Evaluation of TCP Vegas: Emulation and Experiment

Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
http://excalibur.usc.edu/research/vegas/doc/vegas.html

Abstract

This paper explores the claims that TCP Vegas [2] both uses network bandwidth more efficiently and achieves higher network throughput than TCP Reno [6]. It explores how link bandwidth, network buffer capacity, TCP receiver acknowledgment algorithm, and degree of network congestion affect the relative performance of Vegas and Reno.

1 Introduction

Jacobson released his TCP slow-start flow control algorithm [5] in the Tahoe distribution of BSD UNIX and revised it two years later for the Reno distribution [6]. Since then, researchers have implemented the RFC 1323 extensions – bigger TCP windows and time-stamped based RTT exchange – to improve TCP performance over high bandwidth connections. The RFC 1323 extensions do not, however, implement congestion *avoidance*. Last year, Brakmo, O'Malley and Peterson [2] claimed that their sender-side congestion avoidance algorithm, dubbed TCP Vegas, yielded 40-70% better throughput while retransmitting 2-5 times fewer segments than TCP Reno, both in simulation and in live, wide-area Internet measurements. We ported Vegas 0.8 to SunOS 4.1.3 and compared its performance to a native implementation of TCP Reno to test these claims and to attempt to provoke Vegas to misbehave.

In this paper, we confirm that TCP Vegas yields higher network throughput and transfers bytes more efficiently than TCP Reno. Further, we find that Vegas keeps less data in the network than Reno, resulting in shorter RTT averages and variances. We also find that Vegas dramatically outperforms Reno when the destination employs the TCP Tahoe rather than Reno acknowledgment algorithm. In contrast to Brakmo's study, we report more modest throughput gains when Vegas sends to Reno receivers and that in head-to-head transfers, Reno steals bandwidth from Vegas; Vegas' congestion avoidance algorithm intentionally lowers its transmission rate under heavy congestion.

In contrast to Brakmo's, our experiments employed native Reno transmitters and receivers running the NetBSD-1.0 operating system [10]. Further, we did not run the native x-kernel TCP Vegas [1]; we used our port of it to SunOS 4.1.3. We conducted experiments on both an emulated wide-area network and on the live Internet. The emulated network enabled us to explore link speed and propagation delay, switch buffer sizes, and traffic workloads, while still employing actual Reno and Vegas sources[1].

1.1 Congestion Avoidance

Vegas' principle innovation, its congestion avoidance algorithm, replaces slow-start's linear growth regimen. Recall that slow-start TCP, after exponentially opening its congestion window to a safe value, continues to increase the congestion window linearly, aiming to consume network bandwidth that becomes available. Eventually, linear growth either reaches the receiver's advertized window, or it congests some network buffer, and one or more packet losses occur.

In place of constant linear growth, Vegas can increment, decrement, or not adjust the window by one segment every RTT. Vegas aims at keeping a small backlog of data enqueued at network switches. When this backlog builds, Vegas decreases its window; when the backlog

This research was funded in part by AFOSR award number F49620-93-1-0082, NSF NYI award NCR-9457518, NSF small-scale infrastructure grant number CDA-9216321, and Hughes Applied Information Systems, contract ECS-00009.

[1] Source code for our WAN emulator and Vegas port to SunOS is available from the URL at the top of this page.

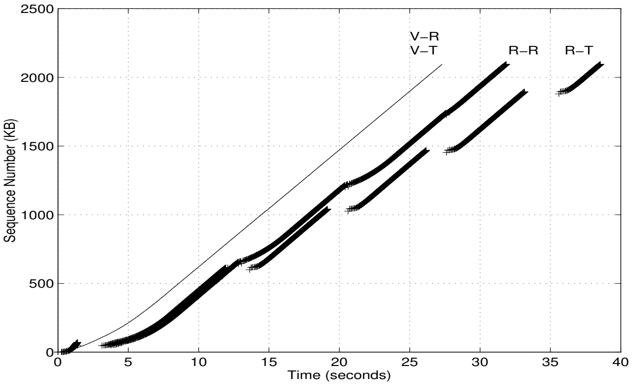

Figure 1: Various 2MB transfers with no competing traffic.

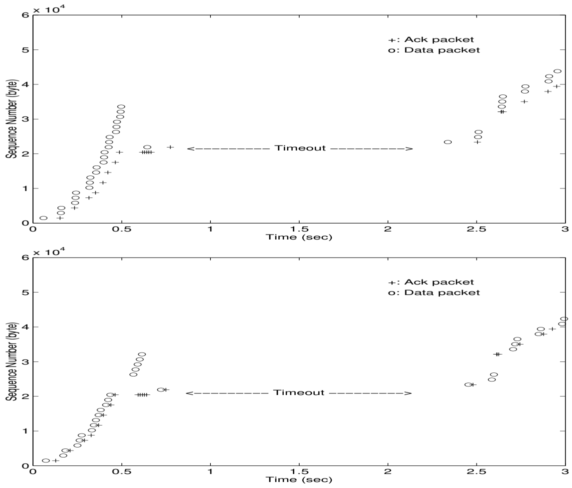

Figure 2: Packet trace of R-R transfer. Top: Reno sender. Bottom: Reno receiver.

shrinks, Vegas increases it. Because Vegas' congestion avoidance algorithm reduces packet drops, its changes to Reno's retransmission algorithm are infrequently invoked.

1.2 Benchmarking TCP

A good way to visualize the differences between Tahoe, Reno, and Vegas is to study sequence number versus time plots of their wide-area conversations. In this section, we present small tutorial examples. Later in the paper, we present plots taken from live and emulated networks under various traffic loads.

Figure 1 plots sequence number versus time of four separate 2MB transfers. Labels R-T, R-R, V-R, and V-T denote Reno-Tahoe, Reno-Reno, Vegas-Reno, and Vegas-Tahoe transmitter-receiver pairs. (Note that since Vegas does not change the Reno receiver, Vegas expects a Reno receiver). The senders and receivers were separated by the four-hop, 50ms propagation delay, 800KBIT/S bottleneck link, 16KB buffer switch, topology shown in Figure 9.

