TCP Vegas: End to End Congestion Avoidance on a Global Internet*

Lawrence S. Brakmo and Larry L. Peterson†

Abstract

Vegas is an implementation of TCP that achieves between 37 and 71% better throughput on the Internet, 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.

Keywords

TCP, Reno, Vegas, protocols, congestion avoidance.

1 Introduction

Few would argue that one of TCP's strengths lies in its adaptive retransmission and congestion control mechanism, with Jacobson's paper [7] 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 an implementation of TCP, based on modifications to the Reno implementation of TCP, that we refer to as TCP *Vegas*. This name is a takeoff 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.

The main result reported in this paper is that Vegas is able to achieve between 37 and 71% 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. Section 4 then presents a comprehensive evaluation of Vegas' performance, including both simulation results and measurements of TCP running over the Internet. Finally, Section 5 discusses several relevant issues and Section 6 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 [6]. Our implementation of Reno was derived by retrofitting the BSD implementation into the x-kernel. Our implementation of Vegas was derived by modifying Reno.

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

†The authors are with the Department of Computer Science, University of Arizona, Tucson, AZ 85721. (email: brakmo@cs.arizona.edu llp@cs.arizona.edu

[1] We limit our discussion to Reno, which is both newer and better performing than Tahoe. Section 5.4 discusses our results relative to newer versions of TCP—Berkeley Network Release 2 (BNR2) and BSD 4.4.

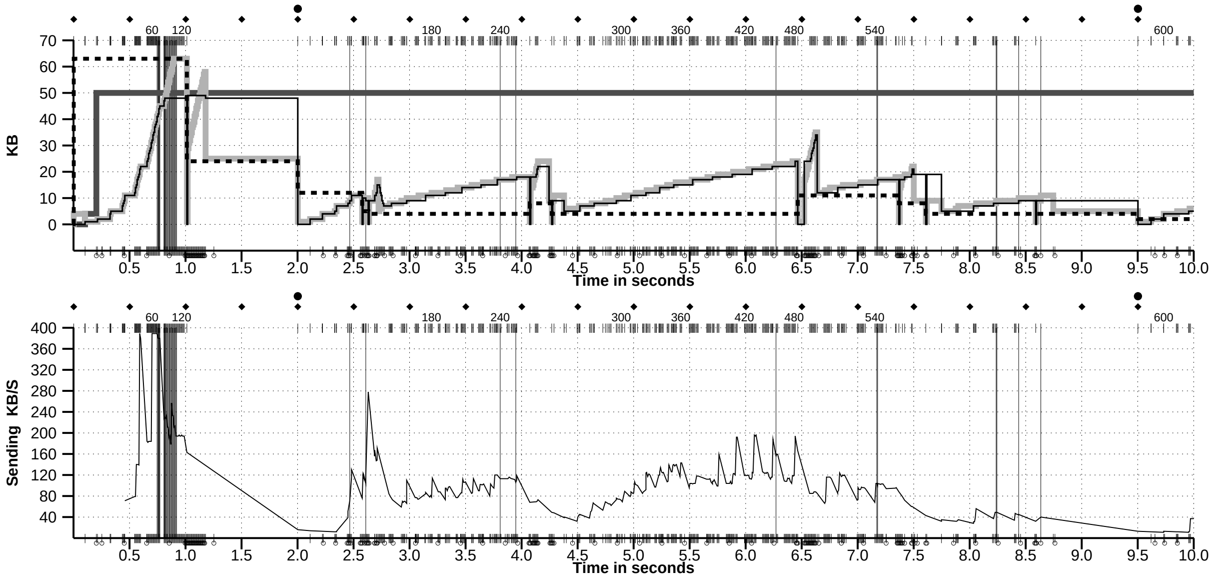

Figure 1: TCP Reno trace examples.

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 [12] and Netsim [5]—we decided to build our own simulator based on the x-kernel. 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* [3]. 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. Fig. 1

gives an example. Since we use graphs like this throughout the paper, we now explain how to read the graph in some detail.

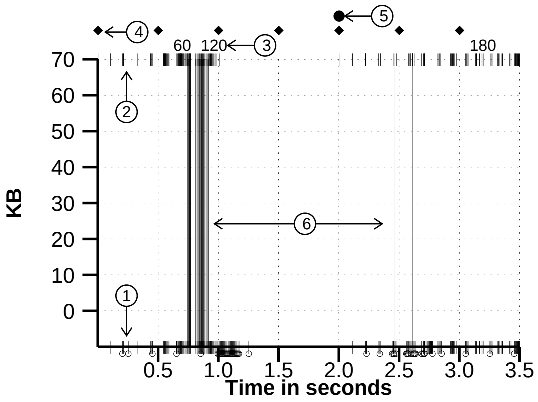

Figure 2: Common elements in TCP trace graphs.

First, all TCP trace graphs have certain features in common, as illustrated in Fig. 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.

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

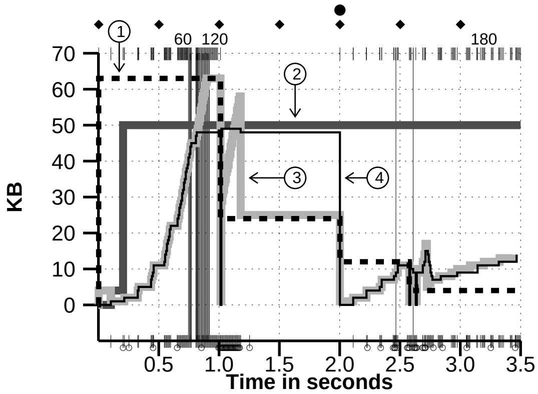

Figure 3: TCP windows graph.

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 advertized window, and defines an upper limit to the number of bytes sent but not yet acknowledged.
3. The light gray line gives the congestion window. It is used for congestion control, and is also 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.

Since the window graph presents a lot of information, it is easy to get lost in the detail. To assist the reader in developing a better understanding of this graph, the Appendix presents a detailed description of the behavior depicted in Fig. 3.

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 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.

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

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 during the initial use of slow-start. The relationship between our techniques and those proposed elsewhere are also discussed in this section in the appropriate subsections.

3.1 New Retransmission Mechanism

Reno uses two mechanisms to detect and then retransmit lost segments. The original mechanism, which is part of the TCP specification, is the retransmit timeout. It is based on round trip time (RTT) and variance estimates computed by sampling the time between when a segment is sent and an ACK arrives. In BSD-based implementations, the clock used to time the round-trip "ticks" every 500ms. Checks for timeouts also occur only when this coarse-grain clock ticks. The coarseness inherent in this mechanism implies that the time interval between sending a segment that is lost until there is a timeout and the segment is resent is generally much longer than necessary. For example, during a series of tests on the Internet, we found that for losses that resulted in a timeout 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.

This unnecessarily large delay did not go unnoticed, and the Fast Retransmit and Fast Recovery mechanisms where incorporated into the Reno implementation of TCP (for a more detailed description see [15]). Reno not only retransmits when a coarse-grain timeout occurs, but also when it receives n duplicate ACKs (n is usually 3). Reno sends a duplicate ACK whenever it receives a new segment that it cannot acknowledge because it has not yet received all the previous segments. For example, if Reno receives segment 2 but segment 3 is dropped, it will send a duplicate ACK for segment 2 when segment 4 arrives, again when segment 5 arrives, and so on. When the sender sees the third duplicate ACK for segment 2 (the one sent because the receiver had gotten segment 6) it retransmits segment 3.

