On End-to-End Congestion Avoidance for TCP/IP

Jim Martin
IBM Corporation
PO Box 12195
RTP, NC, USA 27709
919 254 4447
jjm2 @ us.ibm.com

Arne Nilsson
North Carolina State University
Box 7914, Raleigh, NC 27695
Raleigh, NC 27695
919 515 5130
nilsson @ ncsu.edu

Abstract

A TCP/IP network utilizes several congestion control schemes: end-to-end flow control and congestion avoidance, gateway congestion control, and explicit closed-loop feedback (i.e., source quench). The evolution of TCP/IP includes enhanced gateway congestion control algorithms (i.e., Random Early Detect) and a variety of incremental improvements to TCP including selective acknowledgement and possibly end-to-end congestion avoidance (i.e., TCP/Vegas). We focus on end-to-end congestion avoidance algorithms for TCP, specifically those algorithms that use change in packet transit times as an indicator of network congestion. TCP/Vegas is the most well known algorithm based on this form of congestion control. We find that TCP/Vegas does increase throughput primarily by avoiding time-outs. However its assessment of congestion is prone to significant error which can lead to increased queue levels at the bottleneck link. By studying TCP/Vegas and other algorithms, our goal is to understand the issues associated with end-to-end congestion avoidance schemes that monitor change in packet delays.

This paper is organized as follows. First we introduce end-to-end congestion avoidance. Next, using simulation, we explore the various issues and challenges associated with end-to-end congestion avoidance by demonstrating and analyzing several end-to-end congestion avoidance algorithms. We conclude with a discussion of key issues associated with end-to-end congestion avoidance and identify future work.

Keywords

Congestion control, TCP, TCP/Vegas, congestion avoidance

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35388-3_42

H. R. van As (ed.), *High Performance Networking*
© IFIP International Federation for Information Processing 1998

1 INTRODUCTION

A congestion control scheme can be classified as either reactive or preventive (the latter is also known as congestion avoidance). Additionally, some control schemes require feedback while others do not. A reactive scheme inherently is closed-loop while preventive schemes can be either open or closed-loop. Open-loop control is inherently preventive, employing admittance control and/or traffic policing to prevent congestion from occurring. Closed-loop congestion avoidance, on the other hand, is designed to keep the network at the point of maximum power (i.e., the point where the ratio of throughput versus delay is highest).

The feedback in a closed-loop system is either implicit or explicit. Explicit feedback involves an explicit send of feedback information. Explicit feedback can be characterized by the location of the source of the feedback (i.e., referred to as the level of control), by the mechanism that transfers the feedback to the source and by the actual content of the feedback. Various forms of explicit feedback exist in the Internet today such as TCP's end-to-end flow control [9], source quench [11] and explicit RED [5,6,7].

Unlike explicit feedback, implicit feedback does not involve an explicit "send" or transmission of feedback signals. The implicit feedback (based on time-out or packet loss events) can be detected by either the sender or the receiver. For example, TCP's slow-start and congestion avoidance algorithms rely on packet loss as an implicit indication of network congestion [15]. When the source of the implicit indication is the network, the scheme is typically classified as a form of gateway congestion control.[1]

It is also possible to implement congestion avoidance at the endpoints based on implicit feedback such as changes in packet transit times. The most well known example of this class of congestion control is TCP/Vegas [1,2,6]. However there have been other proposals: Wang and Crowcraft's Tri-S and Dual algorithms [16,17], Haas's Adaptive Admission Congestion Control algorithm [8], and IBM's Adaptive Rate-based (ARB) protocol [12]. Of all of these algorithms, ARB is the most widely deployed as it is the congestion control scheme used in the latest release of SNA [14].

In this report, we study end-to-end congestion avoidance based on implicit feedback for TCP/IP networks. In particular, we focus on the effectiveness of

---

[1] TCP congestion avoidance based on a simple drop-tail router packet drop policy is usually considered to be end-to-end control. If the router participates more actively in congestion management, (e.g., RED or explicit congestion indications), then the router augments the base TCP end-to-end congestion avoidance algorithm with gateway congestion control.

three algorithms (TCP/Vegas, Dual and ARB) that use change in packet delay as an indication of the level of network congestion. The goal is to identify and explore the challenges associated with this form of congestion control. We conclude this paper with a discussion of key issues associated with end-to-end congestion avoidance and identify future work.

2 ANALYSIS OF END-TO-END CONGESTION AVOIDANCE

In this section, we study the congestion avoidance algorithms used by TCP/Vegas, Dual and IBM's ARB protocol. We feel that a study of these algorithms exposes the key issues. We are interested in two network environments: first, an environment where the protocol under observation competes only with similar connections; second, a best-effort IP network where the scheme under investigation must compete with any IP traffic (i.e., from other TCP or UDP connections). To help focus on congestion avoidance, we use TCP/Vegas as the base protocol and either exchange or integrate pieces of the other schemes into TCP/Vegas. This approach allows us to understand the tradeoffs of the different congestion detection schemes without clouding the discussion due to other protocol differences. Our simulation model is based on the ns simulation (v 1.4) which includes a TCP/Vegas model [4].

Figure 1 illustrates the network topology used in the simulations. Roughly half of the simulations use the simple LAN-WAN-LAN involving router's R2 and R3. In this case, the WAN is approximately T1 speed with a propagation delay of 50ms. The other simulations use the multi-hop environment provided by routers R1, R2, R3 and R4. In this case, the bottleneck link is the T1 hop. The WAN link is approximately T1 speed with a propagation delay of 50ms. All packets are 1400 bytes. We use a combination of bulk traffic source models (i.e., ftp traffic) and on/off bursty sources.