Despite the lack of competing traffic, shortly after the R-R and R-T transfers begin, both experience a 2-second timeout. Figure 2 focuses on the first few seconds of a Reno-Reno transfer, this time in the presence of a single competing sender. The figure plots data and acknowledgment sequence numbers versus time, as captured by packet sniffers at the sender and receiver. The figure shows slow-start's exponential growth, which increases the congestion window by a segment for every ACK received. Notice that the receiver employs delayed acknowledgments, sending one ACK for every two data segments received. About half a second into the transfer, the lower trace reveals that a four packet gap fails to reach the receiver, although five more packets arrive above the gap. As Reno receivers acknowledge every out-of-order data packet, the lower trace shows a cluster of five ACKs at the same sequence number. About 50ms later, as captured in the upper trace, these ACKs arrive at the sender. The third of these five duplicate ACKs causes the Reno sender to retransmit a single packet and wait for the corresponding acknowledgment. Although a new ACK eventually arrives, the gap between the highest transmitted segment and the highest acknowledged segment is so large, that Reno must wait for its retransmission timeout to expire. Reno, faced with 4 drops in a single RTT always pays a timeout; depending on the window size, its fast recovery algorithm may or may not patch 2 or 3 segment drops in a single RTT.

Recall that the retransmission timeout is set to the average RTT plus 4 times its variance [11]. Since the RTT statistics are measured in 500ms granularity clock ticks, typical timeouts last 2,000ms.

Refer back to the R-R line of Figure 1. Given the initial drop and subsequent timeout, Reno sets its congestion window threshold so that it can switch from exponential to linear growth. After the timeout, Reno's congestion avoidance algorithm avoids further packet drops until time 13s. Eventually, even linear growth causes a drop, and we see an inflection point in the graph. Careful inspection reveals three inflection points besides the timeout, indicating that Reno drops packets and halves its congestion window three times during this 2MB transfer. Only the first drop causes a timeout; the fast recover algorithm successfully patches the other three drops[2].

In contrast, given a Tahoe receiver, the R-T line shows the transfer stalls five times, once for each drop. The explanation for this is simple; Tahoe receivers do not immediately acknowledge out of sequence segments.

[2] Floyd identifies situations where multiple packet drops can cost Reno several RTT to recover its pace [4].

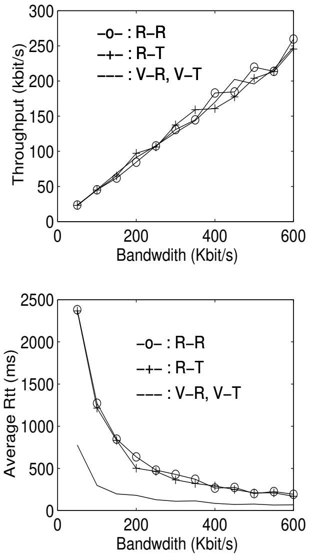

Figure 3: Metrics for two competing, 2MB transfers .

Because they send at most one ACK every 200ms, a Tahoe receiver returns too few duplicate ACKs for a Reno sender to invoke fast retransmission. Hence, as line R-T reveals, the sender experiences five, 2,000ms timeouts.

Refer now to the V-R line of Figure 1. Vegas manages to send the entire 2MB without a single packet drop. Vegas' congestion avoidance algorithm estimates the available link bandwidth and minimum propagation delay, and never increases its congestion window large enough to cause drops. On Reno's initial drop, Vegas gains two to three seconds; for each of Reno's three subsequent drops, Vegas gains a network RTT.

The initial drop explains a portion of the claimed 40-70% throughput improvement in simple topologies without competing traffic. The shorter the transfer and the higher the link speeds, the more significant the time lost by Reno's initial timeout. As the transfer durations increase, the penalty of the initial timeout decreases: Vegas and Reno send the first MB in 14 and 18 seconds, but send the second MB in 13 and 14 seconds respectively.

1.3 Utilization, Delay, Drops, Fairness

Several metrics capture a flow control algorithm's performance: how much of the available network bandwidth does it utilize, how much does it contribute to queuing delay, how frequently does it retransmit segments, and how fairly does it share the network. To illustrate these metrics, we conducted a sequence of experiments where two competing senders transfered 512KB across the network depicted in Figure 9.

Figure 3 displays these metrics for R-R, R-T, and V-R transfers. In head-to-head transfers, it shows that link utilization is roughly independent of TCP variant, but that Vegas senders retransmit five times fewer segments than Reno or Tahoe senders, and that average and variance of path queuing delay are four times lower for a pair of Vegas senders than for a pair of Reno or Tahoe senders. The lower average RTT means that Vegas keeps less data in network buffers than Reno or Tahoe.

As we tune the bottleneck link bandwidth, the metrics fluctuate a bit because each point corresponds to a single experiment. Later in the paper we show that Vegas yields bandwidth to Reno. We found that, in the small topologies with similar RTTs that we studied, all Vegas (or all Reno) senders share the available network bandwidth fairly among themselves.

Below, we summarize the algorithms employed in Vegas-0.8. In Section 3, we review our live experiments. In Section 4, we describe our wide-area network emulator, and, in Section 5, review the stability studies that the emulator enabled.

2 TCP Vegas

Vegas employs three techniques to increase throughput and avoid packet loss and subsequent retransmissions. The first technique, congestion avoidance, linearly adjusts TCP's congestion window upwards or downwards, so as to consume a constant amount of buffer space in network switches. Vegas' congestion avoidance algorithm keeps the number of segments in transit low enough that it runs the risk of having its self-clocking stall. To prevent stalling, Vegas' second technique does two things. It detects packet loss earlier than Reno and uses a slower multiplicative decrease than Reno. In its third technique, Vegas halves the slow-start growth rate, to prevent packet loss and subsequent coarse grain timeouts as illustrated in the R-R line of Figure 1.