The Fast Retransmit and Fast Recovery mechanisms are very successful—they prevent more than half of the coarse-grain timeouts that occur on TCP implementations without these mechanisms. However, some of our early analysis indicated that eliminating the dependency on coarse-grain timeouts would result in at least a 19% increase in throughput.

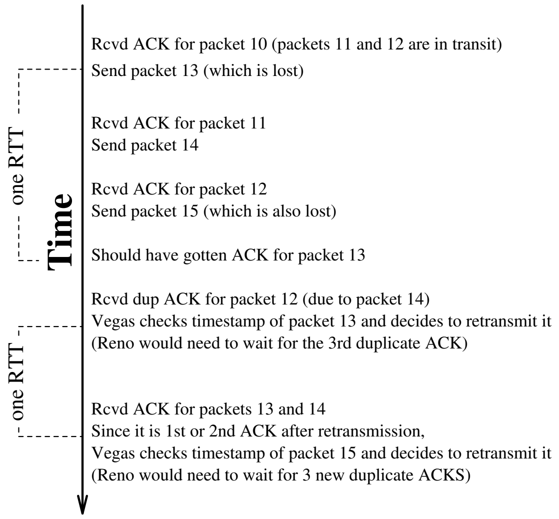

Figure 4: Example of retransmit mechanism.

Vegas, therefore, 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 Fig. 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-grain timeout mentioned above.

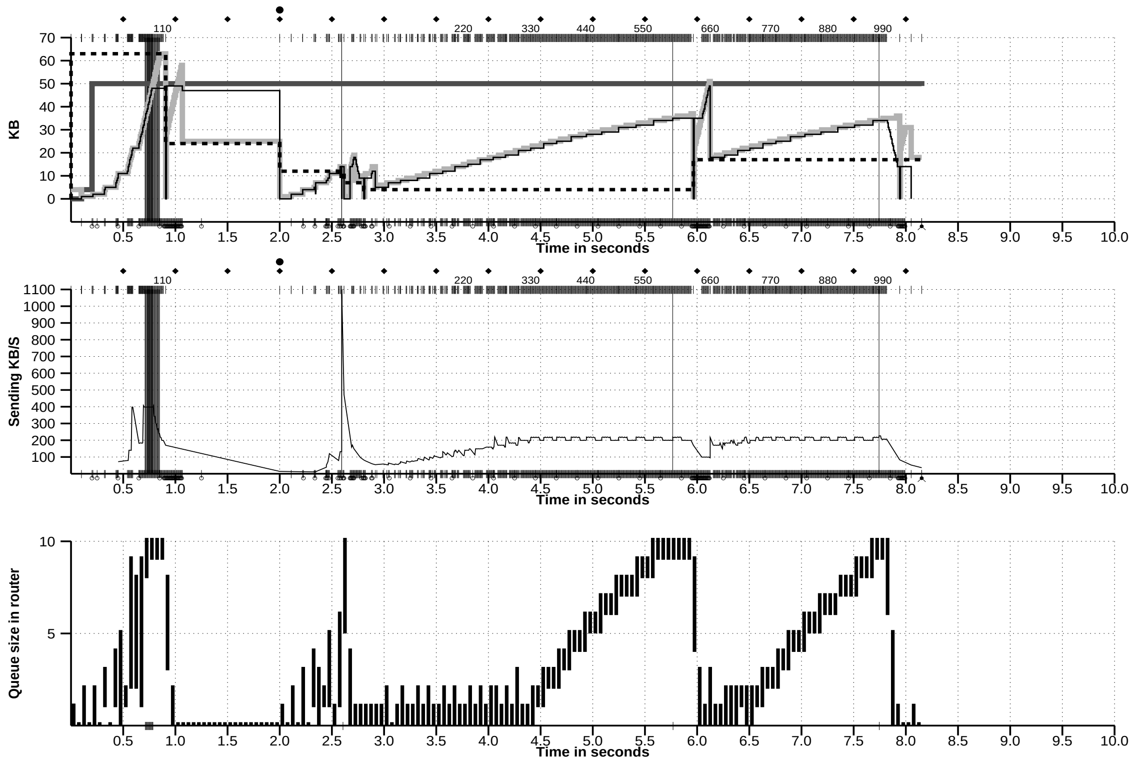

Figure 5: TCP Reno with no other traffic (throughput: 123 KB/s).

- 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 hint to check if a timeout should occur. Since it only checks for timeouts in rare occasions, the overhead is small. Notice that even though one could reduce the number of duplicate ACKs used to trigger the Fast Retransmit from 3 duplicate ACKs to either 2 or 1, it is not recommended as it could result in many unnecessary retransmissions and because it makes assumptions about the likelihood that packets will be delivered out of order.

The goal of the new retransmission mechanism is not just to reduce the time to detect lost packets from the third duplicate ACK to the first or second duplicate ACK—a small savings—but to detect lost packets even though there may be no second or third duplicate ACK. The new mechanism is very successful at achieving this goal, as it further reduces the number of coarse-grained timeouts in Reno by more than half.[3] Vegas still contains Reno's coarse-grain timeout code in case the new mechanisms fail to recognize a lost segment.

Related to making timeouts more timely, 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.[4] 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

[3] This was tested on an implementation of Vegas which did not have the congestion avoidance and slow-start modification described later in this section.

[4] This problem in the BSD versions of Reno has also been pointed out by Sally Floyd[4].

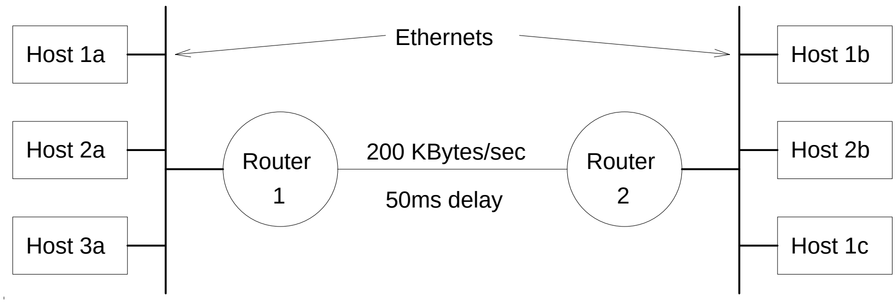

Figure 6: Network configuration for simulations.

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 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 Fig. 5, which shows the trace of a Reno connection sending 1MB of data over the network configuration seen in Fig. 6, with no other traffic sources; i.e., only Host1a sending to Host1b. In this case, the router queue size is ten—each packet is 1.4KB— and the queuing discipline is FIFO.

As seen in Fig. 5, Reno's mechanism to detect the available bandwidth is to continually increase its window size, using up buffers along the connection's path, until it congests the network and segments are lost. It then detects these losses and decreases its window size. Consequently, Reno is continually congesting the network and creating its own losses.[5] These losses may not be expensive if the Fast Retransmit and Fast Recovery mechanisms catch them, as seen with the losses around 7 and 9 seconds, but by unnecessarily using up buffers at the bottleneck router it is creating losses for other connections sharing this router.

