# Improving TCP Congestion Control over Internets with Heterogeneous Transmission Media *

Christina Parsa and J.J. Garcia-Luna-Aceves
Computer Engineering Department
Baskin School of Engineering
University of California
Santa Cruz, California 95064
chris, jj@cse.ucsc.edu

## Abstract

*We present a new implementation of TCP that is better suited to today's Internet than TCP Reno or Tahoe. Our implementation of TCP, which we call TCP Santa Cruz, is designed to work with path asymmetries, out-of-order packet delivery, and networks with lossy links, limited bandwidth and dynamic changes in delay. The new congestion-control and error-recovery mechanisms in TCP Santa Cruz are based on: using estimates of delay along the forward path, rather than the round-trip delay; reaching a target operating point for the number of packets in the bottleneck of the connection, without congesting the network; and making resilient use of any acknowledgments received over a window, rather than increasing the congestion window by counting the number of returned acknowledgments. We compare TCP Santa Cruz with the Reno and Vegas implementations using the ns2 simulator. The simulation experiments show that TCP Santa Cruz achieves significantly higher throughput, smaller delays, and smaller delay variances than Reno and Vegas. TCP Santa Cruz is also shown to prevent the swings in the size of the congestion window that typify TCP Reno and Tahoe traffic, and to determine the direction of congestion in the network and isolate the forward throughput from events on the reverse path.*

## 1 Introduction

Reliable end-to-end transmission of data is a much needed service for many of today's applications running over the Internet (e.g., WWW, file transfers, electronic mail, remote login), which makes TCP an essential component of today's Internet. However, it has been widely demonstrated that TCP exhibits poor performance over wireless networks [1, 17] and networks that have even small degrees of path asymmetries [11]. The performance problems of current TCP implementations (Reno and Tahoe) over internets of heterogeneous transmission media stem from inherent limitations in the error recovery and congestion-control mechanisms they use.

Traditional Reno and Tahoe TCP implementations perform one round-trip time estimate for each window of outstanding data. In addition, Karn's algorithm [9] dictates that, after a packet loss, round-trip time (RTT) estimates for a retransmitted packet cannot be used in the TCP RTT estimation. The unfortunate side-effect of this approach is that no estimates are made during periods of congestion – precisely the time when they would be the most useful. Without accurate RTT estimates during congestion, a TCP sender may retransmit prematurely or after undue delays. Because all prior approaches are unable to perform RTT estimates during periods of congestion, a timer-backoff strategy (in which the timeout value is essentially doubled after every timeout and retransmission) is used to avoid premature retransmissions.

Reno and Tahoe TCP implementations and many proposed alternative solutions [14, 15, 20] use packet loss as a primary indication of congestion; a TCP sender increases its window size, until packet losses occur along the path to the TCP receiver. This poses a major problem in wireless networks, where bandwidth is a very scarce resource. Furthermore, the periodic and wide fluctuation of window size typical of Reno and Tahoe TCP implementations causes high fluctuations in delay and therefore high delay variance at the endpoints of the connection – a side effect that is unacceptable for delay-sensitive applications.

Today's applications over the Internet are likely to operate over paths that either exhibit a high degree of asymmetry or which appear asymmetric due to significant load differences between the forward and reverse data paths. Under such conditions, controlling congestion based on acknowledgment (ACK) counting as in TCP Reno and Tahoe results in significant underutilization of the higher capacity forward link due to loss of ACKs on the slower reverse link [11]. ACK losses also lead to very bursty data traffic on the forward path. For this reason, a better congestion control algorithm is needed that is resilient to ACK losses.

In this paper, we propose TCP Santa Cruz, which is a new implementation of TCP implementable as a TCP option by utilizing the extra 40 bytes available in the options field of the TCP header. TCP Santa Cruz detects not only the initial stages of congestion, but can also identify the *direction* of congestion, i.e., it determines if congestion is developing in the forward path and then isolates the forward throughput from events such as congestion on the reverse path. The direction of congestion is determined by estimating the *relative delay* that one packet experiences with respect to another; this *relative delay* is the foundation of our congestion control algorithm. Our approach is significantly different from rate-controlled congestion control approaches, e.g., TCP Vegas [2], as well as those that use an increasing round-trip time (RTT) estimate as the primary indication of congestion [22, 18, 21], in that TCP Santa Cruz does not use RTT estimates in any way for congestion control. This represents a fundamental improvement over the latter approaches, because RTT measurements do not permit the sender to differentiate between delay variations due to increases or decreases in the forward or reverse paths of a connection.