2.1 Congestion Avoidance

Other researchers have proposed active congestion avoidance algorithms [7, 13, 8], but not transformed them into working code. Congestion avoidance schemes aim to minimize packet drops yet not underutilize the network by being too conservative. Figure 4 contrasts the Vegas and Reno congestion avoidance schemes as captured during two separate 512KB transfers between USC and LBL. The upper plots show sequence number versus time of Reno and Vegas while the lower plots highlight their respective congestion window sizes. As these

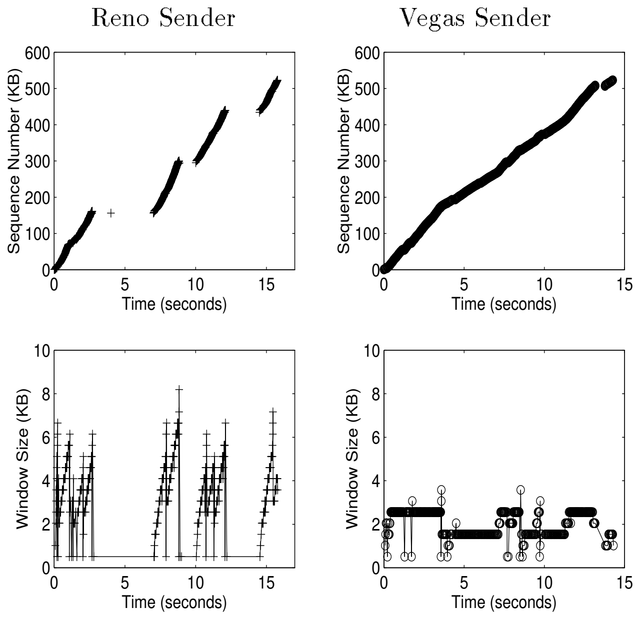

Figure 4: Live experiments. Upper: captured sequence number versus time. Lower: congestion window versus time.

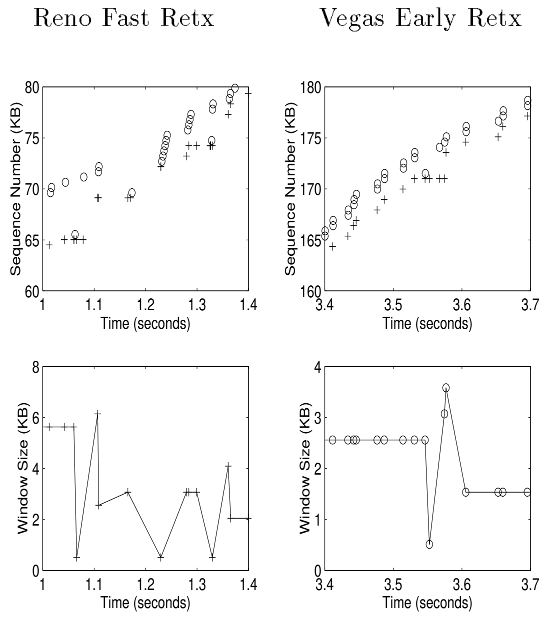

Figure 5: Close up of previous figure. Upper: sequence number versus time. Lower: congestion window versus time.

traces correspond to single measurements, we include them simply to illustrate the two algorithms, not their respective bandwidths.

Reno linearly grows its congestion window until a packet is lost. After receiving three duplicate ACKs, it sends a single segment. At this point, Reno either experiences a coarse timeout or, via its fast retransmission algorithm, recovers with half (or smaller) of its previous window size. In contrast, once Vegas settles on a window size, it neither grows nor shrinks it until the network appears faster or slower. Its non-growth is apparent in Figure 4, as its window size remains flat for periods of several seconds. Vegas increases its window at time 11, again at time 12, and revises it downward at time 13, and again at time 14, all without suffering a drop. Also notice how Reno suffers several timeouts while Vegas suffers none.

A congestion avoidance scheme adjusts its window size according to some metric. Jain's Congestion Avoidance using Round trip Delay (CARD) [7] suggests a metric based on the shape of the throughput divided by RTT curve, which Jain calls *Power*. Wang's Tri-S scheme [13] uses a metric based on the derivative of the sending rate with respect to window size, dr/dw.

In contrast, Vegas' congestion metric is an estimate of the amount of data buffered at network switches. Vegas controls its window size to keep the measured backlog bounded by parameters α and β. In our experiments, we employed $\alpha = 1$ and $\beta = 3$. Excessive backlog queues excessive data in network switches, while insufficient backlog leads to poor utilization of the available bandwidth.

In its congestion avoidance stage, once per RTT, Vegas estimates the amount of data queued at switches. It can elect to keep its window unchanged or it can revise it up or down by one segment, attempting to keep the estimated backlog within α and β. In contrast, Reno always increases the window by one segment per RTT.

When estimating the backlog, Vegas computes the difference between its *actual* and *expected* throughput. The actual throughput is simply the congestion window size divided by its average, measured RTT. The expected throughput is the window size divided by the *minimum*[3] observed RTT.

2.2 Earlier Segment Loss Detection

Reno retransmits the oldest unacknowledged segment when its coarse-grain timer expires or upon receiving three duplicate ACKs. In contrast, as soon as Vegas receives a single duplicate ACK, it can choose to retransmit the oldest unacknowledged segment. On receiving a duplicate ACK, Vegas resends the corresponding segment if the segment's fine grain timer has expired. Notice that Vegas does not retransmit a segment when the timer expires; retransmission is conditioned on receiving a duplicate ACK.

[3] Conceivably, changes in network routing can cause Vegas to overestimate the minimum network RTT.

Figure 5, a closeup of the previous Figure, illustrates Vegas' early segment loss detection in action. At time 3.55, the Vegas sender receives a duplicate ACK, discovers that the segment's fine grain retransmission timer has expired, and retransmits the segment. Vegas sends a new packet for the second duplicate, similar to the way that Reno does for duplicates that arrive during fast retransmission. We see that, when the retransmitted segment is acknowledged, Vegas reduces its congestion window multiplicatively.