As an aside, it is possible to set up the experiment in such a way that there are little or no losses. This is done by limiting the maximum window size such that it never exceeds the delay-bandwidth product of the connection plus the number of buffers at the bottleneck. This was done, for example, in [7]. However, this only works when one knows both the available bandwidth and the number of available buffers at the bottleneck. Given that one doesn't have this information under real conditions, we consider such experiments to be somewhat unrealistic.

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 [10]). These changes can be seen in Fig. 5 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 [17] is based on reacting to 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 [10] 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

[5] It is possible to set the experiment in such a way that there are little or no losses. This is done by limiting the maximum window size such that it never exceeds the delay-bandwidth product of the connection plus the number of buffers at the bottleneck. However, this only works when one knows both the available bandwidth and the number of available buffers at the bottleneck. Given that one doesn't have this information under real conditions, we consider such experiments to be somewhat unrealistic.

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 [16] 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, or more specifically, changes in the sending 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 the measured throughput rate with an expected throughput rate. The basis for this idea can be seen in Fig. 5 in the region between 4 and 10 seconds. As the window size increases we expect the throughput (or sending rate) to also increase. But the throughput cannot increase beyond the available bandwidth; beyond this point, any increase in the window size only results in the segments taking up buffer space at the bottleneck router.

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.[6] If we assume that we are not overflowing the connection, then the expected throughput is given by:

$$\mathit{Expected} = \mathit{WindowSize} \,/\, \mathit{BaseRTT}$$

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.[7]

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

[6] Although we don't know the exact value for the BaseRTT, our experience suggests our algorithm is not sensitive to errors in the BaseRTT.

[7] 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.

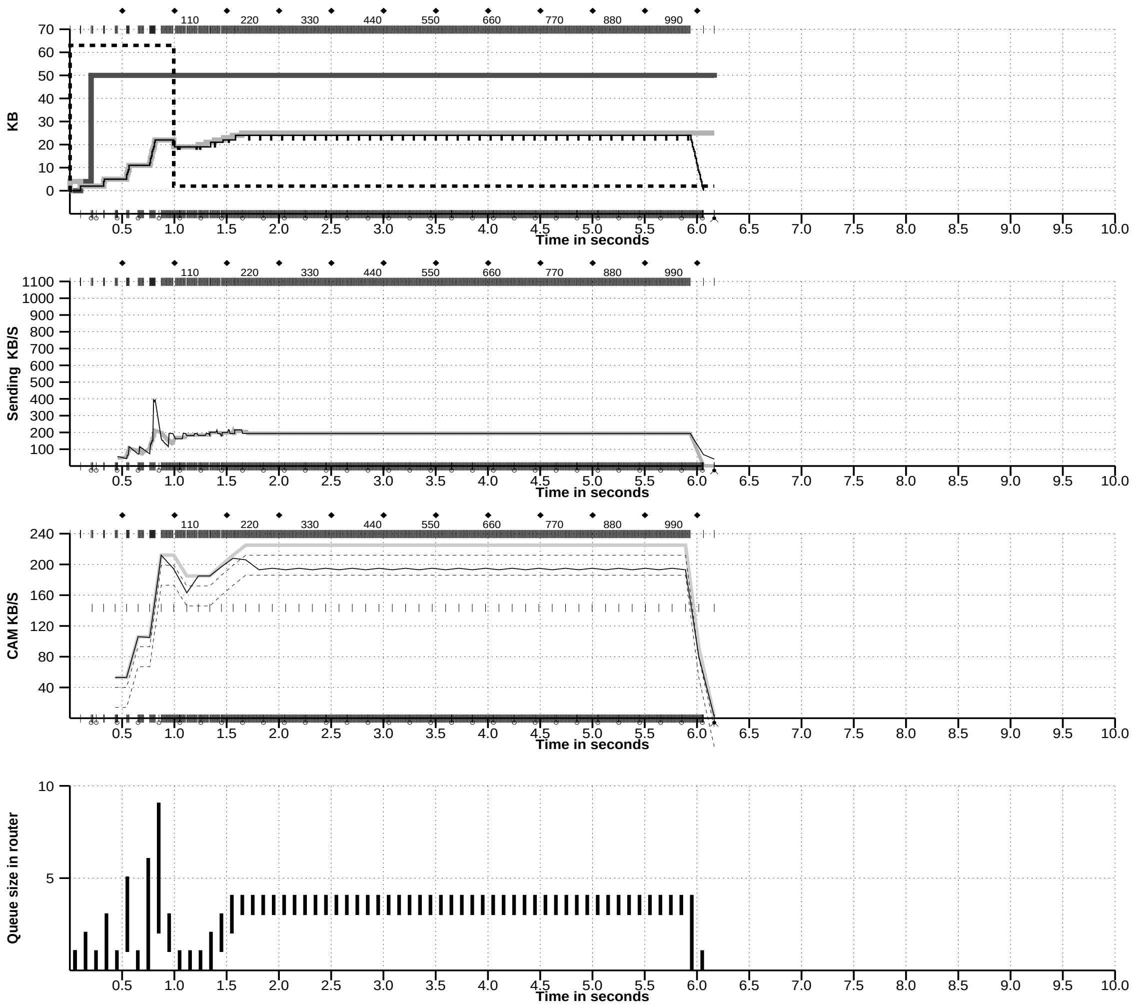

Figure 7: TCP Vegas with no other traffic (throughput: 169 KB/s).

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 one and β to three. This can be interpreted as an attempt to use at least one, but no more than three extra buffers in the connection. We settled on these values for α and β as they are the smallest feasible values. We want α to be greater than zero so the connection is using at least one buffer at the bottleneck router. Then, when the aggregate traffic from the other connections decreases (as is bound to happen every so often), our connection can take advantage of the extra available bandwidth immediately without having to wait for the one RTT delay necessary for the linear increase to occur. We want β to be two buffers greater than α so small sporadic changes in the available bandwidth will not create oscillations in the window size. In other words, the use of the $\alpha - \beta$ region provides a damping effect.

Even though the goal of this mechanism is to avoid congestion by limiting the number of buffers used at the bottleneck, it may not be able to achieve this when there are a large number of "bulk data" connections going through a bottleneck with a small buffer size. However, Vegas will successfully limit the the window growth of connections with smaller round-trip times. The mechanisms in Vegas are not meant to be the ultimate solution, but they represent a considerable enhancement to those in Reno.

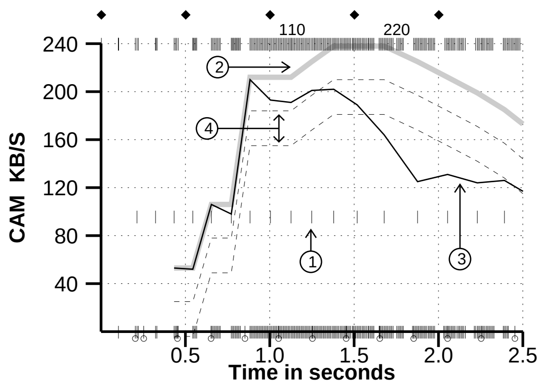

Figure 8: Congestion detection and avoidance in Vegas.

Fig. 7 shows the behavior of TCP Vegas when there is no other traffic present; this is the same condition that Reno ran under in Fig. 5. 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 (Fig. 8) keyed to the following explanation:

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.