TCP Santa Cruz provides a better error-recovery strategy than

* This work was supported in part at UCSC by the Office of Naval Research (ONR) under Grant N00014-99-1-0167.

Reno and Tahoe do by providing a mechanism to perform RTT estimates for every packet transmitted, including retransmissions. This eliminates the need for Karn's algorithm and does not require any timer-backoff strategies, which can lead to long idle periods on the links. In addition, when multiple segments are lost per window we provide a mechanism to perform retransmissions without waiting for a TCP timeout.

Section 2 discusses prior related work to improving TCP and compares those approaches to ours. Section 3 describes the algorithms that form our proposed TCP implementation and shows examples of their operation. Section 4 shows via simulation the performance improvements obtained with TCP Santa Cruz over the Reno and Vegas TCP implementations. Finally, Section 5 summarizes our results.

## 2 Previous Work

Congestion control for TCP is an area of active research; solutions to congestion control for TCP address the problem either at the intermediate routers in the network [8, 13, 6] or at the endpoints of the connection [2, 7, 20, 21, 22].

Router-based support for TCP congestion control can be provided through RED gateways [6], a solution in which packets are dropped in a fair manner (based upon probabilities) once the router buffer reaches a predetermined size. As an alternative to dropping packets, an Explicit Congestion Notification (ECN) [8] bit can be set in the packet header, prompting the source to slow down. Current TCP implementations do not support the ECN method. Kalampoukas et al. [13] propose an approach that prevents TCP sources from growing their congestion window beyond the bandwidth delay product of the network by allowing the routers to modify the receiver's advertised window field of the TCP header in such a way that TCP does not overrun the intermediate buffers in the network.

End-to-end congestion control approaches can be separated into three categories: rate-control, packet round-trip time (RTT) measurements, and modification of the source or receiver to return additional information beyond what is specified in the standard TCP header [20]. A problem with rate-control and relying upon RTT estimates is that variations of congestion along the reverse path cannot be identified and separated from events on the forward path. Therefore, an increase in RTT due to reverse-path congestion or even link asymmetry will affect the performance and accuracy of these algorithms. In the case of RTT monitoring, the window size could be decreased (due to an increased RTT measurement) resulting in decreased throughput; in the case of rate-based algorithms, the window could be increased in order to bump up throughput, resulting in increased congestion along the forward path.

Wang and Crowcroft's DUAL algorithm [21] uses a congestion control scheme that interprets RTT variations as indications of delay through the network. The algorithm keeps track of the minimum and maximum delay observed to estimate the maximum queue size in the bottleneck routers and keep the window size such that the queues do not fill and thereby cause packet loss. An adjustment of $\pm\frac{1}{8}cwnd$ is made to the congestion window every other round-trip time whenever the observed RTT deviates from the mean of the highest and lowest RTT ever observed. RFC 1323 [7] uses the TCP Options to include a timestamp in every data packet from sender to receiver to obtain a more accurate RTT estimate. The receiver echoes this timestamp in each ACK packet and the round-trip time is calculated with a single subtraction. This approach encounters problems when delayed ACKs are used, because it is then unclear to which packet the timestamp belongs. RFC 1323 suggests that the receiver return the earliest timestamp so that the RTT estimate takes into account the delayed ACKs, as segment loss is assumed to be a sign of congestion, and the timestamp returned is from the sequence number which last advanced the window. When a hole is filled in the sequence space, the receiver returns the timestamp from the segment which filled hole. The downside of this approach is that it cannot provide accurate timestamps when segments are lost.

Two notable rate-control approaches are the Tri-S [22] scheme and TCP Vegas [2]. Wang and Crowcroft's Tri-S algorithm [22] computes the achieved throughput by measuring the RTT for a given window size (which represents the amount of outstanding data in the network) and comparing the throughput when the window is increased by one segment. TCP Vegas has three main components: a retransmission mechanism, a congestion avoidance mechanism, and a modified slow-start algorithm. TCP Vegas provides faster retransmissions by examining a timestamp upon receipt of a duplicate ACK. The congestion avoidance mechanism is based upon a once per round-trip time comparison between the ideal (expected) throughput and the actual throughput. The ideal throughput is based upon the best RTT ever observed and the observed throughput is the throughput observed over a RTT period. The goal is to keep the actual throughput between two threshhold values, $\alpha$ and $\beta$, which represent too little and too much data in flight, respectively.