Vegas records a millisecond resolution timestamp for each segment it transmits. Upon receiving an ACK, Vegas retrieves the corresponding segment's timestamp and calculates the segments' RTT in millisecond resolution. Vegas computes the fine resolution timeout interval just like Reno computes the coarse grain one, as the average RTT plus four times its variance.

Vegas employs its fine grain timestamps after a retransmission to construct a selective acknowledgment algorithm of sorts. Immediately after a retransmission, Vegas gives special attention to the first two ACK's that acknowledge previously unacknowledged data. Vegas tries to guess whether this is a multiple packet drop by checking the fine grain timeout of unacknowledged segments. If the oldest of these timeouts has expired, Vegas immediately retransmits the segment.

In contrast, Reno fast retransmission is optimized to patch single packet drops. After receiving non-duplicate ACKs following a retransmission, Reno (like Vegas) sends new segments, but does not resend previously transmitted ones. When a multiple packet loss occurs, duplicate ACKs arrive for the next dropped segment. Reno's fast retransmission algorithm, depending on window size, can patch 2 or 3 packet drops, but must resort to a timeout for drops of 4 or more packets. The Vegas algorithm patches multiple packet drops more effectively than Reno.

When Vegas retransmits a segment due to duplicate ACKs and reduces its sending window, it also tries to distinguish whether packet loss occurred due to the current or the previous window size. Vegas tries to avoid a situation that can arise with multiple packet losses: Reno's first retransmission patches the first hole, but the next hole causes a second invocation of the fast retransmission algorithm, halving the congestion window again. To avoid this ambiguity, Vegas compares the timestamp of its last window decrease with the original transmission timestamp of the retransmitted segment. If the window has already been decreased, Vegas does not multiplicatively decrease it on receiving duplicate ACKs for this segment.

2.3 Modified Slow-Start Mechanism

As soon as Vegas detects queue buildup during slow-start, it transitions to its congestion avoidance stage. Further, during slow-start, Vegas doubles its window every other RTT, as opposed to Reno's every RTT.

2.4 Overhead of Vegas

Most of Vegas ported to SunOS easily and just involved adding calls to various Vegas procedures and adding a doubly linked queue for storing each unacknowledged segment's fine grain timestamp and retransmission count. Commented source code is available from our Web server.

Running Vegas adds two sources of overhead to your TCP stack: managing the queue used for early segment loss detection and running the congestion avoidance code.

Since the early segment loss detection scheme needs to measure the fine grain RTT of every segment, Vegas allocates a queue entry upon sending a segment and frees this entry when the segment's ACK arrives. When ACKs arrive out of sequence, the oldest queue entry is inspected and the corresponding segment is retransmitted if expired. If an ACK acknowledges several segments, the corresponding queue entries are freed. Managing the queue is cheap.

The congestion avoidance code computes the actual and expected throughput, and is called once every RTT. In practice, this means a pair of arithmetic computations are made ten to a hundred times a second.

3 Live Experiments

Brakmo reported that Vegas yielded 38% higher throughput than Reno for 512KB transfers across a 17-hop Internet path. We report 4-20% Vegas speedups for 512KB transfers across a 9-hop Internet path to a development Reno receiver developed by Jacobson's group at LBL[4]. We also report 300% speedups to a Tahoe receiver just 5-hops away.

In our experiments, at various times during the day, we recorded a sequence of 20 paired observations. Each paired observation consisted of a few second wait, a

[4] Because most commercial implementations of TCP implement Tahoe rather than Reno style acknowledgments, we were only able to conduct experiments to a single Reno receiver. Jacobson's development Reno receiver immediately acknowledges every packet when a conversation starts up and switches to delayed acknowledgments after two receiver-advertised windows of data are received. The receiver switches back to immediate acknowledgments when the sender has been idle for more than a network RTT.

Reno transfer, another few second wait, followed by a Vegas transfer. We recorded sequence versus time traces for each observation.

Figures 6 and 7 show two typical pairs of observations made to the Reno and Tahoe receivers[5]. For ease of reference, we plot the conversation pairs as if they were competing transfers. The lower graphs highlight the sequence numbers of retransmitted packets. The Vegas and Reno sequence number plots of Figure 6 and the emulated experiment recorded in Figure 1 appear quite similar. Reno begins the transfer and immediately suffers a two second timeout, while Vegas completes the entire exchange without a timeout and only two retransmitted segments.

Figure 7 illustrates Vegas' congestion avoidance in action. Notice how the Vegas sender retransmits two segments at time 2.5 seconds, but continues to slow its sending rate through time 5 seconds. At time 5 seconds, Vegas notices that the network can absorb more traffic, and begins to increase its congestion window again. In contrast, Reno retransmits some segments at time 3, recovers through Reno fast retransmission, and enters its congestion avoidance phase, eventually causing another loss and timeout at time 4. While Reno obtained higher throughput in this observation pair, it retransmitted 14 segments and paid a timeout, while Vegas retransmitted 6 segments with no timeouts.

Figure 8 plots 90% confidence intervals for several metrics from one afternoon's sequence of observations. Notice that with confidence 90%, Vegas retransmits fewer segments and lowers RTT average and variance. For the Reno receiver, Vegas averaged about 20% higher throughput than Reno, although the variance of our observations is high and the confidence intervals overlap[6]. For the Tahoe receiver, Vegas' congestion avoidance enabled it to obtain 300% higher throughput than Reno. Since most receivers in the Internet implement Tahoe, switching to a Vegas transmitter may offer significant speedup.

4 WAN Emulator

To investigate Vegas' behavior under controlled workloads, we conducted experiments on our wide-area network emulator. The emulator consists of a dozen workstations, each outfitted with 2 to 6 Ethernet interfaces. An operating system patch makes each interface have the characteristic of a wide-area network link and each workstation behave as a limited buffer router. Topologies comparable to ones used in recent flow and congestion control studies can be configured.

