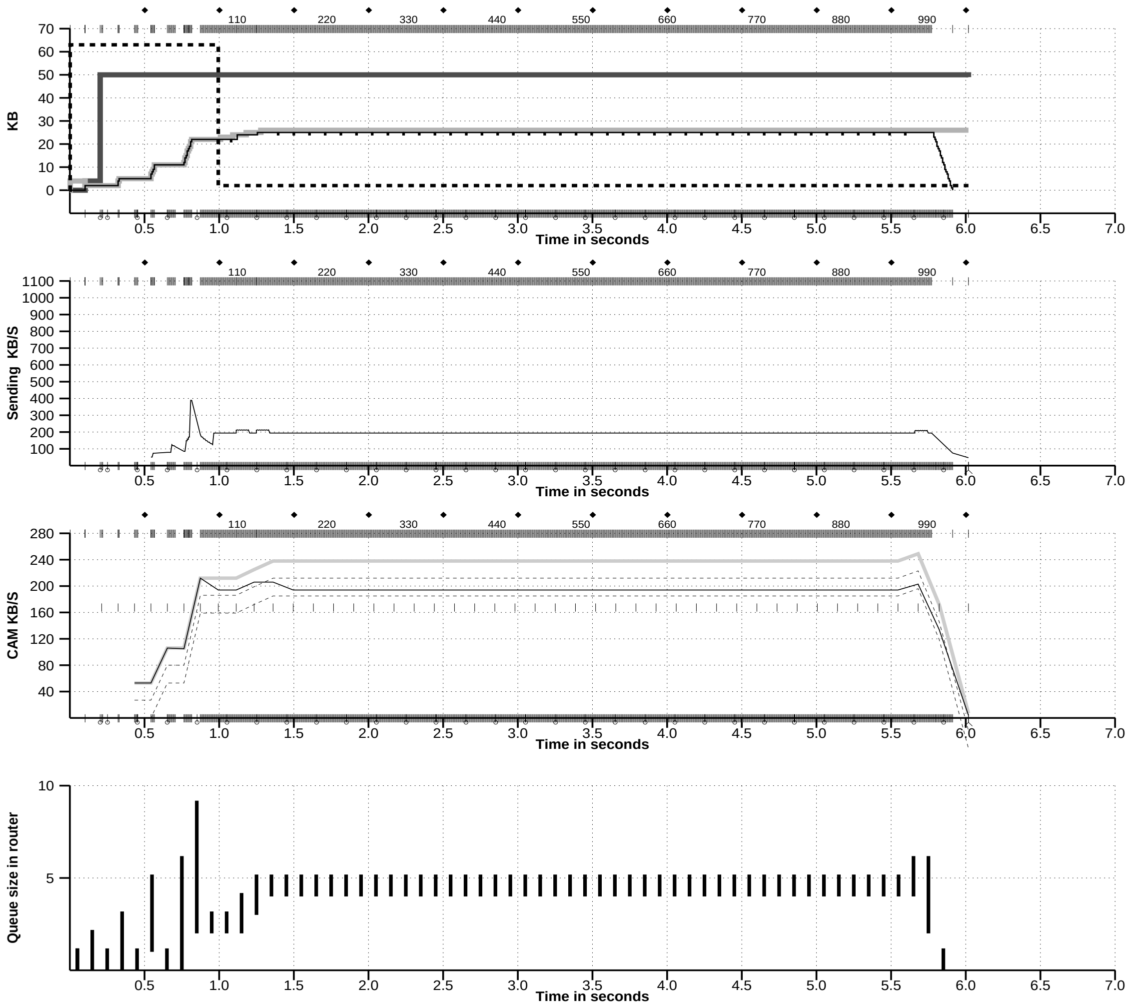

Figure 7: TCP Vegas with No Other Traffic (Throughput: 169 KB/s).

the measured throughput rate with the expected throughput rate. The simple idea that Vegas exploits is that the number of bytes in transit is directly proportional to the expected throughput, and therefore, as the window size increases—causing the bytes in transit to increase—the throughput of the connection should also increase.

Vegas uses this idea to measure and control the amount of *extra* data this connection has in transit, where by extra data we mean data that would not have been sent if the bandwidth used by the connection exactly matched the available bandwidth of the network. The goal of Vegas is to maintain the "right" amount of extra data in the network. Obviously, if a connection is sending too much extra data, it will cause congestion. Less obviously, if a connection is sending too little extra data, it cannot respond rapidly enough to transient increases in the available network bandwidth. Vegas' congestion avoidance actions are based on changes in the estimated amount of extra data in the network, and not only on dropped segments.