Because we are interested in solutions to TCP's performance problems applicable over different types of networks and links, our approach focuses on end-to-end solutions. However, our work is closely related to a method of bandwidth probing introduced by Keshav [10]. In this approach, two back-to-back packets are transmitted through the network and the interarrival time of their ACK packets is measured to determine the bottleneck service rate (the conjecture is that the ACK spacing preserves the data packet spacing). This rate is then used to keep the bottleneck queue at a predetermined value. For the scheme to work, it is assumed that the routers are employing round-robin or some other fair service discipline. The approach does not work over heterogeneous networks, where the capacity of the reverse path could be orders of magnitude slower than the forward path because the data packet spacing is not preserved by the ACK packets. In addition, a receiver could employ a delayed ACK strategy, which is common in many TCP implementations, and congestion on the reverse path can interfere with ACK spacing and invalidate the measurements made by the algorithm.

## 3 TCP Santa Cruz – Protocol Description

TCP Santa Cruz provides improvement over TCP Reno in two major areas : congestion control and error recovery. The congestion control algorithm introduced in TCP Santa Cruz determines when congestion exists or is developing on the forward data path – a condition which cannot be detected by a round-trip time estimate. This type of monitoring permits the detection of the incipient stages of congestion, allowing the congestion window to increase or decrease in response to early warning signs. In addition, TCP Santa Cruz uses relative delay calculations to isolate the forward throughput from any congestion that might be present along the reverse path.

The error recovery methods introduced in TCP Santa Cruz perform timely and efficient early retransmissions of lost packets, eliminate unnecessary retransmissions for correctly received packets when multiple losses occur within a window of data, and provide RTT estimates during periods of congestion and retransmission (*i.e.*, eliminate the need for Karn's algorithm). The rest of this section describes these mechanisms.

## 3.1 Congestion Control

### 3.1.1 Eliminating RTT ambiguity using relative delays

Round-trip time measurements alone are not sufficient for determining whether congestion exists along the data path. Figure 1 shows an example of the ambiguity involved when only RTT measurements are considered. Congestion is indicated by a queue along the transmission path. The example shows the transmission of two data packets and the returning ACKs from the receiver. If only round-trip time (RTT) measurements were used, then measurements $RTT_1 = 4$ and $RTT_2 = 5$, could lead to an incorrect conclusion of developing congestion in the forward path for the second packet. The true cause of increased RTT for the second packet is congestion along the reverse path, not the data path. Our protocol solves this ambiguity by introducing the notion of the *relative forward delay*.

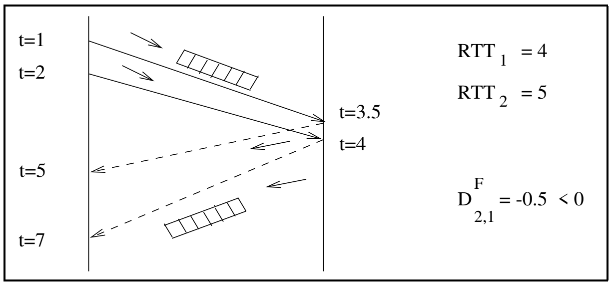



**Figure 1. Example of RTT ambiguity**

*Relative delay* is the increase and decrease in delay that packets experience with respect to each other as they propagate through the network. These measurements are the basis of our congestion control algorithm. The sender calculates the *relative delay* from a timestamp contained in every ACK packet that specifies the arrival time of the packet at the destination. From the *relative delay* measurement the sender can determine whether congestion is increasing or decreasing in either the forward or reverse path of the connection; furthermore, the sender can make this determination for every ACK packet it receives. This is impossible to accomplish using RTT measurements.

Figure 2 shows the transfer of two sequential packets transmitted from a source to a receiver and labeled #1 and #2. The sender maintains a table with the following two times for every packet: (a) the transmission time of the data packet at the source, and (b) the arrival time of the data packet at the receiver, as reported by the receiver in its ACK. From this information, the sender calculates the following time intervals for any two data packets $i$ and $j$ (where $j > i$): $S_{j,i}$, the time interval between the transmission of the packets; and $R_{j,i}$, the inter-arrival time of the data packets at the receiver. From these values, the *relative forward delay*, $D^F_{j,i}$, can be obtained:

$$D^F_{j,i} = R_{j,i} - S_{j,i} \tag{1}$$

where $D^F_{j,i}$ represents the change in forward delay experienced by packet $j$ with respect to packet $i$.