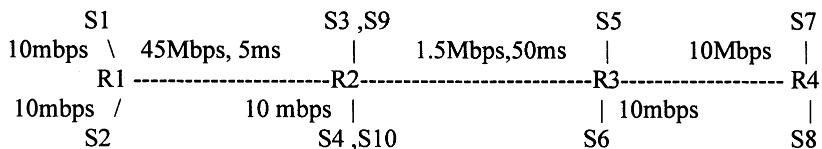

Figure 1

TCP-Vegas

The aspect of TCP/Vegas that is of interest to us is the Congestion Avoidance Mechanism (CAM). CAM monitors throughput, comparing a measured throughput with an expected throughput. The *Throughput_Diff* is the difference between an *Expected_Throughput* and an *Actual_Throughput*. The *Expected_Throughput* is the current window divided by the *BaseRTT*, where the latter is the minimum round trip time observed by the connection (which

should converge to the uncongested round trip time). As long as the *Expected_Throughput* is accurate, the *Throughput_Diff* * *BaseRTT* (the *Diff*) is an estimate for the amount of extra data that the connection has in transit. Vegas attempts to estimate the amount of extra data and maintain the "right" amount in the network. By keeping some amount of data queued in the network, Vegas hopes to keep network utilization high.

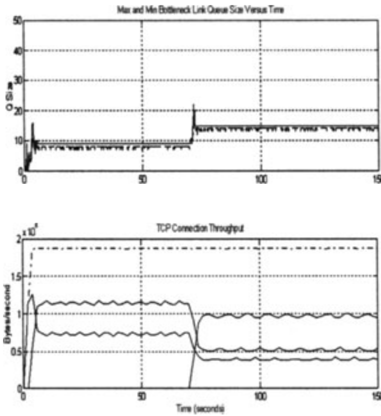

Figure 2

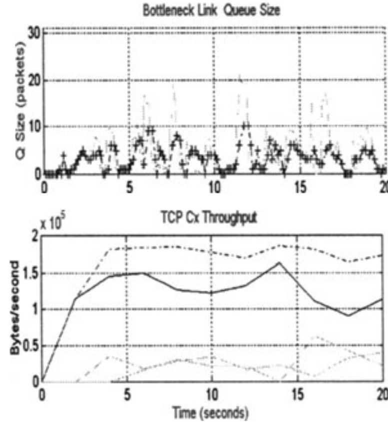

Figure 3

Figure 2 illustrates three bulk transfer Vegas connections competing over the T1 WAN hop illustrated in Figure 1. All three connections flow from R2 to R3. The top graph shows the queue level of R2's output queue. The solid line plots the maximum queue level sampled every .1 second time interval and the dashed line plots the minimum queue level during each sample period. The router queue depth is 50 packets ensuring that packet loss does not occur.

The three solid curves in the lower graph of Figure 2 plot the throughput (measured in bytes per second each 1 second) of each connection. The dashed-dotted line plots the utilization of the bottleneck link. Vegas clearly utilizes the available bandwidth, however there is a fairness problem. The first 2 connections start at time 0 and converge to uneven shares of available bandwidth. Once the third connection starts, it obtains a significantly larger share of bandwidth than the other connections. The behavior is the same if we add a random delay in the TCP sender in the range of [0,6ms][2]. Therefore, the problem is not a phase effect. Initially, we thought the problem might be caused by the "no change" state that Vegas tries to reach (i.e., in between the *alpha and beta* thresholds). We set the thresholds equal (i.e., set both *alpha*

[2] We modified the ns TCP/Vegas implementation such that timestamps are recorded before the send delay. This injects "noise" in the Vegas congestion assessment. The TCP sender is coded such that if it has multiple packets to send (i.e., if an Ack causes the *cwnd* to increase), the random send delay occurs once prior to the burst. After the delay expires, the sender is allowed to send all packets in the burst instantaneously.

and *beta* to a value of 3) to eliminate occurrences of a "no change" CAM decision. This led to similar results as in Figure 2.

There are actually two problems that explain the behavior observed in Figure 2. For static networks (i.e., a network that consists of constant Vegas senders that reach some steady state), it is likely that the system will converge to an unfair state. Once in this state, the increase/decrease algorithm is not sufficient to move the system to a fair state. The second problem is that once the system enters steady state, the existing connections have pushed the network such that there is some amount of sustained queueing. Late starting connections can not detect the congestion and instead will add to the existing congestion forcing the existing connections to reduce their send rates. Therefore, in a congested system that has reached steady state, late starting connections will obtain an unfairly large share of available bandwidth.

A more dynamic network environment will reduce the probability of the system entering a steady state. Figure 3 illustrates the same scenario as in Figure 2 except that connections 2 and 3 are configured to use an on/off bursty traffic source rather than an ftp source. The bursty sources have mean rates on the order of 100000 bps. The lower curve shows that the two bursty connections (the two light lines) obtain bandwidth from connection 1 (the dark line) in a fair manner. The dashed-dotted line illustrates that Vegas is able to utilize on the order of 90% of the available bandwidth. The upper plot of Figure 3 shows the minimum queue level at the bottleneck link (the light dashed line). The "+" marks represent the Vegas sampled queue level (i.e., the *Diff*).