We now describe the algorithm in detail. Note that the algorithm is not in effect during slow-start. Vegas' behavior during slow-start is described in Section 3.3.

First, define a given connection's *BaseRTT* to be the RTT of a segment when the connection is not congested. In practice, Vegas sets *BaseRTT* to the minimum of all measured round trip times; it is commonly the RTT of the first segment sent by the connection, before the router queues increase due to traffic generated by this connection. If we assume that we are not overflowing the connection, then the expected throughput is given by:

$$Expected = WindowSize \,/\, BaseRTT$$

where *WindowSize* is the size of the current congestion window, which we assume for the purpose of this discussion, to be equal to the number of bytes in transit.

Second, Vegas calculates the current *Actual* sending rate. This is done by recording the sending time for a distinguished segment, recording how many bytes are transmitted between the time that segment is sent and its acknowledgement is received, computing the RTT for the distinguished segment when its acknowledgement arrives, and dividing the number of bytes transmitted by the sample RTT. This calculation is done once per round-trip time.[3]

Third, Vegas compares *Actual* to *Expected*, and adjusts the window accordingly. Let *Diff* = *Expected* - *Actual*. Note that *Diff* is positive or zero by definition, since *Actual* > *Expected* implies that we need to change *BaseRTT* to the latest sampled RTT. Also define two thresholds, $\alpha < \beta$, roughly corresponding to having too little and too much extra data in the network, respectively. When *Diff* $< \alpha$, Vegas increases the congestion window linearly during the next RTT, and when *Diff* $> \beta$, Vegas decreases the congestion window linearly during the next RTT. Vegas leaves the congestion window unchanged when $\alpha <$ *Diff* $< \beta$.

Intuitively, the farther away the actual throughput gets from the expected throughput, the more congestion there is in the network, which implies that the sending rate should be reduced. The β threshold triggers this decrease. On the other hand, when the actual throughput rate gets too close to the expected throughput, the connection is in danger of not utilizing the available bandwidth. The α threshold triggers this increase. The overall goal is to keep between α and β extra bytes in the network.

Because the algorithm, as just presented, compares the difference between the actual and expected throughput rates to the α and β thresholds, these two thresholds are defined in terms of KB/s. However, it is perhaps more accurate to think in terms of how many extra *buffers* the connection is occupying in the network. For example, on a connection with a *BaseRTT* of 100ms and a segment size of 1KB, if $\alpha = 30$KB/s and $\beta = 60$KB/s, then we can think of α as saying that the connection needs to be occupying at least three extra buffers in the network, and β saying it should occupy no more than six extra buffers in the network.

In practice, we express α and β in terms of buffers rather than extra bytes in transit. During linear increase/decrease mode—as opposed to the slow-start mode described below—we set α to two and β to four. This can be interpreted as an attempt to use at least two, but no more than four extra buffers in the connection.

Figure 7 shows the behavior of TCP Vegas when there is no other traffic present; this is the same condition that Reno ran under in Figure 6. There is one new type of graph in this figure, the third one, which depicts the congestion avoidance mechanism (CAM) used by Vegas. Once again, we use a detailed graph (Figure 8) keyed to the following explanation:

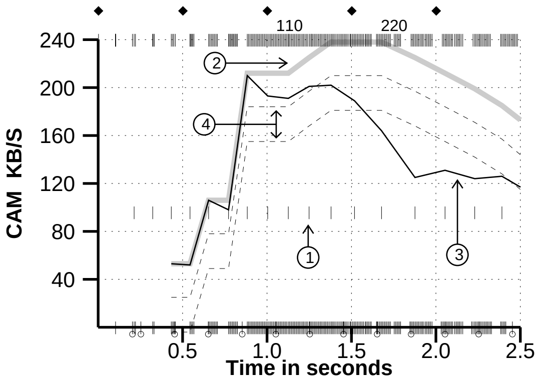

Figure 8: Congestion detection and avoidance in Vegas.