Fig. 9 shows a trace of a Vegas connection transferring one MByte of data, while sharing the bottleneck router with *tcplib* traffic. The third 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 and the thick line is a running average (size 3). The bottom graph shows the output of the bottleneck link which has a maximum bandwidth of 200KB/s. The figure clearly shows Vegas' congestion avoidance mechanisms at work and how its throughput adapts to the changing conditions on the network. For example, as the background traffic increases at 3.7 seconds (third graph), the Vegas connection detects it and decreases its window size (top graph) which results in a reduction in its sending rate (second graph). When the background traffic slows down at 5, 6 and 7.5 seconds, the Vegas connection increases its window size, and correspondingly its sending rate. The bottom graph shows that most of the time there is a 100% utilization of the bottleneck link.

In contrast, Fig. 10 shows the behavior of Reno under similar conditions. It shows that there is very little correlation between the window size and the level of background traffic. For example, as the background traffic increases at 3.7 seconds, the Reno connection keeps increasing its window size until there is congestion. This results in losses, both to itself and to connections which are part of the background traffic. The graph only shows the first 10 seconds of the one MByte transfer; it took 14.2 seconds to complete the transfer. The bottom graph shows that there is under-utilization of the bottleneck link.

The important thing to take away from this information is that Vegas' increased throughput is not a result of its taking bandwidth away from Reno connections, but due to a more efficient utilization of the bottleneck link. In fact, Reno connections do slightly better when the background traffic is running in top of Vegas as compared to when the traffic is running on top of Reno (see Section 4).

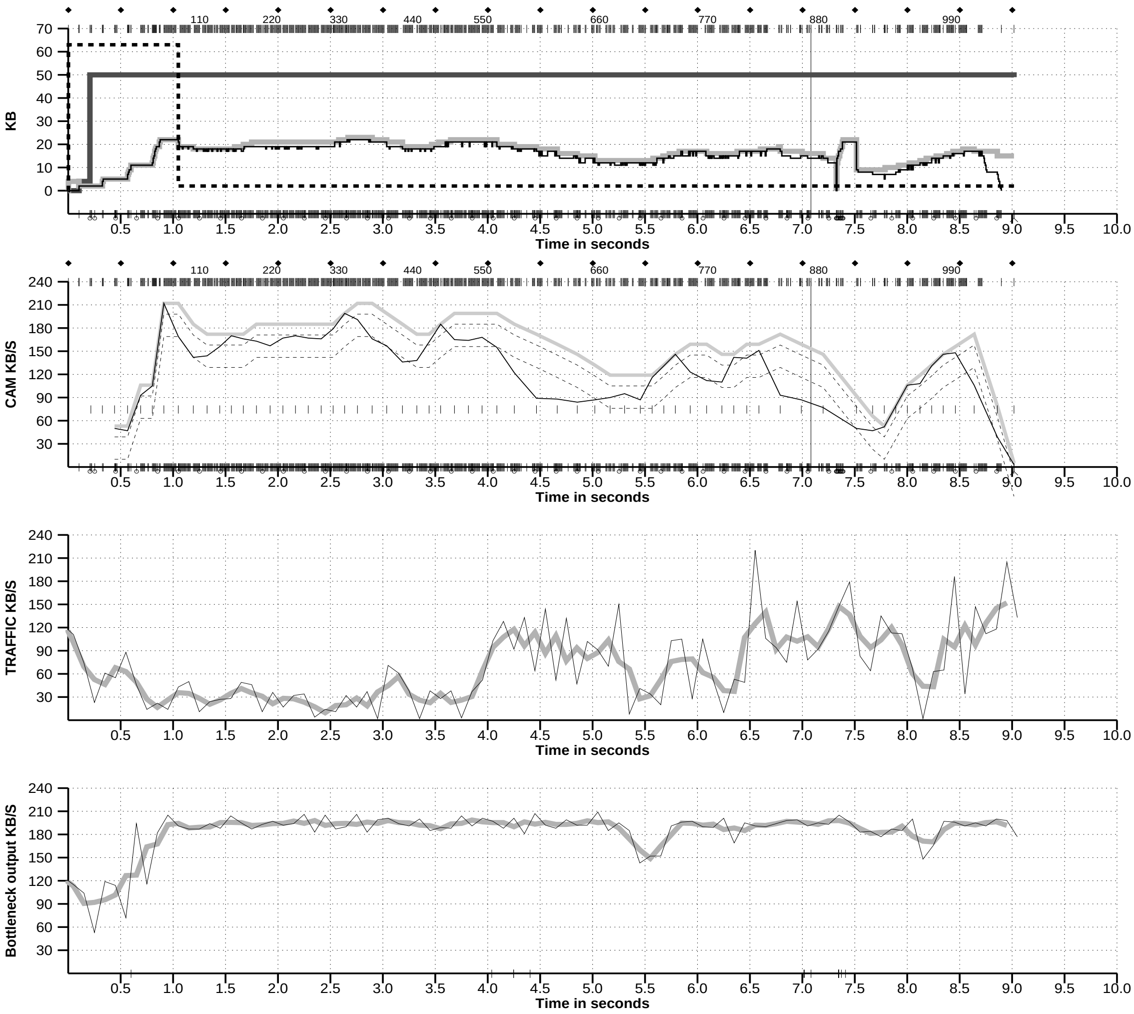

Figure 9: TCP Vegas with *tcplib*-generated background traffic.

3.3 Modified Slow-Start Mechanism

TCP is a ‘self-clocking’ protocol, that is, it uses ACKs as a ‘clock’ to strobe new packets into the network [7]. When there are no segments in transit, such as at the beginning of a connection or after a retransmit timeout, there will be no ACKs to serve as a strobe. Slow-start is a mechanism used to gradually increase the amount of data in-transit; it attempts to keep the segments uniformly spaced. The basic idea is to send only one segment when starting or restarting after a loss, then as the ACKs are received, to send and extra segment in addition to the amount of data acknowledged in the ACK. For example, if the receiving host sends an acknowledgment for each segment it receives, the sending host will send 1 segment during the first RTT, 2 during the second RTT, 4 during the third, and so on. It is easy to see that the increase is exponential, doubling its sending rate on each RTT.

The behavior of the slow-start mechanism can be seen in Fig. 3 and Fig. 10. It occurs twice, once during the interval between 0 and 1 seconds, and again in the interval between 2 and 2.5 seconds; the latter after a coarse-grain timeout. The behavior of the initial slow-start is different from the ones that occur later in one important respect. During the initial slow-start, there is no *a priori* knowledge of the available bandwidth that can be used to stop the exponential growth of the window, whereas when slow-starts