In the lightly loaded network illustrated by Figure 3, CAM responds to congestion well although it does not track network congestion precisely. As described earlier, Vegas detects congestion by looking for changes in throughputs. The *Actual_Throughput* is based on the number of packets transmitted during the past RTT and on the actual RTT sample. The original Vegas proposal suggested that one packet each RTT be selected to probe for congestion. The Vegas implementation based on NetBSD actually uses the average RTT's for all packets (since Vegas times all packets) that were acknowledged during a measurement interval [1]. The idea being to filter noise associated with individual RTT samples.

In the best case, an end-to-end delay measurement algorithm tracks queueing at the bottleneck link caused by the aggregate traffic from all sources. In the worst case, the algorithm tracks queueing caused only by the connection itself. The following analysis shows that Vegas does not accurately track neither network level queueing nor queueing caused by the Vegas sender. In certain situations, the algorithm's congestion assessment effectively becomes meaningless as the *Diff* value converges to a fixed window value. The latter case explains previous analysis results of Vegas that conclude that during periods of heavy congestion, the scheme digresses to TCP/Reno behavior[2].

First, we show a simple example (a single Vegas connection with no competition) where CAM accurately tracks the queue level at the bottleneck

link. Using the network shown in Figure 2 as an example, assume that a single Vegas always has 1400 byte packets available to send. At the point where the T1 link is fully utilized, the Vegas *cwnd* is 13.4 packets (i.e., a bandwidth-delay product of 13.4 packets). The *expected* throughput is naturally 1.5Mbps and the *actual* throughput should be the same. The next RTT, the *cwnd* will be 14.4. The *expected* throughput will be just over 1612800 bps. Assuming exactly 1 packet experienced a waiting time of a packet transmission time at the T1 link, the *rtt* should be .1 + 1400*8/1.5Mbps or .1075 seconds. The *Actual_throughput* will therefore be 14.4 packets / .1075 seconds or roughly 1.5 Mbps. Multiplying the difference in throughput by the *BaseRTT* corresponds to a *Diff* in packets of roughly 1400 bytes or 1 packet.

When Vegas competes with a low to moderate amount of traffic, the following helps explain the behavior of the *Diff* samples as illustrated by the upper curve in Figure 3. The *Diff* value in bytes can be written as:

$$Diff = (W/BaseRtt - W'/Rtt) * BaseRtt$$

where W is the current window, W' is the amount of data sent during the measurement period, and *Rtt* is the current RTT sample. As long as the sender has data to send, no packets are lost and the receiver ACKs each packet, W' will be the current window, W. Therefore:

$$Diff = W - W * BaseRtt / Rtt$$
$$Diff = W(1 - BaseRtt / Rtt)$$

Clearly, *Diff* is 0 when $BaseRtt = Rtt$, and is positive when $Rtt > BaseRtt$. Also note that the $Rtt = BaseRtt + Qt$ where Qt reflects queueing delays. The upper bound of $(1\text{-}BaseRtt/Rtt)$ is 1 which means that the largest *Diff* value that can ever be observed is the current window size. The $(1\text{- } BaseRttt/Rtt)$ term essentially grows linearly with increasing *Rtt* , however the W will also decrease in response to a positive *Diff*. As the *Rtt* increases, the rate of increase of the *Diff* is dampened as W decreases. This explains the behavior of the *Diff* curve in Figure 4 As the queue builds, the term $(1\text{-}BaseRtt/Rtt)$ increases however the *Diff* value might actually decrease as W decreases in response to the congestion. The scheme is fair in the sense that a connection with a high bandwidth will react more aggressively to increases in *Rtt's*. However, as the following discussion will show, the scheme loses its effectiveness when operating in periods of heavy congestion.

Vegas does not decrease the window if $Diff <= beta$. The point where the algorithm stops decreasing the window is:

$Wmin\ (1 - BaseRtt/Rtt) = beta$ where $Wmin$ is the lowest window value that is to be used.

$$Wmin = beta / (1 - BaseRtt/Rtt)$$

For large *Rtt's* with respect to *BaseRtt*, the *Wmin* approaches *beta*. We will see that in heavily congested networks (i.e., networks where there is a large amount of sustained queueing), the Vegas algorithm effectively holds the sender to a fairly constant window of *beta* packets (3 for our simulations). We

refer to this as the overload state. Given this, the throughput of Vegas (in packets per second) in heavily congested networks can actually be predicted as follows:

$$Vegas_throughput = beta \ / \ (BaseRtt + (Qavg * MSS * 8) / Cbl)$$

where *Qavg* is the average queue length sampled each RTT, the *MSS* is the maximum segment size and the *Cbl* is the bottleneck link capacity.

Figure 4 illustrates the results of a simulation run based on the multi-hop network shown in Figure 1 with 3 ftp TCP/Vegas connections (S1, S2 and S3 all to R4) compete with 3 on/off bursty TCP/Vegas connections (S4,S9 and S10 to R3). The bottleneck router can buffer up to 50 packets. The simulation is intended to capture the behavior of a heavily congested network. The minimum and maximum queue levels of the T1 link shown in the upper plot (the dark nd the dotted line respectively) demonstrates that CAM is not able to prevent sustained congestion from occurring. The "+" marks plot the *Diff* samples from the first Vegas connection. The lower plot shows the three ftp Vegas connection's throughputs (we do not show the throughputs of the bursty connections, only the effective utilization). The throughput curves illustrate the bias towards late starting connections (connection 1 gets a lower share of available bandwidth once the system converges). If we start a fourth ftp connection at time 20 seconds, it would obtain a much larger share of bandwidth than any of the other connections.

