An Evaluation of TCP Vegas by Live Emulation

Peter B. Danzig, Zhen Liu, Limin Yan
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{danzig, zhenliu, lyan}@catarina.usc.edu
Office: (213) 740-4780 Fax: (213) 740-7285
USC Computer Science Technical Report 94-588

Abstract

Since it was published in 1988, no one has implemented, deployed, and published a successful enhancement to Van Jacobson’s Slow-Start flow control algorithm [6] for wide-area TCP networks that is backwards compatible with existing implementations. No one until Brakmo, O’Malley, and Peterson [2] who claim their TCP Vegas modifications yield both simultaneous throughput improvement and increased network efficiency. This paper presents evidence that the Vegas performance enhancements to TCP flow control proposed do not always simultaneously yield both increased throughput and decreased load on the network. We arrived at this conclusion by conducting experiments on an emulated wide-area network that we constructed. This paper reviews the Vegas algorithm, describes our network emulator, and contrasts our experimental results with those of Brakmo et al.

Researchers frequently simulate rather than conduct live experiments of new or “tuned” wide-area network flow and congestion control algorithms sometimes because they are experimenting with service disciplines not yet implemented in deployed networks [20, 9, 10], because some practical aspects of the algorithm have yet to be worked out [18], and because live experiments must be conducted carefully to obtain statistically significant comparisons [2]. This raises a quandary: neither packet-by-packet simulation nor live experiments can convince practitioners to adopt a potentially less stable flow control algorithm. What is needed is a wide-area network in which algorithms can be rigorously tested. This section describes our wide-area emulator environment, a proving ground for algorithms that have graduated from the simulator, but are not yet ready to be widely deployed. The emulator is simply a collection of workstations, each outfitted with 2 to 6 Ethernet interfaces, and each outfitted with a small operating system patch. We configure the emulated network into topologies comparable to the ones that appear in recent flow and congestion control studies, including the Vegas study [2].

1 Introduction

Since it was published in 1988, no one has implemented, deployed, and published a successful enhancement to Van Jacobson’s Slow-Start flow control algorithm [6] for wide-area TCP networks that is backwards compatible with existing implementations. No one until Brakmo, O’Malley, and Peterson [2] who claim their TCP Vegas modifications yield both simultaneous throughput improvement and increased network efficiency. This paper presents evidence that the Vegas performance enhancements to TCP flow control do not always simultaneously yield both increased throughput and decreased load on the network. We arrived at this conclusion by independently implementing the key Vegas innovations inside SunOS 4.1.3 and then contrasting its performance with the native SunOS TCP implementation on an emulated wide-area network that we constructed. This paper reviews the Vegas algorithm and our implementation of it, describes our network emulator, and contrasts our experimental results with those of Brakmo et al.

2 A WAN Emulator from Workstations

The operating system patch that implements the wide-area network emulator can configure the bandwidth, output buffer size, propagation delay, and bit error rate of each point-to-point Ethernet link. This permits us to emulate network hardware of different link bandwidths which results in different transmission delay and queuing delay of data. After assigning a unique IP addresses to each interface, plugging in the point-to-point Ethernet cables, configuring the routing tables of each workstation, and assigning the link bandwidth, propagation
delay, bit error rate, and output-buffer size, the emulator is ready for live experiments.

Figure 1 illustrates one emulated topology that we used in our Vegas experiments. 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. We studied various propagation delays, link bandwidths, and link output-buffer sizes between hosts. For example, if we set the link between condesa and alameda to have 25 ms propagation delay and 200 kb/s bandwidth, and the link between alameda and montara to 10 ms propagation delay and 56 Kb/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 kb/s link, through a limited buffer, to a 200kb/s link.

The algorithm for the emulator is given in Figure 2. Before releasing packets onto the network, we intercept them and put them on a queue. For the given assigned link bandwidth, we calculate the packet’s transmission delay, which is the number of bits in the packet divided by the link bandwidth. We look up the given propagation delay assigned to the link, add it to the transmission delay, and call this sum the “network delay”. We keep the packet enqueued for the queuing delay of any packets enqueued ahead of it plus the network delay. At the appropriate time, we forward the packet onto the link.