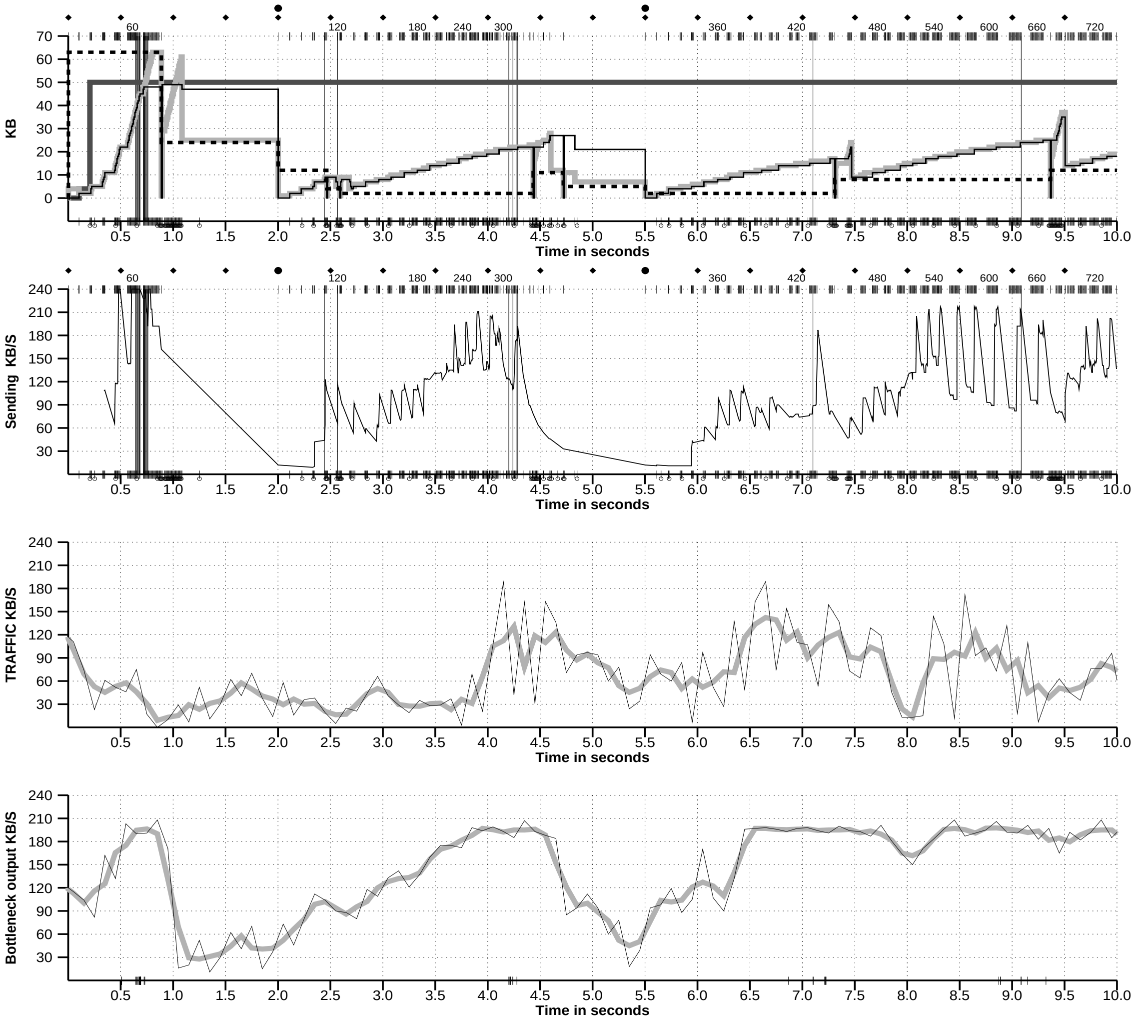

Figure 10: TCP Reno with *tcplib*-generated background traffic.

occurs in the middle of a connection, there is the knowledge of the window size used when the losses occurred —Reno considers half of that value to be safe.

Whenever a retransmit timeout occurs, Reno sets the threshold window to one half of the congestion window. The slow-start period ends when the exponentially increasing congestion window reaches the threshold window, and from then on, the increase is linear, or approximately one segment per RTT. Since the congestion window stops its exponential growth at half the previous value, it is unlikely that losses will occur during the slow-start period.

However, there is no such knowledge of a safe window size when the connection starts. If the initial threshold window value is too small, the exponential increase will stop too early, and it will take a long time—by using the linear increase—to arrive at the optimal congestion window size. As a result, throughput suffers. On the other hand, if the threshold window is set too large, the congestion window will grow until the available bandwidth is exceeded, resulting in losses on the order of the number of available buffers at the bottleneck router; these losses can be expected to grow as network bandwidth increases.

What is needed is a way to find a connection's available bandwidth which does not incur these kind of losses. Towards this end, we incorporated our congestion detection mechanism into slow-start with only minor modifications.

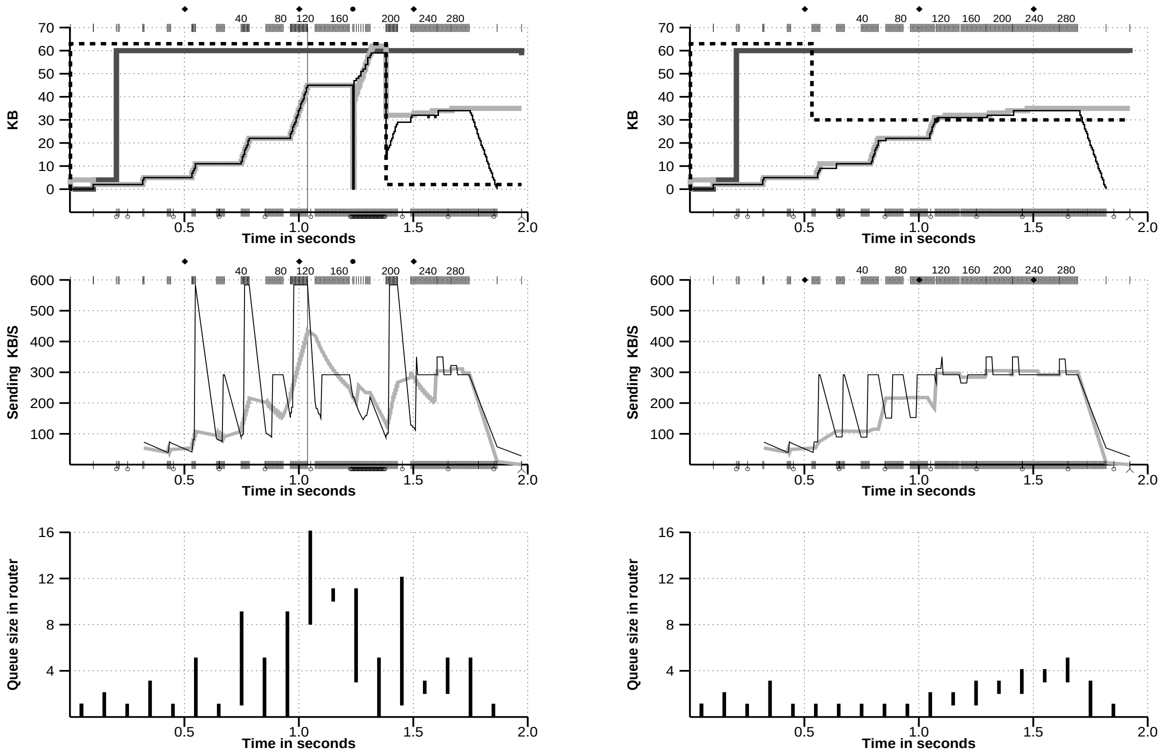

Figure 11: TCP Vegas on the left, experimental on the right.

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 the equivalent of one router buffer, Vegas changes from slow-start mode to linear increase/decrease mode.

The behavior of the modified slow-start can be seen in Fig. 7 and Fig. 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 an extra segment for each ACK received). This mechanism is highly successful at preventing the losses incurred during the initial slow-start period, as quantified in the next section.