1. The small vertical line—once per RTT—shows the times when Vegas makes a congestion control decision; i.e., computes *Actual* and adjusts the window accordingly.
2. The gray line shows the *Expected* throughput. This is the throughput we should get if all the bytes in transit are able to get through the connection in one *BaseRTT*.
3. The solid line shows the *Actual* sending rate. We calculate it from the number of bytes we sent in the last RTT.
4. The dashed lines are the thresholds used to control the size of the congestion window. The top line corresponds to the α threshold and the bottom line corresponds to the β threshold.

Figure 9 shows a trace of Vegas sharing the bottleneck router with *tcplib* traffic. The bottom graph shows the output produced by the TRAFFIC protocol simulating the TCP traffic. The thin line is the sending rate in KB/s as seen in 100ms intervals; the thick line is a running average (size 3). The graph clearly shows Vegas' congestion avoidance mechanisms at work and how its throughput adapts to the changing conditions on the network.

3.3 Modified Slow-Start Mechanism

Reno's slow-start mechanism is very expensive in terms of losses when it is not limited by a small send buffer or a slow host. Since it doubles the size of the congestion window every RTT while there are no losses—which is equivalent to doubling the attempted throughput every RTT—when it finally overruns the connection's bandwidth, we can expect losses in the order of *half* the current congestion window, more if we encounter a burst from another connection.

[3] We have made every attempt to keep the overhead of Vegas' congestion avoidance mechanism as small as possible. To help quantify this effect, we ran both Reno and Vegas between SparcStations connected by an Ethernet, and measured the penalty to be less than 5%. This overhead can be expected to drop as processors become faster.

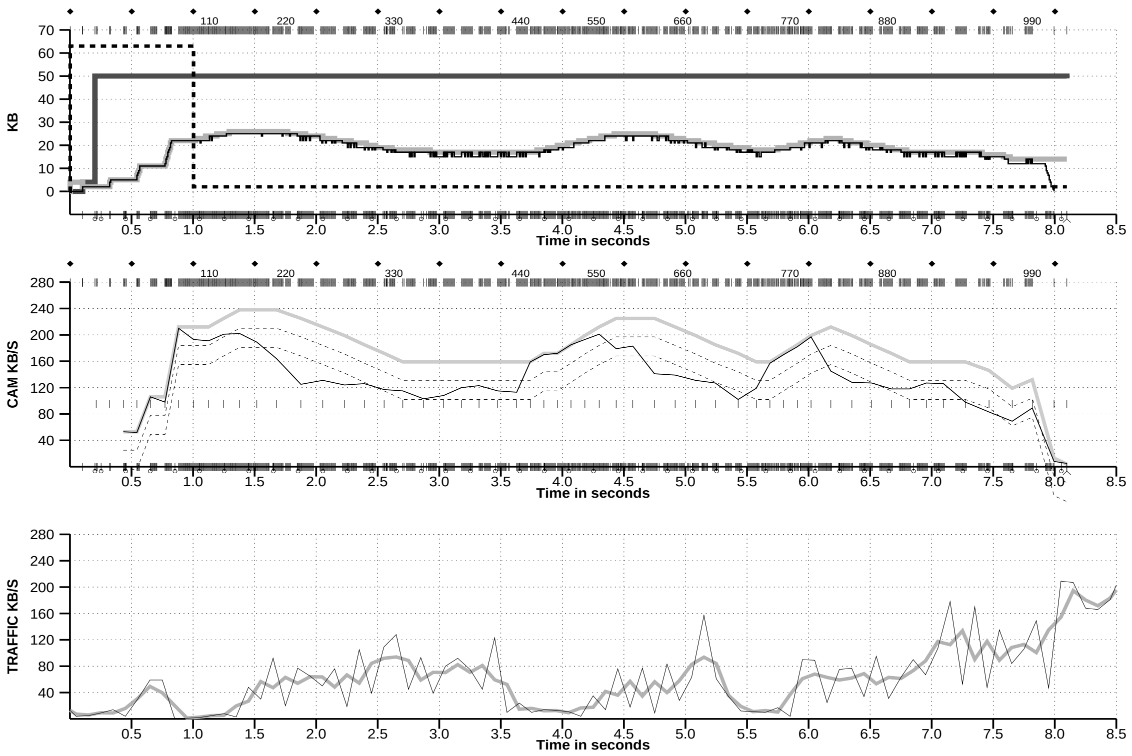

Figure 9: TCP Vegas with *tcplib*-Generated Background Traffic.