[5] See http://excalibur.usc.edu/research/vegas/doc/live for more live traces.

[6] We would have performed experiments to more destinations, but lacked access to more Reno receivers.

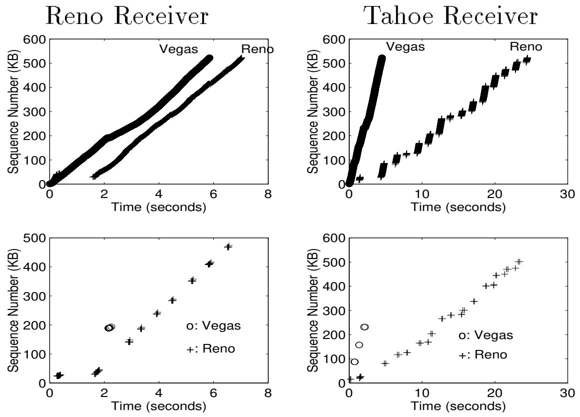

Figure 6: Live experiments: sequence number versus time and retransmitted packets versus time of Vegas and Reno senders.

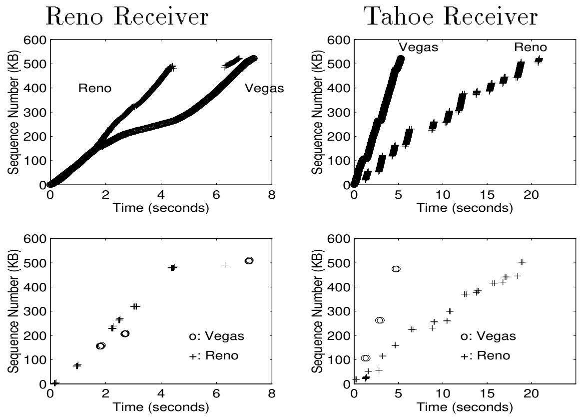

Figure 7: Live experiments: sequence number versus time and retransmitted packets versus time of Vegas and Reno senders.

After assigning a unique IP address to each interface, stringing point-to-point Ethernet cables, configuring the routing tables of each workstation, and assigning each link's bandwidth, propagation delay, bit-error rate, and output-buffer size, the emulator is ready for live experiments.

Figure 9 illustrates one emulated topology that we used. Labels "le1", "le2", "qe3" refer to network interfaces, and labels "jalama", "condesa", and "alameda" refer to specific workstations. Labels "1.1", "1.2", "2.2" correspond to the last two digits of our IP network number, 204.57.0.0. For example, if we set the link between condesa and alameda to have 25 ms propagation delay and 200 KBIT/S bandwidth, and the link between alameda and montara to 10 ms propagation delay and

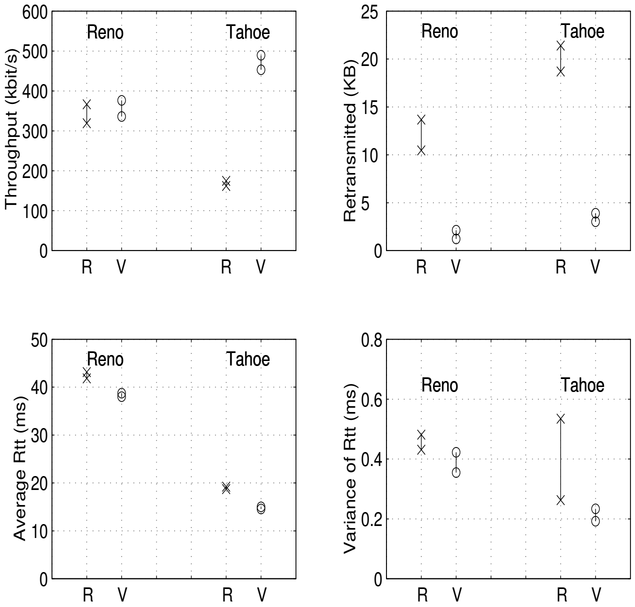

Figure 8: Metrics and 90% confidence intervals for afternoon, live Internet experiments.

56 KBIT/S bandwidth, then all packets sent from montara to condesa will be subject to 35 ms of propagation delay, and will travel from a 56 KBIT/S link, through a limited buffer, to a 200 KBIT/S link.

4.1 Implementation Details

We built the emulator from a code fragment called *hitbox* written by Thomas Skibos. The original hitbox emulates propagation delay and unrecoverable bit-errors, however it neither emulates link output-buffers nor packet transmission times, nor does it permit micro-second resolution packet scheduling. To recognize its roots, we still call our emulator code hitbox.

Hitbox is added to the operating system as a named pseudo-device, which gives programs a handle by which they can change the link propagation delay, bandwidth, buffer size, and drop rate.

In the protocol stack, hitbox sits just above the network interface modules; all outbound packets from the network layer, regardless of protocol (IP, ICMP, ARP), traverse hitbox on their way to the network interfaces. Technically, in UNIX, the network interface output routines are always called through function pointers. We install hitbox by making these function pointers call hitbox rather than the interface output routines, effectively inserting an additional level of indirection between the network and the link layers. Note that a link's emulation parameters only apply unidirectionally from sender to receiver. To emulate a bidirectional link, you must

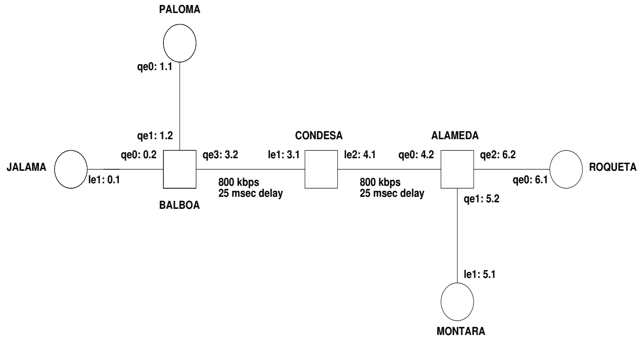

Figure 9: Emulated Topology. Default bandwidth 10 MB/S and delay 1ms, unless otherwise marked.

specify a pair of unidirectional hitboxes, one at each host. This is not a disadvantage; rather it permits the construction of asymmetric networks.