Two problems remain during any slow-start period. First, segments are sent at a rate higher than the available bandwidth—up to twice the available bandwidth, depending on the ACKing frequency (e.g., every segment or every two segments). This results on the bottleneck router having to buffer up to half of the data sent on each RTT, thereby increasing the likelihood of losses during the slow-start period. Moreover, as network speeds increase, so does the amount of buffering needed. Second, while Vegas' congestion avoidance mechanism during the initial slow-start period is quite effective, it can still overshoot the available bandwidth, and depends on enough buffering at the bottleneck router to prevent losses until realizing it needs to slow down. Specifically, if the connection can handle a particular window size, then Vegas will double that window size—and as a consequence, double the sending rate—on the next RTT. At some point the available bandwidth will be exceeded.

We have experimented with a solution to both problems. To simplify the following discussion, we refer to the alternative version of Vegas with an experimental slow-start mechanism as Vegas*. Vegas* is based on using the spacing of the acknowledgments to gauge the available bandwidth. The idea is similar to Keshav's Packet-Pair probing mechanism [13], except that it uses the spacing of four segments sent during the slow-start period rather than two. (Using four segments results in a more robust algorithm than using two segments.) This available bandwidth estimate is used to set the threshold window with an appropriate value, which makes Vegas* less likely to overshoot the available bandwidth.

Specifically, as each ACK is received, Vegas* schedules an event at a certain point in the future, based on its available bandwidth estimate, to increase the congestion window by one maximum segment size. This is in contrast to increasing the window immediately upon receiving the ACK. For example, assume the RTT is 100ms, the maximum segment size is 1 KByte, and the available bandwidth estimate is currently 200 KB/s. During the slow-start period, time is divided into intervals of length equal to one RTT. If during the current RTT interval we are expecting 4 ACKs to arrive, then Vegas* uses the bandwidth estimate (200KB/s) to guess the spacing between the incoming ACKs (1KB / 200KB/s = 5ms) and as each ACK is received, it schedules an event to increase the congestion window (and to send a segment) at 20ms (5×4) in the future.

The graphs in Fig. 11 show the behavior of Vegas (left) and Vegas* (right) during the initial slow-start. For this set of experiments, the available bandwidth was 300KB/s and there were 16 buffers at the router. Looking at the graphs on the left, we see that a packet is lost at around 1 second (indicated by the thin vertical bar) as a result of sending at 400KB/s. This is because Vegas detected no problems at 200KB/s, so it doubled its sending rate, but in this particular case, there were not enough buffers to protect it from the losses. The bottom graph demonstrates the need to buffer half of the data sent on each RTT as a result of sending at a rate twice the available bandwidth.

The graphs on the right illustrate the behavior of Vegas*. It sets the threshold window (dashed line) based on the available bandwidth estimate. This results in the congestion window halting its exponential growth at the right time—when the sending rate equals the available bandwidth and preventing the losses. The middle graph shows that the sending rate never exceeds the available bandwidth (300KB/s) by much. Finally, the bottom graph shows that Vegas* does not need as many buffers as Vegas.

Notice that while the available bandwidth estimate could be used to jump immediately to the available bandwidth by using rate control during one RTT interval, congestion would result if more than one connection did this at the same time. Even though it is possible to congest the network if more than one connection does slow-start at the same time, there is an upper bound on the number of bytes sent during the RTT when congestion occurs regardless of the number of connections simultaneously doing slow-start—about twice the number of bytes that can be handled by the connection. There is no such limit if more than one connection jumps to use the available bandwidth at once. Hence, we strongly recommend against doing this unless it is known *a priori* that there are no other connections sharing the path, or if there are, that they won't increase their sending rate at the same time.

Although these traces illustrate how Vegas*'s experimental slow-start mechanism does in fact address the two problems with Vegas outlined above, simulation data indicates that the new mechanism does not have a measurable impact on throughput, and only marginally improves the loss rate. While additional simulations might expose situations where Vegas* is more beneficial, we have decided to not include these modifications in Vegas. Also, the results presented in Section 4 are for Vegas, not Vegas*.

4 Performance Evaluation

This section reports and analyzes the results from both the Internet and the simulator experiments. The results from the Internet experiments are evidence that Vegas' enhancements to Reno produce significant improvements on both the throughput (37% higher) and the number of losses (less than half) under real conditions. The simulator experiments, allow us to also study related issues such as how do Vegas connections affect Reno connections, and what happens when all connections are running over Vegas. Note that because it is simple to move a protocol between the simulator and the "real world", the all numbers reported in this section are for exactly the same code

4.1 Internet Results

We first present measurements of TCP over the Internet. Specifically, we measured TCP 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,

Table 1: 1MByte transfer over the Internet.

	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 2: Effects of transfer size 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

New York and Washington DC. The results are derived from a set of runs over a seven day period from January 23-29, 1994. Each run consists of a set of seven transfers from UA to NIH—Reno sends 1MB, 512KB, and 128KB, a version of Vegas with $\alpha = 1$ and $\beta = 3$ (denoted Vegas-1,3) sends 1MB, 512KB, and 128KB, and second version of Vegas with $\alpha = 2$ and $\beta = 4$ (denoted 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 1 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. When comparing Vegas and Reno within each run, Vegas outperforms Reno 92% of the time and across all levels of congestion; i.e., during both the middle of the night and during periods of high load. Also, the throughput was a little higher with the bigger thresholds, since the Vegas connection used more buffers at the bottleneck router which could be used to fill bandwidth gaps occurring when the background traffic slowed down. However, the higher buffer utilization at the bottleneck also resulted in higher losses and slightly higher delays. We prefer the more conservative approach of using fewer resources, so have settled on avoidance thresholds of $\alpha = 1$ and $\beta = 3$.

Because we were concerned that Vegas' throughput improvement depended on large transfer sizes, we also varied the size of the transfer. Table 2 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.

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 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. Note that if the transfer size is smaller than about twice the bandwidth-delay product, then there will be no losses for neither Vegas nor Reno (as-

Table 3: One-on-one (300KB and 1MB) transfers.

	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

suming the bottleneck router has enough buffers to absorb temporary sending rates above the connections available bandwidth).

4.2 Simulation Results

This subsection reports the results of series of experiments using the x-kernel based simulator. 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 subsection are on the network configuration shown in Fig. 6. We have also run other topologies and different bandwidth-delay parameters, with similar results.

4.2.1 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 3 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.[8] 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.

Since the probability that there are exactly two connections at one time is small in real life, we modified the experiment by adding *tcplib* background traffic. The results were similar except for the Reno/Vegas experiment in which Reno only had a 6% increase in its retransmission, versus the 43% when there was no background traffic.

This 43% increase in the losses of Reno for the Reno/Vegas case is explained as follows. The Vegas connection starts first, and is using the full bandwidth (200KB/s) by the time the Reno connection starts. When Vegas detects that the network is starting to get congested, it decreases its sending rate to between 80 and 100KB/s. The losses incurred by Reno (about 48KB), are approximately the losses Reno experiences when it is running by itself on a network with 100 to 120KB/s of available bandwidth and around 15 available buffers at the bottleneck router. The reason the losses where smaller for the 300KB transfer in the Reno/Reno experiment is that by the time the 300KB transfer starts, the 1MB connection has stopped transmitting due to the losses in its slow-start, and it won't start sending again until it times out at around 2 seconds. A Reno connection sending 300KB when there is 200KB/s of available bandwidth and 20 buffers at the

[8] 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.