For a given link \( l \), denote the transmission delay of a packet of size \( p \) bits by \( \Delta_t(p) \), the link bandwidth by \( \lambda_l \), the link propagation delay by \( \delta_l \), 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 \( T_{n-1} \). Then the TotalDelay for which a packet is held in queue can be calculated as:

\[
Q_n = \{ Q_{n-1} - (T_n - T_{n-1}) \}^+ \\
\Delta_t(p) = p/\lambda_l \\
TotalDelay = \delta_l + \Delta_t(p) + Q_n
\]  

(1)

A packet is dropped if, at its arrival time, it does not fit in the switch output-buffer. Notice that this is not the same as the size of the holding queue, since some packets in the link holding queue emulate link propagation delay and do not logically reside in the switch. To simulate unrecoverable link bit-errors, packets on the queue can be dropped corresponding to the assigned link error rate; typically we set this rate to zero.

Below we describe this algorithm's implementation in the SunOS 4.1.3 kernel and present our verification of it.

<table>
<thead>
<tr>
<th>Destination Address</th>
<th>Propagation Delay</th>
<th>Bandwidth</th>
<th>Drop Rate</th>
<th>Switch Buffer Size</th>
</tr>
</thead>
</table>

Table 1: Each point-to-point link can be configured as desired

### 2.1 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 original author, we still call our emulator code hitbox.

Hitbox is added to the operating system kernel as a named pseudo-device, which gives programs a handle by which they can change the emulation parameters listed in Table 1.

In the workstation’s packet transmission processing, 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. At the appropriate time, hitbox calls the true 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 specify a pair of unidirectional hitboxes, one at each host. This way, both outgoing and incoming packets see the same link bandwidth, propagation delay, and output buffer sizes. 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 destination 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. Otherwise, it undergoes the following processing.

When a packet arrives, the time for which it should be held by hitbox is computed as the sum of the link propagation delay, packet transmission delay, and switch queuing delay, as described above. Each arriving packets is labeled with the actual send-time at which it
should be forwarded to the physical network interface, which is the sum of the time the packet has arrived at the queue plus its delay. We schedule this packet for transmission after the computed delay occurs using the timeout mechanism common to BSD UNIX systems. Once the timeout occurs, the scheduled packet along with any other enqueued packets with send-time less or equal to the current time are forwarded to the network interfaces.

Unmodified, the BSD scheduler offers 10 ms kernel scheduling resolution. Since link delay and bandwidth emulation requires sub-millisecond scheduling precision, we had to modify these routines. In BSD-styled UNIX, the hardware clock interrupt schedules periodic software clock interrupts; most BSD-systems ignore the presence of any high resolution interval timers. Kernel events are handled by the soft clock, which invokes any clock interrupt handlers that have been installed [13]. In our implementation, we simply forced the clock to interrupt every 1000 microseconds, enduring the incidental overhead. Since clock and timeout scheduling is used extensively within the kernel, ten times more precision incurs ten times more context switches. Workstation performance degraded by 3%, determined by the elapsed time to quicksort 400,000 integers.

2.2 Verification of the Emulator

We verified the correctness of the emulator carefully, using “ping” to verify link propagation delay emulation, FTP experiments to verify the accuracy of link bandwidth emulation, and packet-by-packet tracing to verify that link buffer overflow and packet scheduling occurred accurately. We observed packet scheduling precision to within a millisecond and that obtainable bandwidth through the switch was accurate to within 1% of the declared link bandwidth. We dedicated a workstation to capture packets simultaneously from both the incoming and outgoing Ethernet of one of our emulated switches and showed that switch buffer overflow and packet scheduling occurred accurately under load. Figure 3 illustrates one view of this data, plotting TCP sequence number versus total input-to-output delay for an emulated 10 ms propagation delay, 56Kb/s bandwidth link with 15KB buffer. Since a 15KB buffer corresponds to 120 Kbits, a full buffer takes 150 ms to empty, which is what Figure 3 shows.