When a workstation is configured as a packet switch, it constructs a hash table mapping the packet's next hop to an instance of hitbox. If a packet is not subject to emulation, then it is immediately forwarded to the real output routine of the network interface.

When a packet arrives, hitbox computes the time to hold the packet to model queuing, transmission, and propagation delays. Further, it checks if the packet should be dropped due to limited buffer space. The transmission delay of a packet of size p bits is p/λ, where link bandwidth is λ.

Denote the link propagation delay by Δ, the queuing delay for the current packet as Q_n, the queuing delay experienced by the previous packet as Q_{n-1}, the arrival time of the current packet as T_n, and the arrival time of the previous packet as T_{n-1}. The queuing delay a packet experiences is the queuing delay that the previous packet experiences, minus the difference in packet arrival times. If the packet arrives after the previous packet departs, it experiences no queuing delay at all. Hence, $Q_n = \{Q_{n-1} - (T_n - T_{n-1})\}^+$.

So the total time to hold a packet in hitbox is the sum of the link propagation delay, transmission delay, and queuing delay, $\Delta + p/\lambda + Q_n$. Packet transmission is scheduled using the timeout mechanism common to BSD UNIX systems. When the timeout occurs, all packets with send-time less than or equal to the current time are forwarded to the network interfaces.

Unmodified, the BSD scheduler offers 10ms scheduling resolution. Since link delay and bandwidth emulation require sub-millisecond scheduling, we forced the softclock to interrupt every 1,000 μs, enduring the incidental overhead. Since the operating system uses softclock and timeouts extensively, ten times more precision incurs ten times more interrupts. Workstation performance degraded by 3%, determined by the elapsed time to quicksort 400,000 integers.

A packet is dropped if it would not fit in the switch's output-buffer. Hitbox maintains two queues: one to track the output-buffer size, the other to emulate propagation, transmission, and queuing delay. To simulate noisy network links, hitbox does not forward packets with bit-errors to the real network interface. Although these packets consume transmission time, they never arrive at the next hop. In the experiments reported here, the bit-error rate was zero.

4.2 Verification of the Emulator

We verified the correctness of the emulator using "ping" to test link propagation delay emulation, UDP blasts to verify the accuracy of link bandwidth emulation, and packet-by-packet tracing to verify the accuracy of link buffer overflow and packet scheduling. To do this, we captured packets from an emulated switch's incoming and outgoing Ethernet interfaces using a single workstation to avoid timing skew between traces. Through link speeds of 1MB/S, hitbox faithfully emulates a packet switch.

4.3 Other Emulators

We understand that a group at AT&T started but did not complete a wide-area network emulator out of Transputers [9]. We are unaware of software implementations of wide-area network emulators or of evaluation of flow control algorithms by live emulation. As we now show, the emulator serves as a testbed to check flow and switch scheduling algorithms for stability and correctness.

5 Emulation Experiments

The fact that Vegas offers higher performance under normal network conditions, does not guarantee better performance under extraordinary ones. We emulated the Figure 9 topology to study Vegas' ability to cope with severe and swiftly changing congestion, to evaluate how Vegas and Reno compete for available network bandwidth, and to contrast Vegas and Reno performance as path buffer capacity increases. In the experiments summarized here, the network frequently reached 100% utilization. As the network was saturated, our experiments tested Vegas' stability, not its throughput gains.

We used switch buffer sizes from 4-16 KB. This corresponds to 20ms-160ms of queuing delay per hop at our bottleneck link speed of 800 KBIT/S. Given larger buffer sizes, Reno's congestion window growth could cause large queuing delay for interactive traffic. Given smaller buffer sizes, deployed switches would drop NFS traffic and would be impractical. We used Reno's default 40 KB receiver advertised window size.

	Background Traffic Tahoe				
Metric	20T	20TU	10T	5T	5TS
R KB/S	5.7	5.2	9.7	1.9	13.3
V KB/S	6.8	6.4	7.5	1.2	7.8
R KB Re-tx	26.3	34.9	18.1	26.4	15.0
V KB Re-tx	5.8	7.9	2.5	14.9	1.6
	Background Traffic Vegas				
Metric	20V	20VU	10V	5V	5VS
R KB/S	6.0	4.8	15.2	3.4	28.0
V KB/S	7.8	7.1	10.6	2.0	21.2
R KB Re-tx	25.7	34.6	12.2	13.4	12.1
V KB Re-tx	4.1	7.9	0.5	2.2	0.0

Table 1: Reno (R) and Vegas (V) KB/S and retransmitted KB for 512 KB exchanges with background traffic. Background traffic employs on-off tcplib workload, except for column S. Notation: Tahoe (T), Vegas (V), UDP (U), and static (S) background workloads.

To study Vegas' response to dynamic, severe network congestion, we measured throughput and retransmitted segment counts with 2 to 20 competing, bidirectional Vegas and Reno senders. Below, we draw conclusions about how Vegas and Reno share bandwidth.

To create dynamic traffic, each sender looped between sleeping an exponentially distributed time and exchanging a heavy tailed distribution of bytes (drawn from the FTP file size distribution of our *tcplib* traffic library [3]). During some experiments, we also injected exponentially distributed 8 KB bursts of UDP traffic, as a terribly crude model of un-flow controlled, file system traffic.