The connections with accurate *BaseRTT* values (i.e., the two connections that start at time 0 and 2 respectively in Figure 4) converge to a window of about 4 packets with a sustained throughput close to the predicted throughput of a Vegas connection that has reached the overload state. Late starting connections might not reach the overload state since they will be much more tolerant of congestion (since the *BaseRTT* will include the sustained congestion levels). If too many Vegas connections are in the overload state at the same time, packet loss will occur and the behavior of each connection will digress to TCP/Reno (i.e., oscillating window values). Figure 5 shows a more extreme environment than that depicted in Figure 4. The router buffer capacity is reduced to 20 packets and we add two additional ftp Vegas connections (that start at time 0 and 2 seconds respectively). The packet loss rate is quite high, about 6.2%. Each Vegas connection experiences time-outs (in the range of 2 to 15) contributing to *cwnd* oscillations. This demonstrates the tendency for Vegas to digress to Reno behavior in high packet loss environments.

In the analysis presented so far, we have observed Vegas in an environment where it competes only with other Vegas connections. It is also interesting to see how Vegas behaves when competing against other TCP/Reno connections. Figure 6 illustrates a simulation with one Vegas connection (the first connection that starts at time 0) and two TCP/Reno connections that start at time 5 and 70 seconds respectively. All connections are configured with ftp traffic sources. The simulated network involves the T1 hop between R2 and R3 as illustrated in Figure 1. The lower curve shows synchronized behavior similar to that seen in Figure 2. The Vegas connection increases its throughput until time 2 seconds when the second connection starts. The

second connection (i.e., the Reno connection) reaches its maximum window (36 packets) causing additional sustained queueing that forces the Vegas connection to a state of low throughput. The system remains locked in this state until the second TCP/Reno connection forces packet loss after time 70. Note that after time 70 the system reaches a new synchronized state. However, since the queue levels stay consistently high, the Vegas connection is never able to obtain its fair share.

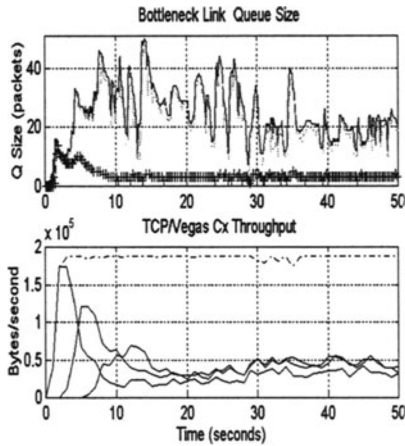

Figure 4

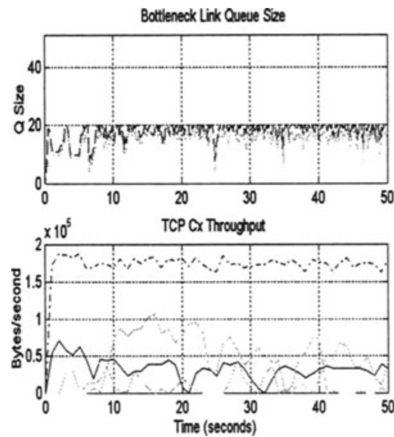

Figure 5

Figure 7 illustrates a more realistic simulation than the previous. Based on the network shown in Figure 1, a Vegas ftp connection from S1 to R4 starts at time 0. Two on/off bursty Reno sources flow from S3 and S4 to R3 that produce bursty cross traffic. The solid dark line in the lower curve illustrates the Vegas throughput and the lighter lines represent the throughput of the two on/off connections. The behavior is similar to the earlier Vegas case where we concluded that CAM is effective at tracking and controlling congestion in a mildy congested network. If we do the same simulation except use all Reno connections, the performance of the ftp connection is similar to the Vegas run. Both achieve roughly the same throughput, neither experience time-outs. One significant difference, however is that the queue levels are more controlled in the Vegas case than in the all Reno simulation.

Figure 8 illustrates how an ftp Vegas connection (the dark link in lower curve) competes with two ftp Reno connections and 3 bursty Reno. The behavior is as observed in the USC analysis where Reno steals bandwidth from Vegas in head-to-head competition simply because Vegas is more sensitive to congestion than Reno[1]. Confirming our earlier analysis, the Vegas connection is essentially limited to the *Wmin* of 3 which, assuming an average queue level of 35 packets (by inspection from Figure 8), should lead to a throughput of 93,000 bps. Based on the Vegas throughput curve we see a Vegas throughput of roughly 120,000 bps.