2.3 Other Emulators

We understand that a group at AT&T started to build a wide-area network emulator [12] out of Transputers, although the emulator was not completed. Beyond this project, we know of no other software implementation of a wide-area network emulator nor evaluation of flow control algorithms by emulation in the literature. As we will show in the next section, the emulator gives you a testbed for evaluating flow and switch scheduling algorithms that can reveal behavior not easily seen in simulation or in live, wide-area experiments. Neuman used our emulator to study his Prospero Resource Manager over a wide-area network [16].

3 Summary of TCP Vegas

Much recent research [7, 18, 18, 11] considers how flow control algorithms can avoid congestion rather than recover from it. In their 1994 SIGCOMM paper, Brakmo et al. announced five refinements to Jacobson’s widely adopted slow-start TCP flow and congestion control algorithm [6]. They named this algorithm TCP Vegas [2]
/* random packet drops */
float error_rate;

/* queue for output buffering */
Queue Fifo[MAXIMUM_FIFO_SIZE];

/* queue that emulates the physical wire */
Queue packet_fifo[OTHER_Q_SIZE];

ifhitbox_output()
{
  /* Implement Bernoulli Packet Loss */
  if (uniform_random_variate() < error_rate) {
    drop the packet;
    return;
  }

  /* Fifo holds packets scheduled for 
   * transmission onto the physical wire 
   * it is the output buffer */
  While (Fifo[queue_tail].send_time <= current_time) {
    Fifo_size--; 
    queue_tail++; 
  }

  /* add current packet size onto the Fifo */
  Fifo_size++; 
  if (Fifo_size < MAXIMUM_FIFO_SIZE) {
    drop the packet;
    Fifo_size--; 
    return;
  }

  /* Compute when to xmt packet on link. This 
   * is essentially the sum of the packet’s 
   * arrival time at this routine plus its queuing 
   * delay plus its transmission delay. The 
   * departure_time determines the amount of time 
   * a packet ought to stay on Fifo. When this 
   * departure_time expires, buffer space for this 
   * packet can be reclaimed since the packet is 
   * already sent. */
  compute departure_time;

  /* The packet_fifo queue emulates the physical 
   * wire. A packet stays on this queue for the 
   * sum of the packet’s transmission delay, queuing 
   * delay, and its propagation delay on the wire. 
   * This sum is effectively the time for a packet 
   * to reach its destination. Note that the 
   * packet_fifo queue operates independently from 
   * the Fifo queue above. */
}

/* When does packet arrive downstream */
compute arrival_time;

Fifo[queue_head++] = departure_time;
packet_fifo[packet_head].arrival_time = arrival_time;
packet_fifo[packet_head++].packet = ptr to the packet;

/* We use BSD’s timeout mechanism to schedule packet 
   * transmission. The timeout handler is called 
   * after the desired number of ticks expires. 
   * Routine ifhit_timeout() xmts the scheduled 
   * packet and other ready packets in the 
   * packet_fifo queue. Here, we initialize a 
   * single timeout for the first packet in the 
   * packet_fifo queue. */

  if (ifhit_timeoutScheduled = FALSE) {
    ifhit_timeoutScheduled = TRUE;
    schedule ifhit_timeout;
  }
}

ifhit_timeout()
{
  while (packet_fifo[packet_tail].arrival_time <= 
          current_time) {
    forward packet_fifo[packet_tail++].packet 
          to the link layer;
  }

  /* Schedule a timeout event in time 
    * packet_fifo[queue_head].arrival_time - 
    * current_time. We only schedule the lead 
    * packet in the packet_fifo queue. As each 
    * timeout occurs, we schedule the next. */
  if (any packet in packet_fifo)
    schedule ifhit_timeout;

Figure 2: Pseudo-code for the emulator
3.1 TCP Vegas - the Motivation

TCP Vegas employs five techniques to increase throughput and decrease packet losses due to congestion. The first two techniques increase the accuracy of the TCP retransmission timer, making it possible to detect lost segments several hundred milliseconds earlier than traditional implementations. The third technique adjusts the TCP window resizing algorithms to work effectively given Vegas’s more sensitive packet loss detection mechanism. The fourth technique reduces back-to-back packet bursts, and the last attempts to avoid congestion in a manner similar to Wang’s Tri-S Scheme [18].