As network bandwidth increases, we can anticipate a similar increase in the expected losses of slow-start. We need a way to find a connection's available bandwidth that does not incur this kind of loss. Towards this end, we incorporated our congestion detection mechanism into slow-start with only minor modifications. To be able to detect and avoid congestion during slow-start, Vegas allows exponential growth only every other RTT. In between, the congestion window stays fixed so a valid comparison of the expected and actual rates can be made. When the actual rate falls below the expected rate by a certain amount—we call this the γ threshold—Vegas changes from slow-start mode to linear increase/decrease mode.

The behavior of the modified slow-start can be seen in Figures 7 and 9. The reason that we need to measure the actual rate with a fixed congestion window is that we want the actual rate to represent the bandwidth allowed by the connection. Thus, we can only send as much data as is acknowledged in the ACK. (During slow-start, Reno sends two segments for each ACK received.)

In summary, we believe that adding congestion detection to slow-start is important, and will become more important as network bandwidth increases. Vegas offers a beginning, but there is a problem that still needs to be addressed. During slow-start, TCP sends segments at twice the rate supported by the connection; i.e., two segments are sent for every ACK received. If there aren't enough buffers in the bottleneck router, Vegas' slow-start with congestion detection may lose segments before getting any feedback that tells it to slow down. We have several solutions in mind that we plan to implement in the near future. One is to use rate control during slow-start, using a rate defined by the current window size and the *BaseRTT*. Another is to slow down as we reach the bandwidth available to the connection (Vegas has enough information to figure this out).

4 Simulation Results

This section reports the results of simulated runs of TCP Vegas and Reno using the x-kernel based simulator. Although we have also measured Vegas and Reno on the actual Internet—see Section 5—the simulator allows us to better control the experiment, and in particular, gives us the opportunity to see whether or not Vegas gets its performance at the expense of Reno-based connections. Note that all the experiments used in this section are on the network configuration shown in Figure 5.

	Reno/Reno	Reno/Vegas	Vegas/Reno	Vegas/Vegas
Throughput (KB/s)	60/109	61/123	66/119	74/131
Throughput Ratios	1.00/1.00	1.02/1.13	1.10/1.09	1.23/1.20
Retransmissions (KB)	30/22	43/1.8	1.5/18	0.3/0.1
Retransmit Ratios	1.00/1.00	1.43/0.08	0.05/0.82	0.01/0.01

Table 1: One-on-One (300KB and 1MB) Transfers.

	Reno	Vegas-1,3	Vegas-2,4
Throughput (KB/s)	58.30	89.40	91.80
Throughput Ratio	1.00	1.53	1.58
Retransmissions (KB)	55.40	27.10	29.40
Retransmit Ratio	1.00	0.49	0.53
Coarse Timeouts	5.60	0.90	0.90

Table 2: 1MByte Transfer with *tcplib*-Generated Background Reno Traffic.

4.1 One-on-One Experiments

We begin by studying how two TCP connections interfere with each other. To do this, we start a 1MB transfer, and then after a variable delay, start a 300KB transfer. The transfer sizes and delays are chosen to ensure that the smaller transfer is contained completely within the larger.

Table 1 gives the results for the four possible combinations, where the column heading Reno/Vegas denotes a 300KB transfer using Reno contained within a 1MByte transfer using Vegas. For each combination, the table gives the measured throughput and number of kilobytes retransmitted for both transfers; e.g., in the the case of Reno/Vegas, the 300KB Reno transfer achieved a 61KB/s throughput rate and the 1MByte Vegas transfer achieved a 123KB/s throughput rate.[4] The ratios for both throughput rate and kilobytes retransmitted are relative to the Reno/Reno column. The values in the table are averages from 12 runs, using 15 and 20 buffers in the routers, and with the delay before starting the smaller transfer ranging between 0 and 2.5 seconds.

The main thing to take away from these numbers is that Vegas does not adversely affect Reno's throughput. Reno's throughput stays pretty much unchanged when it is competing with Vegas rather than itself—the ratios for Reno are 1.02 and 1.09 for Reno/Vegas and Vegas/Reno, respectively. Also, when Reno competes with Vegas rather than itself, the combined number of kilobytes retransmitted for the pair of competing connections drops significantly. The combined Reno/Reno retransmits are 52KB compared with 45KB for Reno/Vegas and 19KB for Vegas/Reno. Finally, note that the combined Vegas/Vegas retransmits are less than 1KB on the average—an indication that the congestion avoidance mechanism is working.