Table 1 contrasts Reno and Vegas average throughput and retransmitted bytes for thirty 512 KB transfers. The table shows eight different background traffic workloads. In every case, Vegas retransmits fewer bytes than Reno. For the highest levels of congestion studied, Vegas obtained slightly higher throughput than Reno. This occured because Reno sources timed out more frequently than Vegas sources since Reno sources suffer more multiple packet drops. However, as the degree of congestion drops, Vegas sources yield bandwidth to Reno sources, because Reno sources keep more data in the network and shut down Vegas' congestion windows. As a rule of thumb, in head-to-head transfers, Reno will get 50% better throughput than Vegas.

The bottleneck link speed of columns 5T and 5V was 100 KBIT/S, rather than the 800 KIT/S used in the other experiments. We expected that Reno would be slower than Vegas, due to the high degree of congestion and multiple packet drops. To our surprise, this did not happen. Instead, while Reno suffers more timeouts, at lower link speeds these timeouts degrade its performance less.

Columns 5TS and 5VS correspond to static, infinite transfers, rather than tcplib background traffic. Under static conditions, Vegas essentially drops no packets.

Contrasting columns 10T and 10V, 5T and 5V, and 5TS and 5VS, we see that Reno steals significant bandwidth from the Vegas background senders. This is evident because single Reno senders obtain 50% to 100% better throughput with Vegas than with Reno background traffic.

Figures 10 and 11 show sequence number versus time plots of Vegas and Reno traffic in a background of 10 and 20 on/off, tcplib senders respectively. Notice how infrequently Vegas retransmits in contrast to Reno.

In all of our experiments with heavy congestion, Vegas used the network more efficiently than Reno, and under most workloads, Reno senders received a better than average share of the network bandwidth.

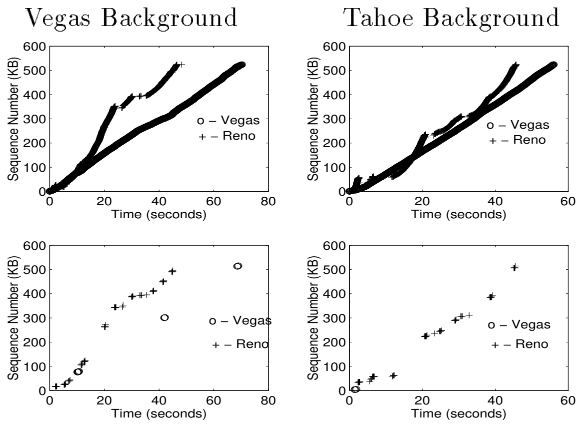

Figure 10: With ten tcplib senders, Vegas retransmits almost no segments, but Reno gets higher throughput.

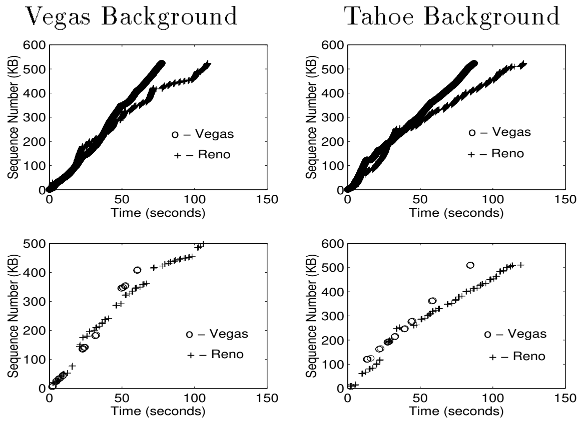

Figure 11: With 20 tcplib senders, Vegas retransmits fewer segments than Reno, and Reno's fast recovery algorithm isn't avoiding timeouts.

6 Decomposing Vegas

This section attempts to give a correct, but intuitive interpretation of our experimental results and our reading of the Vegas code.

6.1 Vegas is Conservative

Vegas, like Reno, is an additive growth, multiplicative decrease flow control algorithm. Vegas distinguishes itself from Reno in that its congestion avoidance algorithm moderates its additive growth regimen, while Reno's congestion avoidance regimen keeps growing its window size until a drop occurs. Vegas can add or subtract one segment to the congestion window every RTT. Reno's congestion avoidance regimen always adds one segment to the congestion window every RTT. In summary, Vegas' additive growth is more cautious than Reno.

Vegas maintains a fine grain retransmit timer for each outstanding segment. This gives Vegas the option of retransmitting a segment on the first duplicate ACK. At first glance, retransmitting a segment based on a fine grain clock sounds unstable. However, on closer inspection, Vegas does not blindly retransmit a segment using its fine grain timer; Vegas' early retransmit is activated by receiving a duplicate ACK. When Vegas chooses to retransmit a segment based on the fine grain timer, it reduces its congestion window by a quarter. This means that when Vegas retransmits a segment earlier than Reno, it reduces its congestion window faster than Reno would. In summary, Vegas' fine grain retransmission algorithm behaves cautiously, like Reno's fast retransmission algorithm.

Vegas' third algorithm, its modified slow-start mechanism, limits the exponential growth of slow-start to every other RTT, rather than Reno's every RTT. Vegas is more cautious than Reno.

Only one aspect of Vegas appears more aggressive than Reno. Vegas uses a multiplicative decrease of 3/4, rather than Reno's 1/2, after a retransmission. This is not always true; if Vegas has retransmitted this segment recently, it defaults to Reno's more conservative multiplicative factor. Since Vegas drops 2 to 5 times fewer segments and maintains smaller congestion windows than Reno, this more aggressive aspect of Vegas is infrequently invoked and contributes little to Vegas' speedup. Our experiments indicate that Vegas could use a multiplicative factor of 1/2 without noticeably degrading its performance.

6.2 Throughput

Vegas' improved throughput, efficiency, and delay statistics are primarily a consequence of congestion avoidance; Vegas' modified window sizing and early segment loss detection techniques contribute only second order effects. We say this because Vegas' throughput improvement comes from three effects. Through its congestion avoidance, Vegas drops fewer packets, so (a) it experiences fewer coarse timeouts, (b) it loses fewer RTTs to Reno's fast retransmission algorithm, and (c) it retransmits 2 to 5 times fewer segments than Reno.