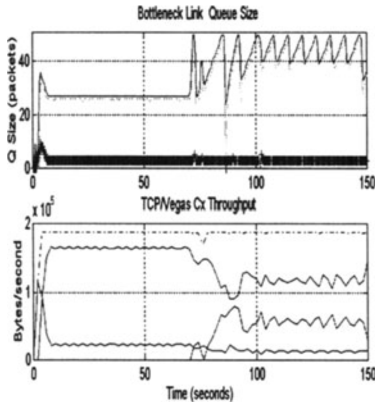

Figure 6

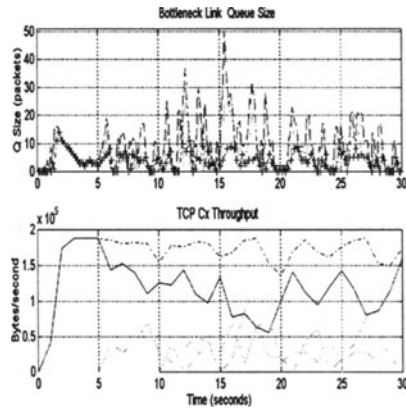

Figure 7

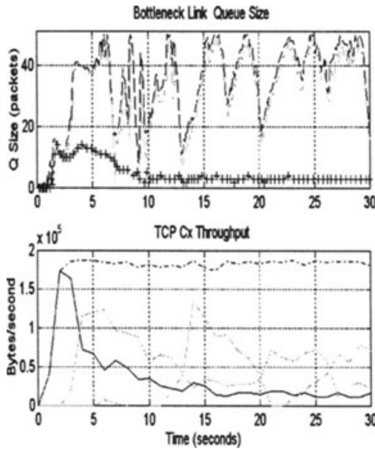

Figure 8

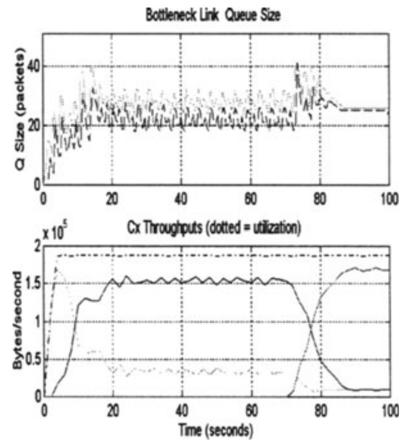

Figure 9

Dual

Dual offers a congestion detection scheme that monitors changes in measured round trip times. It tracks the minimum observed RTT (*RTTmin*) and the maximum observed RTT (*RTTmax*) over the lifetime of the connection. In most network environments, over time these variables converge to the static round trip delay due to propagation delays and to the variable delay representing queueing delay respectively. Each time TCP performs its round trip time calculation, the sampled RTT is compared with a delay threshold defined as:

$$Di = (1-alpha)Dmin + alpha * Dmax.$$

Dual was designed to extend TCP's congestion avoidance and slow start algorithms with the goal of reducing the oscillations caused by slow start and congestion avoidance. The change to TCP to implement Dual is trivial. Slow start and congestion avoidance remain unchanged except every other round trip time, the following comparison is made:

```
If (rtt > Di)
  Cwnd -=min(cwnd,wnd)/8;
```

We have simulated a modified version of the Dual algorithm essentially replacing Vegas's CAM with Dual's congestion detection scheme. More specifically we modified Vegas as follows to implement Dual:

- Continue to time each packet and to aggressively retransmit packets. However, the algorithm will not track changes in throughput and react to decreases.

- The increase algorithm of TCP/Dual is essentially the same as that used by Vegas. We continue to do exponential growth only every other round trip time during slow start. However the decrease algorithms differ. While Vegas decreases the window linearly if the congestion threshold is exceeded, TCP/Dual decreases the window by 12.5% if a round trip time exceeds the *Di* threshold.

- The Dual algorithm indicates that when a timeout occurs, we should reset the *Dmin* and *Dmax* values. While this might be useful to handle path switch situations, we chose not to implement this.

Figure 9 shows the behavior when three Dual connections compete (using the T1 network from Figure1). Due to sustained congestion, late starting connections will have an incorrect threshold causing the connection to act more aggressively than connections with accurate threshold values. By the time the third connection starts, the connection is not able to differentiate between propagation delay and queueing delay. If we do the same simulation, however reduce the router buffer size to 10 packets, the throughput of the three connections converges quickly, although several packets are dropped as the system converges. The difference is that Dual's 12.5% rate reduction is sufficient to clear a small amount of queueing (less than 10 packets). In the case pictured in Figure 9, a 12.5% rate reduction is not sufficient to clear the queue. The algorithm requires the queue levels to on average remain close to 0 so that late starting connections can obtain an accurate *Dmin* value.

Figure 10 shows three Dual connections (1 ftp, 2 bursty) using the multi-hop network between routers R1 and R4 in Figure 1. The top curve shows only the minimum queue level sampled every .02 seconds. The results show that Dual utilizes the bandwidth but has sustained congestion. To test the sensitivity of the algorithm to noisy RTT samples, we configured a random send delay in the range [0-2ms]. There was no difference in behavior. The reason is straightforward: Dual's threshold is on the order of ½ the buffer range. At 16Mbps, a 2 ms delay in the RTT sample corresponds to roughly 3 packets. It would take a much larger delay variation to be detected by Dual. In fact, it is

not until we increase the random send delay range to [0-.01seconds] before performance deteriorates as Dual reacts prematurely to the noisy samples.

Figure 11 shows 1 Dual connection (the dark line in the lower curve) competing with 2 Reno connections (the two light lines that start at time 5 and 10 respectively). Clearly the Reno connections are more aggressive. The system falls into a synchronized state such that the Reno connections utilize the majority of bandwidth. The problem is that the Dual threshold needs to be adjusted (i.e., increased) to compete fairly with Reno.