Accurate RTT Calculation: Reno keeps its round-trip timer in units of 500 milliseconds and re-calculates the timer whenever an acknowledgement, denoted ACK, is received. For perspective, the current Internet round-trip time (RTT) between USC and MIT is roughly 100 milliseconds. This coarse-grained timer makes Reno rather insensitive to minor RTT fluctuations, but consequently Reno can take half a second or more to detect segment loss.

In addition to Reno’s coarse round-trip timer, Vegas manages a RTT estimate in millisecond resolution, and it achieves this by keeping a timestamp for each outstanding segment. When Vegas sends a segment, it reads the microsecond resolution system clock and records this value and segment sequence number in a timestamp field associated with each outgoing segment. Vegas keeps the timestamps of all outstanding segment in a queue for future retrieval. When an ACK is received, Vegas retrieves the timestamp of the segment being acknowledged, according to the segment sequence number in the timestamp. Vegas then reads the send time of this segment, compares it to the current system clock, and calculates the segments’ round trip time in millisecond resolution. Vegas also records the number of times a segment is retransmitted in its timestamp so that it can ignore retransmitted segments in its RTT calculation.

In addition to Reno’s coarse round-trip timer, Vegas maintains four of its own round-trip time values in a Vegas-specific data structure, although it updates these values with the familiar exponential smoothing algorithm used in Reno [6]. The most important of these is the Vegas current retransmission timeout value.

Earlier Segment Loss Detection: Reno retransmits a segment when its coarse-grained timer expires or when it receives three duplicate ACKs. When Reno detects segment loss, it invokes the slow-start algorithm.

In contrast, whenever Vegas receives a duplicate ACK, it checks its timestamp queue. If any of the unacknowledged segments in the queue have exceeded the current RTT – determined by the send time recorded in the timestamp – Vegas retransmits these segments immediately. If Vegas receives 3 duplicate ACKs, it reverts to Reno-style slow-start. After a retransmission, Vegas keeps an eye on the first two non-duplicated ACKs, each time checking for expired segments as in the case of duplicate ACKs and retransmits the expired segments immediately. This tries to patch additional holes in the receiver sequence number space, without waiting for the duplicate ACKs. Note that Vegas keeps all of Reno’s timers as a fall back mechanism, in case Vegas’s retransmission fails to patch the missing segments, which may occur if the network is severely congested.

Since Vegas can determine the current RTT in millisecond resolution, it can detect segment loss hundreds of milliseconds earlier than Reno, which has to wait for the coarse-grained timer to expire or to receive three duplicated ACKs.

Modified window sizing algorithm: Recall that
Reno executes the slow-start algorithm whenever it detects a segment loss; it halves its send window (actually half current send window plus 3 maximum segment size) and then linearly increases the window size with each acknowledgement. In contrast, Vegas only halves its send window when the lost segment is sent \textit{AFTER} the last decrease. The fact that losses happened before the last window decrease does not imply that the network is congested for the \textit{current} congestion window size, and therefore, does not imply that it should be decreased again \cite{2}. For example, consider the following scenario: For some reason Vegas retransmits segment A at some arbitrary time 10. If there are no prior losses at this congestion window size, Vegas halves its current send window. At time 15, suppose Vegas discovers that segment B must be retransmitted and that Vegas originally sent segment B at time 5. Vegas retransmits segment B, but does \textit{NOT} halve its send window because B may have been lost because the window at time 5 was too large, but Vegas has no proof that the current window size is too large.

**Spike Suppression:** Although the slowstart congestion window limits the number of bytes sent during one RTT, it doesn’t control spacing between the packets sending these bytes. In practice, it has been observed that ACKs frequently arrive bunched together in a process dubbed ACK compression \cite{15}. ACK compression causes back-to-back packet “spikes” in Reno’s sending process. Vegas tries to enforce uniform spacing between the packets transmitted at a particular window-size. Essentially, Vegas limits the peak rate that packets can be sent.

**Congestion Avoidance:** As mentioned above, much current research \cite{7, 19, 18} addresses congestion avoidance. Jain’s CARD (Congestion Avoidance using Round-trip Delay) scheme \cite{7} suggests using throughput over round-trip delay curve, which Jain denotes Power, to forecast congestion. When the power curve approaches its peak, it indicates the network is approaching the congestion state.