Table 4: 1MByte transfer with *tcplib*-generated background Reno traffic.

	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

bottleneck router only losses about 3KB.

This type of behavior is characteristic of Reno: by slightly changing the parameters in the network, one can observe major changes in Reno's behavior. Vegas, on the other hand, does not show as much discontinuity in its behavior.

4.2.2 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 Fig. 6, and a 1MByte transfer is running between Host2a and Host2b. In this set of experiments, the *tcplib* traffic is running over Reno.

Table 4 gives the results for Reno and two versions of Vegas—Vegas-1,3 and Vegas-2,4. 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. Again 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 is mostly unaffected by the type of connection doing the 1Mbyte transfer.

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 significantly (less than 4%).

4.2.3 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.

- *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 [18]. Thus, we modified the experiments 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 Fig. 5, 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.

5 Discussion

Throughput and losses are not the only metrics by which a transport protocol is evaluated. This section discusses several other issues that must be addressed. It also comments on the relationship between this work and other efforts to improve end-to-end performance on the Internet.

5.1 Fairness

If there is more than one connection sharing a bottleneck link, we would like for each connection to receive an equal share of the bandwidth. Unfortunately, given the limited amount of information currently available at the connection endpoints, this is unlikely to happen without some help from the routers. Given that no protocol is likely to be perfectly fair, we need a way to decide whether its level of fairness is acceptable or not. Also, given that so far the Internet community has found Reno's level of fairness acceptable, we decided to compare Vegas' fairness levels to Reno's and judge it in those terms.

Before there can be any comparisons, we need a metric. We decided to use Jain's *fairness index* [11], which is defined as follows: given a set of throughputs $(x_1, x_2, \ldots, x_n)$ the following function assigns a fairness index to the set:

$$f(x_1, x2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$$

Given that the throughputs are nonnegative, the fairness index always results in numbers between 0 and 1. If all throughputs are the same, the fairness index is 1. If only k of the n users receive equal throughput and the remaining $n - k$ users receive zero throughput, the fairness index is k/n.

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 appropriate results averaged.

In the case of 2 and 4 connections, with each connection transferring 8 MB, Reno was slightly more fair than Vegas when all connections had the same propagation delay (0.993 vs. 0.989), but Vegas was slightly more fair than Reno when the propagation delay was larger for half of the connections (0.962 vs. 0.953). In the experiments with 16 connections, with each connection transferring 2MB, Vegas was more fair than Reno in all experiments regardless of whether the propagation delays were the same or not (0.972 vs. 0.921).

To study the effect that Reno connections have over Vegas connections (and vice versa) we ran 8 connections, each sending 2 MB of data. The experiment consisted of running all the connections on top of Reno, all the connections on top of Vegas, or one half on top on Reno and the other half on top of Vegas. There was little difference between the fairness index of the eight connections running a particular TCP implementation (Vegas or Reno) and the fairness index of the four connections running the same TCP implementation and sharing the bottleneck with the four connections running the other TCP implementation. Similarly, we saw little difference in the average size of the bottleneck queue.

In another experiment, we ran four connections over background traffic. For this experiment, Vegas was always more fair than Reno. Overall, we conclude that Vegas is no less fair than Reno.

5.2 Stability

A second concern is stability—it is undesirable for a protocol to cause the Internet to collapse as the number of connections increases. In other words, as the load increases, each connection must recognize that it should decrease its sending rate. Up to the point where the window can be greater than one maximum segment size, Vegas is much better than Reno at recognizing and avoiding congestion—we have already seen that Reno does not avoid congestion, on the contrary, it periodically creates congestion.

Once the load is so high that on average each connection can only send less than one maximum segment's worth of data, Vegas behaves like Reno. This is because this extreme condition implies that coarse-grain timeouts are involved, and Vegas uses exactly the same coarse-grain mechanism as Reno. Experimental results confirm this

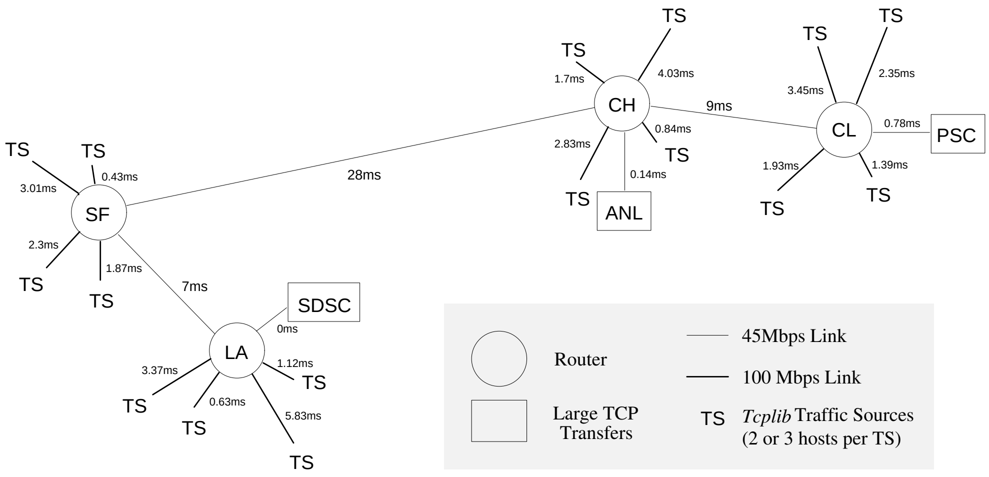

Figure 12: Complex simulation network.

intuition: running 16 connections, with a 50ms one-way propagation delay, through a router with either 10 or 20 buffers and 100 or 200KB/s of bandwidth produced no stability problems.

We have also simulated complex network topologies like the one shown in Fig. 12, which consists of 16 traffic sources each of which contains two or three hosts. Each host, in turn, is running *tcplib*-based traffic. The rectangular boxes represent sources of "bulk data" transfers. The resulting traffic consists of nearly a thousand new connections being established per simulated second, where each connection is either a TELNET, FTP, SMTP or NNTP conversation. No stability problems have occurred in any of our simulations when all of the connections are running Vegas.

In summary, there is no reason to expect Vegas to lead to network collapse. One reason for this is that most of Vegas' mechanisms are conservative in nature—its congestion window never increases faster than Reno's (one maximum segment per RTT), the purpose of the congestion avoidance mechanism is to *decrease* the congestion window before losses occur, and during slow-start, Vegas stops the exponential growth of its congestion window before Reno would under the same conditions.

5.3 Queue Behavior

Given that Vegas purposely tries to occupy between one and three extra buffers along the path for each connection, it seems possible that persistent queues could form at the bottleneck router if the whole world ran Vegas. These persistent queues would, in turn, add to the latency of all connections that crossed that router.

Since the analytical tools currently available are not good enough to realistically model and analyze the behavior of either Reno or Vegas, we must rely on simulations to answer this issue. Our simulations show that average queue sizes under Reno and Vegas are approximately the same. However, they also show that TELNET connections in *tcplib* experience between 18 and 40% less latency, on average, when all the connections are Vegas instead of Reno. This seems to suggest that if the whole world ran Vegas, Internet latency would not be adversely affected.