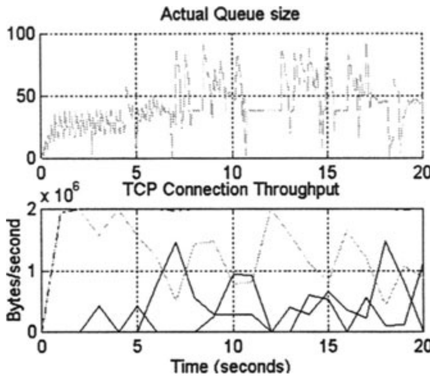

Figure 10

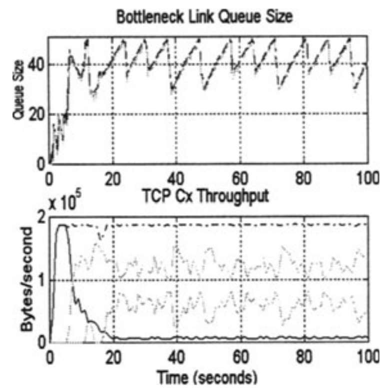

Figure 11

ARB

The Adaptive Rate-based protocol is an end-to-end congestion avoidance scheme used in IBM's Rapid Transport Protocol (RTP) [14]. ARB is a closed-loop, preventive, rate-based congestion control scheme. ARB employs a distributed algorithm that is implemented at the endpoints of an RTP connection. Each endpoint consists of an ARB sender and an ARB receiver. The ARB sender periodically queries the receiver by sending a rate request to the ARB receiver who responds with a rate reply message. The time between successive transmissions of rate request packets is defined as a *measurement_interval*. The *measurement_interval* is typically on the order of a round trip time although to minimize processing overhead, RTP products typically have a minimum *measurement_interval* on the order of .1 second.

The ARB receiver monitors changes in the delay experienced by sequential rate request packets. It maintains its own version of the time between successive rate request packet arrivals (the *receivers_measurement_interval*) and compares to the sender's *measurement_interval* (which are contained in the rate request packets). A positive difference corresponds to additional delay experienced by the probe packet. Likewise, a negative delay corresponds to less waiting time experienced by the probe as compared with the previous probe packet. ARB assumes that trends in the delay change values (i.e., a

running total of delay change samples) are reflective of the current level of congestion in the one-way path between the sender and the receiver. For example, if the *total_delay* is 0, then packets that arrive at the receiver have not experienced congestion. If the *total_ delay* is 50 mseconds, then each packet would have experienced 50 mseconds of queueing delay.

The receiver translates the *total_delay* into an estimate of the amount of queueing at the bottleneck link. In an SNA network, the receiver learns the slowest link speed in the path via the RTP connection setup protocol (known as the *max_bandwidth*). The *queueing_estimate* is simply the *total_delay* divided by the *max_bandwidth* (this gives a total queueing in bytes, RTP converts this into a number of 1000 byte packets). The fundamental control decision behind ARB is made by the receiver based on a threshold of allowed queueing. The sender is allowed to increase its rate as long as the *queueing_estimate* is less than 1 packet. If the *queueing_estimate* is between 1 and 10 packets, the sender is instructed to restrain (i.e., not to change its sending rate). If the *queueing_estimate* is between 10 and 40 packets, the sender is instructed to reduce its send rate. Finally, if the *queueing_estimate* exceeds 40 packets, ARB assumes that this is noise and tells the sender to restrain. The sender adjusts its send rate based on information received in the rate reply message. Refer to [12] for a detailed description of the ARB algorithm.

In the remainder of this section, we present and study a congestion avoidance scheme that extends TCP with the essence of ARB. We are most interested in the ARB's congestion detection scheme and in the receiver's control decision logic. We leave the study of rate control for future work. The key design points of TCP/Arb include the following:

- Keep all aspects of TCP/Reno and TCP/Vegas except remove CAM (i.e., we still want Vegas to time all packets and to remain more aggressive than Reno with retransmission). Therefore slow start and congestion avoidance are preserved.

- The sender periodically forwards a probe packet (using TCP options) that contains a measurement request signal along with the *senders_measurement_interval* (which is simply the amount of time since the sender last issues a measurement request packet).

- The receiver responds to measurement request packets from the sender by calculating the *receivers_measurement_interval* and obtains a *delay* value (by subtracting the receivers and senders measurement intervals). The receiver maintains the commulative delay in a variable called *total_delay*. The receiver also monitors the observed throughput (*throughput = byte_count/receivers_measurement_interval*) and maintains the highest throughput observed (*max_bandwidth)*. The receiver estimates the level of queueing (*queueing_estimate = total_delay / max_bandwidth*). The receiver inserts a *rate_command* message in the Ack (again using TCP options). The command (RATEINCREASE, RATEDECREASE or RESTRAIN) is based on the receiver's *queueing_estimate* with respect to the ARB thresholds. Therefore, if the

receiver's *queueing_estimate* is less than or equal to 1 packet, the receiver issues a RATEINCREASE. If the *queueing_estimate* is between 1 and 10 packets, the receiver issues a RESTRAIN. An RTP receiver will instruct the sender to RATEDECREASE when the *queueing_delay* exceeds 10 packets.

- When the sender receives a *rate_command* of RATEINCREASE, the sender increases as normal (i.e., by incrementing the *cwnd* as required during either slow start or congestion avoidance). The exception is during slow start, the sender increases the send rate exponentially every other round trip time (as done in Vegas). If the sender receives a RATEDECREASE command, the sender reduces its *cwnd* by 50%. If the sender is in slow start when it receives a RATEDECREASE command, it moves to congestion avoidance by setting the *ssthresh* to 2 (TCP/Vegas also does this).

- Probe packets are not issued during periods of recovery. Therefore, during periods of heavy packet loss, the scheme digresses to base TCP.

Figure 12 illustrates 2 TCP/Arb connections competing for bandwidth using the T1 hop between routers R2 and R3 illustrated in Figure 1. The upper curve plots the maximum queue level (dark dashed line), the minimum queue level (light dashed line) and the second connection's *queueing_estimate* (the "+" marks). Notice the the connection underestimates the queueing level. The lower curve of the figure illustrates that the second connection (the light line that starts at 2 seconds) obtains the majority of the bandwidth. Because the second connection underestimates queueing, it obtains a larger share of available bandwidth.