Wang’s Tri-S scheme \cite{18} monitors the derivative of the sending rate with respect to send window size. Every RTT, they increase the send window size by one, and monitor the changes in throughput. If the change is less than half of its value when the window size changes from one to two, then network is running into the congestion.

Keshav’s Packet-Pair scheme \cite{11} sends two back-to-back packets to the receiver, and observe the delay between their corresponding ACKs, then estimates the available path bandwidth.

Vegas’s \textit{active} approach to congestion avoidance is similar to Wang’s Tri-S scheme. Vegas compares the measured throughput with its expected throughput and controls the amount of \textit{extra} data injected into the network. The expected throughput is usually the throughput measured at the beginning of the conversation.

### 3.2 Three-Fifths Vegas

When the authors of Vegas made their source code available to us, they did not include these last two techniques. Since their earlier report, written before these last two techniques were fully implemented, makes roughly the same claims as does their SIGCOMM’94 paper, it is possible that much of Vegas’s claimed performance enhancement is due to the first three techniques \footnote{When the authors of Vegas make their full implementation available to us, we will repeat these experiments}. Below, we describe how we implemented these in SunOS.

### 3.3 Porting TCP Vegas to SunOS

We ported the Vegas concepts, based on Larry Brakmo’s X-kernel implementation of TCP Vegas dated from October 1993, to the SunOS 4.1.3 TCP protocol stack\footnote{In the following, we assume that the reader is fluent in TCP/IP protocols suite and its implementation as described in 4.3BSD UNIX: The book “The Design and Implementation of 4.3BSD UNIX Operating System” \cite{13} is an excellent reference on this subject.}. The SunOS stack implements TCP “Tahoe” rather than “Reno”. Tahoe differs from Reno in its response to three duplicate acknowledgements. Tahoe TCP undergoes slow-starts when a packet loss is detected either when it receives three duplicate acks or upon timeout. In contrast, when Reno detects a packet loss from three duplicate acks, instead of slow-starting it effectively reduces the congestion window by half\footnote{We thank Sally Floyd for clarifying the distinction between Tahoe and Reno}.

The implementation of Vegas is straightforward and leaves most of the SunOS code untouched. So that we could switch between Vegas and the original Tahoe SunOS code, and the Reno TCP code, we made TCP Vegas and TCP Reno socket options that can be toggled on or off by a \texttt{setsockopt()} system call. We created a \textit{Vegas control block} data structure – separate from the existing TCP control block – to hold each conversation’s Vegas specific items. We also created a statistic data structure to hold Vegas and SunOS conversation instrumentation.

The key to implementing the Vegas retransmission policy is its roundtrip timestamp queue. Each element in this queue is a structure that associates a timestamp with an outgoing segment:

```c
struct vegas_stamp {
```

\footnotetext[3]{When the authors of Vegas make their full implementation available to us, we will repeat these experiments}

\footnotetext[4]{In the following, we assume that the reader is fluent in TCP/IP protocols suites and its implementation as described in 4.3BSD UNIX: The book “The Design and Implementation of 4.3BSD UNIX Operating System” \cite{13} is an excellent reference on this subject.}

\footnotetext[5]{We thank Sally Floyd for clarifying the distinction between Tahoe and Reno}
int segment_length;
int segment_send_time;
int retransmission_count;
struct vegas_stamp *next;
);

To reduce the overhead of allocating kernel memory
for each timestamp, We allocate to each conversation a 4
KB page of kernel memory, sliced into 256 16-byte Vegas
timestamps. We wrote our own routines to allocate and
free these timestamps.

Vegas calls \texttt{vegas\_send()} when it sends a segment.
\texttt{vegas\_send()} allocates a new Vegas timestamp and fills
in the current time, length of the segment, and the num-
ber of times it has been transmitted. \texttt{vegas\_send()} keeps
these timestamps on a linked-list, in order of increasing
segment sequence number. This organization optimizes
the retrieval process, since TCP acknowledgement are
cumulative.