[4] Comparing the small transfer to the large transfer in any given column is not meaningful. This is because the large transfer was able to run by itself during most of the test.

4.2 Background Traffic

We next measured the performance of a distinguished TCP connection when the network is loaded with traffic generated from *tcplib*. That is, the protocol TRAFFIC is running between Host1a and Host1b in Figure 5, and a 1MByte transfer is running between Host2a and Host2b. In this set of experiments, the *tcplib* traffic is running over Reno.

Table 2 gives the results for Reno and two versions of Vegas—Vegas-1,3 uses $\alpha = 1$ and $\beta = 3$, and Vegas-2,4 uses $\alpha = 2$ and $\beta = 4$. We varied these two thresholds to study the sensitivity of our algorithm to them. The numbers shown are averages from 57 runs, obtained by using different seeds for *tcplib*, and by using 10, 15 and 20 buffers in the routers.

The table shows the throughput rate for each of the distinguished connections using the three protocols, along with their ratio to Reno's throughput. It also gives the number of kilobytes retransmitted, the ratio of retransmits to Reno's, and the average number of coarse-grained timeouts per transfer. For example, Vegas-1,3 had 53% better throughput than Reno, with only 49% of the losses. Also note that there is little difference between Vegas-1,3 and Vegas-2,4.

These simulations tell us the expected improvement of Vegas over Reno: more than 50% improvement on throughput, and only half the losses. The results from the one-on-one experiments indicate that the gains of Vegas are not made at the expense of Reno; this belief is further supported by the fact that the background traffic's throughput increases by 20% when Reno is competing for resources with Vegas, as compared to when Reno is competing with itself.

We also ran these tests with the background traffic using Vegas rather than Reno. This simulates the situation where the whole world uses Vegas. The throughput and the kilobytes retransmitted by the 1MByte transfers didn't change

	1MB Transfer	
traffic over	Reno	Vegas
Reno (KB/s)	68	82
Vegas (KB/s)	84	85

Table 3: Throughput of Background Traffic When Competing with a 1MB Transfer.

significantly—less than 4%—but this time the throughput of the background traffic was unaffected by the type of protocol running the 1MB transfer. These results are summarized in Table 3.

4.3 Other Experiments

We tried many variations of the previous experiments. On the whole, the results were similar, except for when we changed TCP's send-buffer size. Below we summarize these experiments and their results.

- *One-on-one tests with traffic in the background.* The results were similar. Again, Reno did better when running against Vegas than against itself, but this time, its losses increased by only 6% (versus 43%) in the Reno/Vegas case.

- *Two-way background traffic.* There have been reports of change in TCP's behavior when the background traffic is two-way rather than one-way [12]. Thus, we modified the experiment in Section 4.2 by adding *tcplib* traffic from Host3b to Host3a. The throughput ratio stayed the same, but the loss ratio was much better: 0.29. Reno resent more data and Vegas remained about the same. The fact that there wasn't much change is probably due to the fact that *tcplib* already creates some 2-way traffic—TELNET connections send one byte and get one or more bytes back, and FTP connections send and get control packets before doing a transfer.

- *Different TCP send-buffer sizes.* For all the experiments reported so far, we ran TCP with a 50KB send-buffer. For this experiment, we tried send-buffer sizes between 50KB and 5KB. Vegas' throughput and losses stayed unchanged between 50KB and 20KB; from that point on, as the buffer decreased, so did the throughput. This was due to the protocol not being able to keep the pipe full.

  Reno's throughput initially *increased* as the buffers got smaller, and then it decreased. It always remained under the throughput measured for Vegas. We have previously seen this type of behavior while running Reno on the Internet. If we look back at Figure 6, we see that as Reno increases its congestion window, it uses more and more buffers in the router until it loses packets by overrunning the queue. If we limit the congestion window by reducing the size of the send-buffer, we may prevent it from overrunning the router's queue.

- *Multiple Competing Connections.* We ran simulations with 2, 4, and 16 connections sharing a bottleneck link, where all the connections either had the same propagation delay, or where one half of the connections had twice the propagation delay of the other half. Many different propagation delays were used, with the appropiate results averaged. These simulations where used to obtain preliminary results regarding fairness and the behavior of Vegas under stress.