There are two factors that contribute to connection two's *queueing_estimate* error. First, the connection begins during a period of sustained congestion. The minimum sustained congestion experienced by the second connection is on the order of 3 packets. The connection can not detect the queue buildup which contributes the majority of the *queueing_estimate* error.

The second factor that contributes to the error is due to an inaccurate *max_bandwidth* estimate. The original ARB converts the *total_delay* to an estimate of the amount of data queued in the path based on the *max_bandwidth*. Ideally, the *max_bandwidth* is reflective of the bottleneck link speed. The TCP/Arb receiver monitors observed throughput roughly each round trip time in an attempt to estimate the *max_bandwidth*. Several factors combine to make the connection tend to underestimate the *max_bandwidth*. Clearly, the *max_bandwidth* is affected by the connection's actual sending rate. If the sender can not fill the one-way pipe (e.g., if the maximum window size is too low or if the source does not have enough data to keep the sender busy) the *max_bandwidth* will be low. Additionally, if the sender is paced by the congestion control then the *max_bandwidth* will tend to measure the connection's available bandwidth. However, due to the "ack clumping" phenomenon, TCP is actually quite bursty. For example, assume that a TCP connection has achieved a sustained throughput of ½ the bottleneck link speed.

The connection can actually be modeled as an on/off source, bursting at some high rate (determined by the ack return rate) for a certain duration (depending on the *cwnd*) and then "off" as the bottleneck link is used by other connections [13]. Once the burst is large enough to fill the one-way pipe, the connection will be able to get an accurate *max_bandwidth* estimate. Due to slow start's blind exponential growth, the typical connection (as long as it is not window constrained) will be able to get a fairly accurate estimate of the *max_bandwidth*.[3]

Figure 13 illustrates the *max_bandwidth* for the two connections shown in figure 12. The one-way pipe size is about 7 packets and if filled by the first connection in just over 1 second. At about time 3.3 seconds, Figure 13 indicates the queue level increases by 7 packets for a total congestion level of roughly 10 packets. However, connection 2 only detects an additional 2 packets of queueing. It actually observes the correct increase in *total_delay*. However, as can be seen in Figure 14, the *max_bandwidth* at this time is still low which leads to more error (in addition to the error caused by the sustained congestion). Once in steady state (after time 6 seconds), there are roughly 6-7 packets of sustained queueing. The second connection's *queueing_estimate* observes 2-3. Most of the error is caused by the sustained queueing, however an additional 1 packet error results from the second connection's *max_bandwidth* being off by roughly 25%.

Figure 14 shows 1 ftp TCP/Arb connection (the dark line) competing with 2 bursty TCP/Arb connections over the single hop T1 network shown in Figure 1. Comparing Figure 14 to the equivalent Vegas simulation run (Figure 3), the throughput of the TCP/Arb and Vegas ftp connections are is virtually identical. The difference lies in the queue levels as TCP/Arb controls the queue levels more effectively than Vegas.

Figures 15 illustrates how TCP/Arb behaves when competing with TCP/Reno connections. Figure 15 shows 1 ftp TCP-Arb connection (the dark line) compete with 2 ftp TCP/Reno connections. Given that TCP pushes the network well beyond ARB's threshold level, competing TCP connections generally "beat down" TCP/Arb connections. Because TCP/Arb accurately tracks queueing, it obtains only a fraction of available bandwidth. It is possible to tune the ARB region thresholds such that TCP/Arb connections compete with Reno connections. The problem is that the proper thresholds depend primarily on the queueing behavior at the bottleneck link which clearly can not be statically predicted.

[3] A further improvement is to use packet pair and have the receiver monitor the difference between arrival times of successive packets [10].