When Vegas receives an ACK, it calls \texttt{vegas\_receive()},
this routine takes the sequence number ACKed, retrieves
the corresponding timestamps, finds out the send time
and number of transmissions of this segment, and com-
putes the current RTT in millisecond resolution. Ve-
gas keeps a separate set of round-trip time values be-
sides the original TCP timers. The Vegas round-trip
time rtt, smoothed round-trip time srtt, round-trip time
variance rttvar, and the current retransmission timeout
value rttimeout are all kept in the \texttt{Vegas control block}.
Vegas updates its timers with the identical smoothing al-
gorithm used in Tahoe and Reno TCP.

The \texttt{tcp\_fast\_timd()} routine, which UNIX invokes ev-
ever 200 milliseconds, calls \texttt{vegas\_expire()} with the cur-
rent time and conversation RTT as parameters. Ve-
gas\_expire() searches the conversation's Vegas times-
tamp queue looking for \texttt{expired} segments, segments that
are older than the current RTT, and returns the size of
the \texttt{expired} segment. Vegas then retransmits these
\texttt{expired} segments according to its retransmission policy.

The Vegas timestamp queue routines are implemented in
\texttt{inet/tcp\_vegas.h}, and \texttt{inet/tcp\_vegas.c}. Most of
the changes to the original SunOS code occur in \texttt{inet-}
\texttt{tcp\_timer.c} and \texttt{inet/tcp\_input.c}. We have at-
ttempted to preserve the original X-kernel Vegas code
as much as possible in our port, to minimize the chance
of misrepresenting Vegas.

The original SunOS 4.1.3 TCP code keeps TCP statis-
tics on a per protocol basis, rather than a per-conversation
basis. We re-instrumented the TCP protocol stack to
record conversation-level details and placed these in a
conversation-specific \texttt{tcpstat} data structure which
created.

\begin{verbatim}
struct connstat {
    u_long n_rexmt_timeouts;
    u_long n_packets_rxmt;
    u_long n_data_bytes_rxmt;
    u_long n_packets_rxtcur;
    u_long n_bytes_rxtcur;
    u_long n_dup_acks;
}
\end{verbatim}

During our experiments, before closing a connection
we read and record its connection statistics.

\section{Experiments with Vegas}

This section summarizes several emulated and live ex-
periments that we conducted to compare Vegas and
Tahoe TCP. We intended to compare Vegas with Reno,
but we were caught by surprise that SunOS implements
Tahoe, not Reno. We plan on comparing all three TCP
variants soon. We focus on their bulk transport be-
havior under uncongested and highly congested condi-
tions. For bulk transfer flow control algorithms, the
two metrics best capture Vegas' performance are (1)
the total number of bytes transmitted from the start
to the end of a particular TCP conversation and (2) the
bandwidth achieved by this conversation. The bytes
transmitted figure includes bytes retransmitted while
the achieved bandwidth measure excludes retransmit-
ted bytes. Clearly, the higher the achieved bandwidth
and the lower the total bytes transmitted, the more ef-
ficient the flow control algorithm. An important third
metric, the fairness with which network bandwidth is
distributed, can be determined by the first two metrics.

Several years ago, we wrote that flow control algo-
rithms should tested under average case as well as worse
case load \cite{4}, and we released a TCP workload \cite{3} rep-resentative of wide-area network traffic. Brakmo's average
case workload simulations were, in fact, driven with our
traffic workload library. He did not, however, publish
simulation results of heavily congested networks. For
this reason, we present our studies of lightly loaded net-
works first.

\subsection{Live Internet Experiments}

We started by replicating Brakmo's live Internet ex-
periments \cite{2}. For 50 repetitions, we FTP'd a 512 Kbyte
file cross country from USC in Los Angeles to NYU in
New York, alternating between our Vegas and Tahoe im-
plementations. Since these were live experiments, each
repetition yielded different bandwidth and transmitted
a different number of total bytes. Our 50 repetitions
Figure 4: Live internet experiment late at night

Figure 5: Live internet experiment in the afternoon

were a sufficient number to obtain non-overlapping confidence intervals [8].