  To judge fairness, we chose Jain's *fairness index* [8]. In the case of 2 and 4 connections, with each connection transfering 8 MB, Reno was slightly more fair than Vegas when all connections had the same propagation delay, but Vegas was more fair than Reno when the propagation delay was larger for half of the connections. In the experiments with 16 connections, with each connection transfering 2 MB, Vegas was more fair than Reno in all experiments. Overall, Vegas is at least as fair as Reno.

  There were no stability problems in the case of 16 connections sharing the bottleneck link, even though there were only 20 buffers at the router. Although Vegas' congestion avoidance mechanisms could not work effectively due to the limited number of buffers, Vegas only had half the coarse-grained timeouts as Reno due to Vegas' improved retransmit mechanism.

5 Internet Results

This section discusses measurements of TCP over the Internet. Because it is simple to move a protocol between the simulator and the "real world", the numbers reported in this section are for exactly the same code as in the previous section. Our results must be considered preliminary because they were limited to transfers between the University of Arizona (UA) and the National Institutes of Health (NIH). The connection consists of 17 hops, and passes through Denver, St. Louis, Chicago, Cleveland, New York and Washington DC.

The results are derived from a set of runs over a seven day period from January 23-29. Each run consists of a

	Reno	Vegas-1,3	Vegas-2,4
Throughput (KB/s)	53.00	72.50	75.30
Throughput Ratio	1.00	1.37	1.42
Retransmissions (KB)	47.80	24.50	29.30
Retransmit Ratio	1.00	0.51	0.61
Coarse Timeouts	3.30	0.80	0.90

Table 4: 1MByte transfer over the Internet.

	1024KB		512KB		128KB	
	Reno	Vegas	Reno	Vegas	Reno	Vegas
Throughput (KB/s)	53.00	72.50	52.00	72.00	31.10	53.10
Throughput Ratio	1.00	1.37	1.00	1.38	1.00	1.71
Retransmissions (KB)	47.80	24.50	27.90	10.50	22.90	4.00
Retransmit Ratio	1.00	0.51	1.00	0.38	1.00	0.17
Coarse Timeouts	3.30	0.80	1.70	0.20	1.10	0.20

Table 5: Effects of transfer size over the Internet.

set of seven transfers from UA to NIH—Reno sends 1MB, 512KB, and 128KB, Vegas-1,3 sends 1MB, 512KB, and 128KB, and Vegas-2,4 sends 1MB. We inserted a 45 second delay between each transfer in a run to give the network a chance to settle down, a run started approximately once every hour, and we shuffled the order of the transfers within each run.

Table 4 shows the results for the 1MB transfers. Depending on the congestion avoidance thresholds, it shows between 37% and 42% improvement over Reno's throughput with only 51% to 61% of the retransmissions. Note that Vegas out-performed Reno on 92% of the transfers, and across all levels of congestion; i.e., during both the middle of the night and during periods of high load.

Because we were concerned that Vegas' throughput improvement depended on large transfer sizes, we also varied the size of the transfer. Table 5 shows the effect of transfer size on both throughput and retransmissions for Reno and Vegas-1,3. First, observe that Vegas does better relative to Reno as the transfer size decreases. In terms of throughput, we see an increase from 37% to 71%. The results are similar for retransmissions, as the relative number of Vegas retransmissions goes from 51% of Reno's to 17% of Reno's.

Next, notice that the number of kilobytes retransmitted by Reno starts to flatten out as the transfer size decreases. When we decreased the transfer size by half, from 1MB to 512KB, we see only a 42% decrease in the number of kilobytes retransmitted. When we further decrease the transfer size to one-fourth its previous value, from 512KB to 128KB, the number of kilobytes retransmitted only decreases by 18%. This indicates that we are approaching the average number of kilobytes retransmitted due to Reno's slow-start losses. From these results, we conclude that there are around 20KBs retransmitted during slow-start, for the conditions of our experiment.

On the other hand, the number of kilobytes retransmitted by Vegas decreases almost linearly with respect to the transfer size. This indicates that Vegas eliminates nearly all losses during slow-start due to its modified slow-start with congestion avoidance.

6 Discussion

Most of the experiments reported in the previous two sections show the benefits of running a Vegas connection when most of the traffic is from Reno connections. An equally interesting question is what happens when the whole world runs Vegas. Simulations show that if there are enough buffers in the routers—meaning that Vegas' congestion avoidance mechanisms can function effectively—a higher throughput and a faster respone time result. For example, simulations running *tcplib* traffic over both Reno and Vegas show that the average response time in TELNET connections is around 25% faster when using Vegas as compared to Reno.