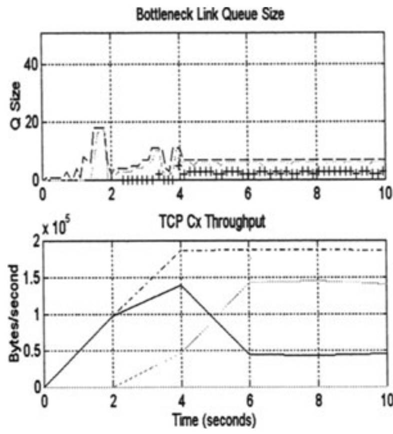

Figure 12

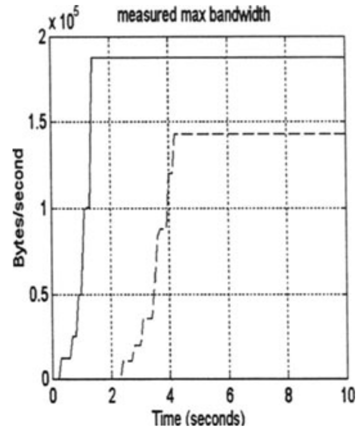

Figure 13

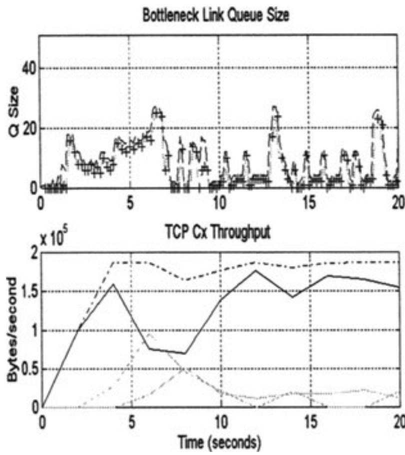

Figure 14

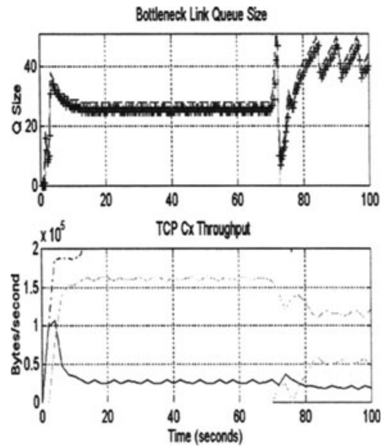

Figure 15

3 CONCLUSIONS

Based on the discussion and analysis presented in this report, we conclude with the following observations

- Congestion avoidance works properly only if the network does not suffer from sustained congestion. It is unclear how prevalent sustained congestion actually is in the Internet. However, we would like to believe that such conditions are the exception rather than the norm

- The scheme should track bottleneck queue behavior as accurately as possible. Changes in delay are the purest indicator of queue behavior. Converting a commulative delay into an estimate of queueing at the bottleneck link adds error into the control decisions as it must be based on an estimate of the bottleneck link speed.

- Path switches in the Internet are fairly common. Clearly, an algorithm such as TCP/Vegas that records the minimum round trip time measurement as a baseline can lead to disasterous results. One solution might be to deduce a path switch (via a burst of packet loss) and to reset the baseline round trip time measurement. However, clearly, path switches are problematic for end-to-end control schemes.

- Although not we did not show the simulations, we find that a one-way congestion estimate is superior to a scheme that measures congestion in both directions. For example if the return path in a TCP/Vegas connection has significantly more delay (due to congestion, different propagation delay or different link speeds), it will not fully utilize available bandwidth in the forward direction. A scheme like TCP/Arb that measures delay in a one-way direction eliminates the error.

- We have not shown the corresponding analysis, however we find that noise in the delay samples can cause sometimes significant error in the control decisions. Therefore the congestion samples must be filtered in some manner, or more likely, the threshold (i.e., the point at which the controller decides to react to congestion) must be adaptive to support changing conditions. Crucial issues involved with finding the optimal threshold point include:

  - The scheme must learn the noise floor. The goal is to find the minimum threshold such that the queue levels are contained. Clearly this conflicts with the goal of high link utilizations.

  - When competing against TCP (or other greedy protocols), a congestion avoidance scheme needs to learn the upper bound for the threshold.

  - More frequent feedback indications can be used to converge more quickly to the optimal threshold level.

In this paper we have shown that in certain environments and conditions, an end-to-end congestion avoidance algorithm based on packet transit delay measurements can be beneficial by avoiding packet loss and stabilizing response times. We have also identified several factors that can cause these algorithms to not work correctly. Unfortunately, today's best effort Internet can not guarantee that these factors such as path switches or sustained congestion will not occur.

4 REFERENCES

1. J. Ahn, P. Danzig, Z. Liu, L. Yan, "Evaluation of TCP Vegas: Emulation and Experiment", ACM SIGCOMM95.
2. O. Ait-Hellal, E. Allman, "Analysis of TCP-Vegas and TCP-Reno", ICC, June 1997.
3. L. Brakmo, S. O'Malley, L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance", ACM SIGCOMM94, 1994.
4. K. Fall, S. Floyd, S. McCanne, network simulation ns, "http://www-mash.cs.berkeley.edu/ns/ns.html, 1996.
5. S. Floyd, "TCP and Explicit Congestion Notification", ACM Computer Communications Review, October 1994.
6. S. Floyd, K. Ramakrshnan, "A Proposal to add Explicit Congestion Notification (ECN) to Ipv6 and to TCP", Internet Draft, Nov 1997, <draft-kksjf-ecn-00.txt>.
7. S. Floyd, V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", IEEE/ACM Transactions on Networking, August 1993.
8. Z. Haas, "Adaptive Admission Control", ACM SIGCOMM911, 1991.
9. V. Jacobson, "Congestion Avoidance and Control", ACM SIGCOMM88, 1988.
10. S. Keshav, "Packet-Pair Flow Control", http://www.cs.cornell.edu/skeshav/doc/94/2-17.ps.
11. A. Mankin, K. Ramakrishnan, "Gateway Congestion Control Survey" , RFC 1254, 1991.
12. J. Martin, A. Nilsson, "Congestion Control in HPR", IEEE GLOBECOM97.
13. J. Martin, et. Al., "A Comparison of TCP/Reno and RTP Transport Protocols", IBM Technical Report TR-29.2337.
14. J. Nilausen, "APPN Networks", Wiley 1994.
15. R. Stevens, TCP/IP Illustrated, Volume 1, Addison-Wesley, 1994.
16. Z. Wang, J. Crowcroft, "A New Congestion Control Scheme: Slow-start and Search (Tri-S), ACM Computer Communication Review, V21 #1, January 1991.
17. Z. Wang, J. Crowcroft, "Eliminating Periodic Packet Losses in the 4.3-Tahoe BSD TCP Congestion Control Algorithm", ACM Computer Communication Review, April 1992.

5 BIOGRAPHY

Jim Martin has been a software developer for IBM for the last 7 years. He is currently a PhD student at North Carolina State University. His research has focused on congestion control. Specifically, he is researching transport protocols with respect to congestion control for the Internet.