Figure 4 and Figure 5 histograms the obtained bandwidth and total bytes transmitted for the 50 repetitions. Solid ticks indicate a Vegas experiment and dashed lines indicate Tahoe.

We could not replicate Brakmo's claim of a 40 - 70% improvement in throughput. As Figure 5 shows, Vegas yields 18% higher bandwidth than Tahoe at 3:00 PM California time, not a bad result, but yields 8% lower throughput at 3:00 AM, California time. In both cases, Vegas transmitted a few percent more bytes than Tahoe did, a statistic not reported by Brakmo. We cannot tell from live experiments whether Vegas can yield higher bandwidths than Tahoe because, as a flow control algorithm, it is being greedy or because it is a better flow algorithm. For this, we turn to the emulator.

4.2 Emulation Experiment: Uncongested Network

We emulated the wide-area network detailed in Figure 1 where the balboa-condesa and condesa-alameda links were configured at 200 kbps bandwidth with 25 millisecond one-way delay, and the links between hosts jalama, paloma, roqueta, montara and their respective routers were configured at 10Mbps bandwidth and 1 millisecond one-way delay. We set the buffer size on routers balboa, condesa, and alameda to 16 kbytes. We chose this buffer size because, at 200 kbps, this holds 0.64 seconds worth of traffic, which we considered a reasonable amount.

In our first experiment a single TCP conversation sends 512 kbytes of data from jalama to roqueta, several hops away. We tested 25 conversations each of Vegas and 25 of Tahoe and histogram the results. Figure 6 shows that Vegas yielded 12% average lower throughput and transmitted 1.7% more bytes than Tahoe.

In an attempt to duplicate performance similar to Brakmo's experiment, we configured the balboa-condesa and condesa-alameda links at a rather high 2% packet loss fraction. Note that this does not model buffer overflow losses, but would simply model an independently distributed, Bernoulli packet loss. Under this condition, Vegas yielded 58% higher bandwidth than Tahoe, while transmitted 4.5% more bytes.

The difference in Vegas performance in the lossless and lossful network leads us to conjecture that the difference in Vegas's daytime and nighttime performance is due to high buffer overflow rate at Internet routers. Perhaps traffic not subject to flow control, such as the Internet MBONE, is causing a statically high rate of router buffer overflow. Perhaps one of Arizona's routers suffers a high loss rate?

4.3 Emulation Experiment: congested network

A flow control algorithm that performs great under light load must also be stable under heavy load, or the network could suffer congestion collapse. We conducted a
small experiment to see if Vegas remained stable and performed well under heavy load. We started 512KB concurrent TCP transfers, one every 3 seconds, until we had 10 concurrent sessions. Since the duration of each session is around 200 seconds, the 10 sessions remained mostly concurrent. Under this load, Vegas throughput was 3% lower than Tahoe’s, and it transmitted 3% more bytes than Tahoe.

We then added cross traffic between paloma and montara and conducted various experiments, some with tcplib traffic and some with ten concurrent 1MB TCP transfers. We focus on this latter cross traffic. Figure 7 records the result of experiment with Tahoe cross traffic, and Figure 8 records the result with Vegas cross traffic.

These two figures demonstrate that Vegas senders are more aggressive than Tahoe senders and therefore obtain more of the available network bandwidth. With Tahoe cross traffic, observe that Vegas senders obtain 50% higher throughput than Tahoe senders. With Vegas cross traffic, observe that Vegas senders obtain 26% higher throughput than Tahoe senders. Note that in these two cases, Vegas senders transmit 13% and 20% more bytes than their Tahoe counterparts. At first glance, this looks great for Vegas. However, these numbers also reveal that Tahoe senders lost 32% of their throughput when we switched from Tahoe to Vegas cross traffic. Vegas gathers an unfair share of network bandwidth.

### 4.4 Serendipity

Quite accidentally, we conducted a set of emulation experiments in which Vegas performed quite badly. In these experiments, all links on the emulated network operated at their full 10 MB/s Ethernet capacity, but queueing and propagation delay estimates were rounded to the nearest 10 millisecond boundary. Consequently, within any 10 millisecond interval, a periodic train of evenly spaced packets would get forwarded as a back-to-back burst followed by an idle period up to the next 10 millisecond interval. While this was the result of mis-configuring the emulator kernel, it does illustrate a particular workload that Tahoe handled much better than Vegas.