5.4 BSD Variations

TCP has been a rather fluid protocol over the last several years, especially in its congestion control mechanism. Although the general form the original mechanism described in [7] has remained unchanged in all BSD-based implementations (e.g., Tahoe, Reno, BNR2, BSD 4.4), many of the "constants" have changed. For example, some implementations ACK every segment and some ACK every other segment; some increase the window during linear growth by one segment per RTT and some increase by half a segment per RTT plus 1/8th the maximum segment size per ACK received during that RTT; and finally, some use the timestamp option and some do not.

We have experimented with most of these variations and have found the combination used in our version of Reno, as reported in this paper, to be the among the most effective. For example, we found the latest version of TCP, that found in BSD 4.4-lite,[9] achieves 14% worse throughput than our Reno during Internet type simulations [2]. Also, others[1] have compared Vegas with the SunOS implementation of TCP, which is derived from Reno, and have reached conclusions similar to those in this paper.

5.5 Alternative Approaches

In addition to improving TCP's congestion control mechanism, there is a large body of research addressing the general question of how to fairly and effectively allocate resources in the Internet. We conclude this section by discussing the relevance of TCP Vegas to these other efforts.

One example is gaining much attention is the question of how to guarantee bandwidth to real-time connections. The basic approach requires that a more intelligent buffer manager be placed in the Internet routers [14]. One might question the relevance of TCP Vegas in light of such mechanisms. We believe end-to-end congestion control will remain very important for two reasons. First, a significant fraction of the data that will flow over the Internet will not be of a real-time nature; it will be bulk-transfer applications (e.g., image transfer) that want as much bandwidth as is currently available. These transfers will be able to use Vegas to compete against each other for the available bandwidth. Second, even for a real-time connection, it would not be unreasonable for an application to request (and pay for) a minimally acceptable bandwidth guarantee, and then use a Vegas-like end-to-end mechanism to acquire as much additional bandwidth as the current load allows.

As another example, selective ACKs [8, 9] 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.

[9] This is the implementation of TCP available at ftp.cdrom.com, dated 4/10/94.

6 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 37 to 71% 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. We have also given evidence that Vegas is just as fair as Reno, that it does not suffer from stability problems, and that it does not adversely affect latency.

A Detailed Graph Description

To assist the reader develop a better understanding of the graphs used throughout this paper, and to gain a better insight of Reno's behavior, we describe in detail one of these graphs. Figure 13 is a trace of Reno when there is other traffic through the bottleneck router. The numbers in parenthesis refer to the type of line in the graph.

In general, output is allowed while the UNACK-COUNT (4) (number of bytes sent but not acknowledged) is less than the congestion window (3) and less than the send window (2). The purpose of the congestion window is to prevent, or more realistically in Reno's case, to control congestion. The send window is used for flow control, it prevents data from being sent when there is no buffer space available at the receiver.

The threshold window (1) is set to the maximum value (64KB) at the beginning of the connection. Soon after the connection is started, both sides exchange information on the size of their receive buffers, and the send window (2)

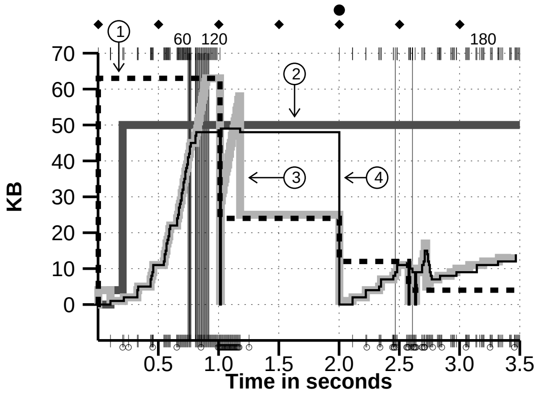

Figure 13: TCP windows graph.

is set to the minimum of the sender's send buffer size and the receiver's advertized window size.

The congestion window (3) increases exponentially while it is less than the threshold window (1). At 0.75 seconds, losses start to occur (indicated by the tall vertical lines). More precisely, the vertical lines represent segments that are later retransmitted (usually because they were lost). At around 1, second a loss is detected after receiving 3 duplicate ACKs and Reno's Fast Retransmit and Fast Recovery mechanisms go into action. The purpose of these mechanisms is to detect losses before a retransmit timeout occurs, and to keep the pipe full (we can think of a connection's path as a water pipe, and our goal is to keep it full of water) while recovering from these losses.

The congestion window (3) is set to the maximal allowed segment size (for this connection) and the UNACK-COUNT is set to zero momentarily, allowing the lost segment to be retransmitted. The threshold window (1) is set to half the value that the congestion window had before the losses (it is assumed that this is a safe level, that losses won't occur at this window size).

The congestion window (3) is also set to this value after retransmitting the lost segment, but it increases with each duplicate ACK (segments whose acknowledgement number is the same as previous segments and carry no data or new window information). Since the receiver sends a duplicate ACK when it receives a segment that it cannot acknowledge (because it has not received all previous data), the reception of a duplicate ACK implies that a packet has left the pipe.

This implies that the congestion window (3) will reach the UNACK-COUNT (4) when half the data in transit has been received at the other end. From this point on, the reception of any duplicate ACKs will allow a segment to be sent. This way the pipe can be kept full at half the previous value (since losses occurred at the previous value, it is assumed that the available bandwidth is now only half its previous value). Earlier versions of TCP would begin the slow-start mechanism when losses were detected. This implied that the pipe would almost empty and then fill up again. Reno's mechanism allows it to stay filled.

At around 1.2 seconds, a non-duplicate ACK is received, and the congestion window (3) is set to the value of the threshold window (1). The congestion window was temporarily inflated when duplicate ACKs were received as a mechanism for keeping the pipe full. When a non-duplicate ACK is received, the congestion window is reset to half the value it had when losses occurred.

Since the congestion window (3) is below the UNACK-COUNT (4), no more data can be sent. At 2 seconds, a retransmit timeout occurs (see black circle on top), and data starts to flow again. The congestion window (3) increases exponentially while it is below the threshold window (1). A little before 2.5 seconds, a segment is sent that will later be retransmitted. Skipping to 3 seconds, we notice the congestion window (3) increasing linearly because it is above the threshold window (1).

Acknowledgments

Thanks to Sean W. O'Malley for his help and insightful comments and 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] J.-S. Ahn, P. B. Danzig, Z. Liu, and L. Yan. Experience with TCP Vegas: Emulation and Experiment. In *Proceedings of the SIGCOMM '95 Symposium*, Aug. 1995. In press.

[2] L. S. Brakmo and L. L. Peterson. Performance Problems in BSD4.4 TCP. *ACM Computer Communication Review*, 1995. In press.

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

[4] S. Floyd. TCP and Successive Fast Retransmits. Technical report, Lawrence Berkeley Laboratory, 1994. Available from anonymous ftp from ftp.ee.lbl.gov:papers/fastretrans.ps.

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

[6] 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.

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

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

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

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

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

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

[13] S. Keshav. A Control-Theoretic Approach to Flow Control. In *Proceedings of the SIGCOMM '91 Symposium*, pages 3–15, Sept. 1991.

[14] D. C. . S. S. R. Braden. Integrated Services in the Internet Architecture: an Overview. Request for Comments 1633, Sept. 1994.

[15] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Publishing Co., New York, 1994.

[16] 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.

[17] 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.

[18] 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.

List of Figures

List of Tables