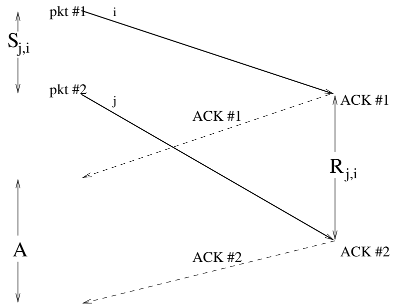



**Figure 2. Transmission of 2 packets and corresponding relative delay measurements**

Figure 3 illustrates how we detect congestion in the forward path. As illustrated in Figure 3(a), when $D^F_{j,i} = 0$ the two packets experience the same amount of delay in the forward path. Figure 3(b) shows that the first packet was delayed more than the second packet whenever $D^F_{j,i} < 0$. Figure 3(c) shows that the second packet has been delayed with respect to the first one when $D^F_{j,i} > 0$. Finally, Figure 3(d) illustrates out-of-order arrival at the receiver. In the latter case, the sender is able to determine the presence of multiple paths to the destination by the timestamps returned from the receiver. Although the example illustrates measurements based on two consecutive packets, TCP Santa Cruz does not require that the calculations be performed on two sequential packets; however, the granularity of the measurements depends on the ACK policy used.

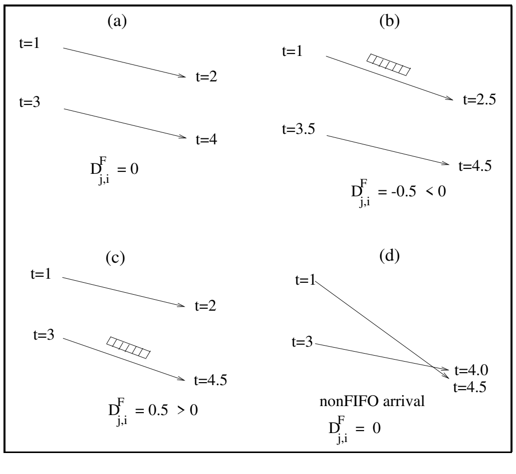



**Figure 3. FORWARD PATH: (a)Equal delay (b)1st packet delayed (c)2nd packet delayed (d)non-FIFO arrival of packets**

### 3.1.2 Congestion Control Algorithm

At any time during a connection, the network queues (and specifically the bottleneck queue) are in one of three states: increasing

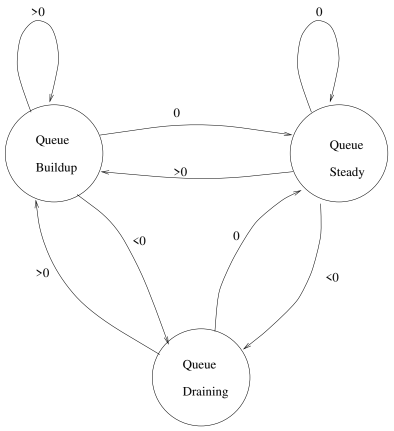



**Figure 4. Based upon the value of $D^F_{j,i}$, the bottleneck queue can be currently filling, draining or maintaining.**

in size, decreasing in size, or maintaining their current state. The state diagram of Figure 4 shows how the computation of the relative forward delay, $D^F_{j,i}$, allows the determination of the change in queue state. The goal in TCP Santa Cruz is to allow the network queues (specifically the bottleneck queue) to grow to a desired size; the specific algorithm to achieve this goal is described next.

The positive and negative *relative delay* values represent additional or less queueing in the network, respectively. Summing the *relative delay* measurements over a period of time provides an indication of the level of queueing at the bottleneck. If the sum of *relative delays* over an interval equals 0, it indicates that, with respect to the beginning of the interval, no additional congestion or queueing was present in the network at the end of the interval. Likewise, if we sum relative delays from the beginning of a session, and at any point the summation equals zero, we would know that all of the data for the session are contained in the links and not in the network queues (assuming the queues were initially empty).