Figure 9 contrasts the throughput and total bytes transmitted by 5 replicas of the experiment for Tahoe sources and for Vegas sources. In this experiment, 10 sources competed to send 2 megabytes of data across the network. At time zero, one source starts and every second another source starts up until all ten transmit simultaneously.

Examination of Figure 9 shows that Vegas sources transmit 4-6% more bytes than Tahoe sources yet achieve only 2/3 the throughput of Tahoe sources.

One can argue that the emulator’s switching discipline was highly unorthodox and undesirable. However, since burstiness in local and wide-area traffic is only roughly understood [17, 14] we believe that flow and congestion control algorithms must perform well under stress even if its clearly superior under light loads.

The behavior of the our serendipity network closely resembles a network switch in which best-effort TC P has
to share the network with isochronous, higher-priority realtime traffic. In this case, the switch can only forward best-effort traffic when no real-time traffic is enqueued. In such a case, the TCP traffic will bunch together and will be transmitted in bursts. This experience cautions us of the dangers of using too fine-grained a clock as a retransmission timers without increasing the variance in round trip times tolerated before a retransmission is invoked.

5 Conclusions and Future Work

This study of TCP Vegas demonstrates the importance of subjecting flow and congestion control algorithms to controlled WAN emulation experiments, in addition to detailed simulation and live experiments. We say this because packet simulation of flow control algorithms depend crucially on the granularity of timing [1] and because in live experiments it is frequently difficult to determine how the new algorithm affects the performance of users of the old algorithm. This paper describes how to build such an emulator out of your existing workstations. Our emulator software and TCP Vegas implementation for SunOS are available 4.

Our emulator revealed that our version of TCP Vegas does not outperform the SunOS Tahoe implementation. We found that, under all conditions, that Vegas transmits 2-3% more bytes than Tahoe. This means that Vegas uses the network less efficiently than Tahoe and could be considered a cause of network congestion. We found that, under heavy load and under tcplib load, that Vegas sources compete unfairly with Tahoe sources. In direct contradiction to Brakmo's results, we find that under heavy congestion, Vegas provides lower throughput. Finally, we found that with lossy links, Vegas can provide higher throughput, but again transmits more bytes than Tahoe.

5.1 Live, Simulated, or Emulated?

You can develop a flow control algorithm that, under simulation, outperforms the old algorithm, but that doesn't outperform it when it is deployed. For example, Floyd and Jacobson demonstrated [5] that, when simulated, TCP demonstrated a stable unfairnesses that depends on round trip times that they dubbed a “phase-effect”. Their simulations showed that phase effects disappear completely when the network carries over 40% non-TCP traffic, but that they were otherwise present. However, no one has yet observed phase effects in a real network and probably you can't create phase effects in today's high speed real networks. Hence, if you modified TCP to avoid phase effects, your simulation benefits but not the real world.

5.2 Future Work

Since we wrote our Vegas implementation from the source code of the original X-kernel implementation, we are confident that it faithfully represents the original. However, since the authors of Vegas have not yet made their implementation of congestion avoidance and spike suppression available to us, we can only claim that if the complete Vegas implementation outperforms Tahoe, then it's due to its congestion avoidance and spike suppression algorithms. If our result hold up against further scrutiny, then we can claim that the first 3 ideas behind Vegas do not contribute to its performance or that Vegas requires all five algorithms in concert.

It is conceivable that the improved performance of Vegas on the X-kernel are due to an inferior implementation of Reno on the X-kernel. In cooperation with Brakmo and Peterson, in the coming months we hope to definitively answer these questions. In the mean time, our experience should serve to caution the Internet community against adopting TCP Vegas without the results of further live emulated experiments.

TCP Vegas is controversial; several email messages from the key participants are available5. We believe that our emulation study places the burden of proof back on the Vegas team.

References


4http://excalibur.usc.edu/research/vegas/doc/vegas.html

5http://excalibur.usc.edu/research/vegas/discussion