As the load increases and/or the number of router buffers decreases, Vegas's congestion avoidance mechanisms are not as effective, and Vegas starts to behave more like Reno. Under heavy congestion, Vegas behaves very similarly to Reno, since Vegas "falls back" to Reno's course-grained timeout mechanism.

The important point to keep in mind is that up to the point that congestion is bad enough for Vegas' behavior to degenerate into Reno, Vegas is *less* aggressive in its use of router buffers than Reno. This is because Vegas limits its use of router buffers as specified by the β threshold, whereas Reno increases its window size until there are losses—which means all the router buffers are being used.

Selective ACKs [5, 6] have been proposed as a way to decrease the number of unnecessarily retransmitted packets and to provide information for a better retransmit mechanism than the one in Reno. Although the selective ACK mechanism is not yet well defined, we make the following observations about how it compares to Vegas. First, it only relates to Vegas' retransmission mechanism; selective ACKs by themselves affect neither the congestion nor the the slow-start mechanisms. Second, there is little reason to believe that selective ACKs can significantly improve on Vegas in terms of unnecessary retransmissions, as there were only 6KB per MB unnecessarily retransmitted by Vegas in our Internet experiments. Third, selective ACKs have the potential to retransmit lost data sooner on future networks with large delay/bandwidth products. It would be interesting to see how Vegas and the selective ACK mechanism work in tandem on such networks. Finally, we note that selective ACKs require a change to the TCP standard, whereas the Vegas modifications are an implementation change that is isolated to the sender.

Vegas' congestion detection algorithm depends on an accurate value for *BaseRTT*. If our estimate for the *BaseRTT* is too small, then the protocol's throughput will stay below the available bandwidth; if it is too large, then it will overrun the connection. Our experience is that the protocol does well with its current choice of *BaseRTT*. However, we plan to study this more carefully in the future.

7 Conclusions

We have introduced several techniques for improving TCP, including a new timeout mechanism, a novel approach to congestion avoidance that tries to control the number of extra buffers the connection occupies in the network, and a modified slow-start mechanism. Experiments on both the Internet and using a simulator show that Vegas achieves 40 to 70% better throughput, with one-fifth to one-half as many bytes being retransmitted, as compared to the implementation of TCP in the Reno distribution of BSD Unix.

In many respects, this work is still preliminary. First, we need to test Vegas under a wider set of conditions, and in particular, a more comprehensive fairness study needs to be done. Second, we believe that more attention needs to be paid to avoiding congestion during slow-start, and as pointed out in Section 3.3, we are currently experimenting with some promising strategies.

Acknowledgements

Thanks to Lew Berman from the National Library of Medicine for providing a machine on the East Coast that we could use in our experiments.

References

[1] P. Danzig and S. Jamin. tcplib: A Library of TCP Internetwork Traffic Characteristics. Technical Report CS-SYS-91-495, Computer Science Department, USC, 1991.

[2] A. Heybey. The network simulator. Technical report, MIT, Sept. 1990.

[3] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[4] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of the SIGCOMM '88 Symposium*, pages 314–32, Aug. 1988.

[5] V. Jacobson and R. Braden. TCP Extensions for Long-Delay Paths. Request for Comments 1072, Oct. 1988.

[6] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. Request for Comments 1323, May 1992.

[7] R. Jain. A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. *ACM Computer Communication Review*, 19(5):56–71, Oct. 1989.

[8] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for ExperimentalDesign, Measurement, Simulation and Modeling*. John Wiley and Sons, Inc., New York, 1991.

[9] S. Keshav. REAL: A Network Simulator. Technical Report 88/472, Department of Computer Science, UC Berkeley, 1988.

[10] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). *ACM Computer Communication Review*, 21(1):32–43, Jan. 1991.

[11] Z. Wang and J. Crowcroft. Eliminating Periodic Packet Losses in 4.3-Tahoe BSD TCP Congestion Control Algorithm. *ACM Computer Communication Review*, 22(2):9–16, Apr. 1992.

[12] L. Zhang, S. Shenker, and D. D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proceedings of the SIGCOMM '91 Symposium*, pages 133–147, Sept. 1991.