The congestion control algorithm of TCP Santa Cruz operates by summing the relative delays from the beginning of a session, and then updating the measurements at discrete intervals, with each interval equal to the amount of time to transmit a windowful of data and receive the corresponding ACKs. Since the units of $D^F_{j,i}$ is time (seconds), the relative delay sum must then be translated into an equivalent number of packets (queued at the bottleneck) represented by this delay. In other words, the algorithm attempts to maintain the following condition:

$$n_{t_i} = N_{op} = n_{t_{i-1}} + M_{W_{i-1}} \tag{2}$$

where $n_{t_i}$ is the total number of packets queued at the bottleneck at time $t_i$; $N_{op}$ is the operating point (the desired number of packets, per session, to be queued at the bottleneck); $M_{W_{i-1}}$ is the additional amount of queueing introduced over the previous window $W_{i-1}$; and $n_{t_1} = M_{W_0}$.

**The operating point:** The operating point, $N_{op}$, is the desired number of packets to reside in the bottleneck queue. The value of $N_{op}$ should be greater than zero; the intuition behind this decision is that an operating point equal to zero would lead to underutilization of the available bandwidth because the queues are always empty, *i.e.,* no queueing is tolerated. Instead, the goal is to allow a small amount of queueing so that a packet is always available for forwarding over the bottleneck link. For example, if we choose $N_{op}$ to be 1, then we expect a session to maintain 1 packet in the bottleneck queue, i.e., our ideal or desired congestion window would be one packet above the bandwidth delay product (BWDP) of the network.

**Translating the relative delay:** The relative delay gives an indication of the change in network queueing, but provides no information on the actual number of packets corresponding to this value. We translate the sum of relative delays into the equivalent number of queued packets by first calculating the average packet service time, $pktS$ $(sec/pkt)$, achieved by a session over an interval. This rate is of course limited by the bottleneck link.

Our model of the bottleneck link, depicted in Figure 5, consists of two delay parameters: the queueing delay, $\tau_Q$; and the output service time, $pktS$ (the amount of time spent servicing a packet). The queueing delay is variable and is controlled by the congestion control algorithm (by changing the sender's congestion window) and by network cross-traffic. The relative delay measurements provide some feedback about this value. The output rate of a FIFO queue will vary according to the number of sessions and the burstiness of the arrivals from competing sessions. The packet service rate is calculated as

$$pktS = \frac{R}{\#\ \text{pkts received}} \tag{3}$$

where $R$ is the difference in arrival time of any two packets as calculated from the timestamps returned by the receiver. Because $pktS$ changes during an interval, we calculate the average packet service time, $\overline{pktS}$, over the interval. Finally, we translate the sum of relative delays over the interval into the equivalent number of packets represented by the sum by dividing the *relative delay* summation by the average time to service a packet. This gives us the number of packets represented by the delay over an interval:

$$M_{W_{i-1}} = \frac{\sum D^F_k}{\overline{pktS}} \tag{4}$$

where $k$ are packet-pairs within window $W_{i-1}$. The total queueing in the system at the end of the interval is determined by Eq. 2.

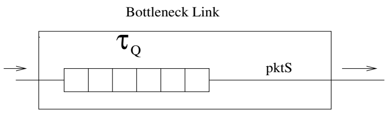



**Figure 5. Bottleneck Link: Delay consists of two parts: $\tau_Q$, the delay due to queueing and $pktS$, the packet service time over the link.**

**Adjusting the window:** The TCP Santa Cruz congestion window is adjusted such that Eq. 2 is satisfied within a range of $N_{op} \pm \delta$, where $\delta$ is some fraction of a packet. Adjustments are made to the congestion window only at discrete intervals, *i.e.,* in the time taken to empty a windowful of data from the network. Over this interval, $M_{W_{i-1}}$ is calculated and at the end of the interval it is added to $n_{t_{i-1}}$. If the result falls within the range of $N_{op} \pm \delta$, the congestion window is maintained at its current size. If, however, $n_{t_i}$ falls below $N_{op} - \delta$, the system is not being pushed enough and the window is increased linearly during the next interval. If $n_{t_i}$ rises above $N_{op} + \delta$, then the system is being pushed too high above the desired operating point, and the congestion window is decreased linearly during the next interval.

In TCP Reno, the congestion control algorithm is driven by the arrival of ACKs at the source (the window is incremented by $1/cwnd$ for each ACK while in the congestion avoidance phase). This method of ACK counting causes Reno to perform poorly when ACKs are lost [11]; unfortunately, ACK loss becomes a predominant feature in TCP over asymmetric networks [11]. Given that the congestion control algorithm in TCP Santa Cruz makes adjustments to the congestion window based upon delays through the network and not on the arrival of ACKs in general, the algorithm is robust to ACK losses.