Effects (a), (b), and (c) depend on RTT and available bandwidth. Given Reno and Vegas' shared 500ms retransmission timer granularity, a typical coarse timeout costs between 500ms and 2,000ms. Given a RTT of 100ms, a fast retransmit that patches a single segment costs Reno 100ms and perhaps some lost pace because it halved its congestion window. Given a wide-area bandwidth of 100 KB/S, each retransmitted 512 byte segment costs 5 ms of network time.

We can express the amount of time saved by Vegas in terms of nto–the decrease in coarse grain timeouts, $nfrtx$–the decrease in Reno fast retransmissions invoked (roughly the number of single packet drops), and $nrtx$–the decrease in the number of retransmitted segments.

$$time\ saved \approx \sum_{n=0}^{nto} 1250ms \ + \sum_{n=0}^{nfrtx} RTT \ + \sum_{n=0}^{nrtx} \frac{512}{bw}$$

Roughly, a coarse grain timeout costs an order of magnitude more than a single (or recoverable 2 or 3) packet drop and fast retransmission, and the time lost to retransmitted segments only matters when path bandwidth is low.

For small window sizes, both Reno and Vegas can suffer coarse timeouts when self-clocking fails due to packet loss. This is exacerbated by receiver delayed ACKs. For larger window sizes, Vegas' congestion avoidance decreases the chance of multiple segment drops. Also, Vegas recovers from multiple segment drops better than Reno.

Reno grows its congestion window to consume path buffer capacity. As its congestion window gets large, the probability of a multiple segment loss and subsequent coarse timeout increases. Since path buffer capacity grows with hop count, the relative Vegas speedup increases with hop count. Path buffer capacity probably explains why we observed 20% speedup over our 9 hop path while Brakmo [2] observed 40% speedup over his 22 hop path.

6.3 Selective ACK, Timeout Granularity, RED Gateways

Selective acknowledgments, which require cooperating TCP receivers, can eliminate the coarse grain timeouts caused by multiple packet losses in a single RTT. For this reason, selective acknowledgments would eliminate the largest term in our expression for Vegas speedup, but would not eliminate the middle and last term of the speedup expression. These last two terms, at the bandwidths we considered, make Vegas about 3-8% faster than Reno. Alternatively, changing the granularity of the Reno coarse timer from 500ms to 200ms (which doesn't require cooperating receivers) would reduce the savings that Vegas obtains by decreasing the frequency of timeouts.

Random Early Drop (RED) gateways [12] are another approach to TCP congestion avoidance. While implementing selective acknowledgments requires changing TCP receivers, RED gateways require changing network switches. The question of how RED gateways treat competing Reno and Vegas traffic deserves attention. Reno, since it tends to occupy more switch buffers than Vegas, may be punished harsher.

Since Vegas requires changes to the TCP sender only, individuals can easily deploy it. Ubiquitously deploying RED gateways and/or selective acknowledgment TCP have a bit of inertia to overcome.

6.4 Summary

This paper reproduced the claims about TCP Vegas made in [2]. Skepticism about Vegas is partially based on early descriptions of Vegas that were posted to the end-to-end e-mail list.

At todays Internet bandwidths, Vegas offers improved throughput of at least 3-8% over Reno while reducing packet losses and subsequent retransmitted segments by a factor of 2 to 5. Vegas is more resilient to different receiver acknowledgment strategies than are Reno and Tahoe. Vegas speedup improves with path buffer capacity and hop count. Vegas remains more efficient than Reno for all the workloads we studied. While this paper did not study Vegas on gigabit networks, we suspect that it will still outperform Reno. However the interaction between Vegas and other congestion avoidance techniques deserves study.

Acknowledgments

We would like to thank Lawrence Brakmo and Larry Peterson for making Vegas 0.8 available to us and for their feedback on our work. We also thank Van Jacobson who made his prototype Reno receiver available to us for our live experiments. We would especially like to think Sally Floyd for her valuable comments and encouragement, as well as Jim Kurose for his comments and encouragement. Finally, we would like to thank two anonymous referees for their brutal but perspicacious comments.

References

[1] Lawrence Brakmo. TCP Vegas Release 0.8, November 15, 1994. ftp://ftp.cs.arizona.edu/xkernel/new-protocols/Vegas.Tar.Z.

[2] Lawrence S. Brakmo, Sean W. O'Malley, and Larry. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM '94*, pages 24–35, May, 1994. ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.ps.

[3] Peter B. Danzig and Sugih Jamin. tcplib: A library of TCP/IP traffic characteristics. *USC Networking and Distributed Systems Laboratory TR CS-SYS-91-01*, October, 1991. ftp://catarina.usc.edu/pub/jamin/tcplib.

[4] Sally Floyd. TCP and successive fast retransmits. ftp://ftp.ee.lbl.gov/papers/fastretrans.ps, February 24, 1995.

[5] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM 88*, pages 273–288, 1988.

[6] Van Jacobson. Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno. *Proceedings of the British Columbia Internet Engineering Task Force*, July 1990.

[7] Raj Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *Computer Communication Review*, 19(5):56–71, October 1989.

[8] Srinivasan Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM '91*, pages 3–15, Zürich, Switzerland, September 1991. ACM.

[9] Aleta Lapone, Nicholas Maxemchuk, and Henning Schulzrinne. The Bell Laboratories Network Emulator. Technical memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, September 1993.

[10] Netbsd 1.0 operating system source distribution. ftp://gatekeeper.dec.com/pub/BSD/NetBSD. See http://excalibur.usc.edu/vegas/netbsd-patch for a necessary patch to the window resizing algorithm.

[11] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison Wesley, 1994.

[12] Curtis Villamizar and Cheng Song. High performance TCP in ANSNET. *Computer Communication Review*, 24(5):45–61, October 1995.

[13] Zheng Wang and Jon Crowcroft. A new congestion control scheme: Slow start and search (Tri-S). *Computer Communication Review*, 21(1):32–43, January 1991.