**Startup** Currently, the algorithm used by TCP Santa Cruz at startup is the `slow start` algorithm used by TCP Reno with two modifications. First, the initial congestion window, $cwnd$, is set to two segments instead of one so that initial values for $D^F_{i,j}$ can be calculated. Second, the algorithm may stop `slow start` before $ssthresh > cwnd$ if any relative delay measurement or $n_{t_i}$ exceeds $N_{op}/2$. Once stopped, `slow start` begins again only if a timeout occurs. During `slow start`, the congestion window doubles every round-trip time, leading to an exponential growth in the congestion window. One problem with `slow start` is that such rapid growth often leads to congestion in the data path [2]. TCP-SC reduces this problem by ending `slow start` once any queue buildup is detected.

## 3.2 Error Recovery

### 3.2.1 Improved RTT estimate

TCP Santa Cruz provides better RTT estimates over traditional TCP approaches by measuring the round-trip time (RTT) of every segment transmitted for which an ACK is received, including retransmissions. This eliminates the need for Karn's algorithm [9] (in which RTT measurements are not made for retransmissions) and timer-backoff strategies (in which the timeout value is essentially doubled after every timeout and retransmission). To accomplish this, TCP Santa Cruz requires each returning ACK packet to indicate the precise packet that caused the ACK to be generated and the sender must keep a timestamp for each transmitted or retransmitted packet. Packets can be uniquely identified by specifying both a sequence number and a retransmission copy number. For example, the first transmission of packet 1 is specified as 1.1, the second transmission is labeled 1.2, and so forth. In this way, the sender can perform a new RTT estimate for every ACK it receives. Therefore, ACKs from the receiver are logically a triplet consisting of a cumulative ACK (indicating the sequence number of the highest in-order packet received so far), and the two-element sequence number of the packet generating the ACK (usually the most recently received packet). For example, ACK (5.7.2) specifies a cumulative ACK of 5, and that the ACK was generated by the second transmission of a packet with sequence number 7. As with traditional TCP implementations, we do not want the RTT estimate to be updated too quickly; therefore, a weighted average is computed for each new value received. We use the same algorithm as TCP Tahoe and Reno; however, the computation is performed for every ACK received, instead of once per RTT.

### 3.2.2 ACK Window

To assist in the identification and recovery of lost packets, the receiver in TCP Santa Cruz returns an *ACK Window* to the sender to indicate any holes in the received sequential stream. In the case of multiple losses per window, the ACK Window allows TCP-SC to retransmit all lost packets without waiting for a TCP timeout.

The ACK Window is similar to the bit vectors used in previous protocols, such as NETBLT [4] and TCP-SACK [5][15]. Unlike TCP-SACK, our approach provides a new mechanism whereby the receiver is able to report the status of every packet within the current transmission window.[1] The ACK Window is maintained as a vector in which each bit represents the receipt of a specified number of bytes beyond the cumulative ACK. The receiver determines an optimal granularity for bits in the vector and indicates this value to the sender via a one-byte field in the header. A maximum of 19 bytes are available for the ACK window to meet the 40-byte limit of the TCP option field in the TCP header. The granularity of the bits in the window is bounded by the receiver's advertised window and the 18 bytes available for the ACK window; this can accommodate a 64K window with each bit representing 450 bytes. Ideally, a bit in the vector would represent the MSS of the connection, or the typical packet size. Note this approach is meant for data intensive traffic, therefore bits represent at least 50 bytes of data. If there are no holes in the expected sequential stream at the receiver, then the ACK window is not generated.

Figure 6 shows the transmission of five packets, three of which are lost and shown in grey (1,3, and 5). The packets are of variable size and the length of each is indicated by a horizontal arrow. Each bit in the ACK window represents 50 bytes with a 1 if the bytes are present at the receiver and a 0 if they are missing. Once packet #1 is recovered, the receiver would generate a cumulative ACK of 1449 and the bit vector would indicate positive ACKs for bytes 1600 through 1849. There is some ambiguity for packets 3 and 4 since the ACK window shows that bytes 1550 – 1599 are missing. The sender knows that this range includes packets 3 and 4 and is able to infer that packet 3 is lost and packet 4 has been received correctly. The sender maintains the information returned in the ACK Window, flushing it only when the window advances. This helps to prevent the unnecessary retransmission of correctly received packets following a timeout when the session enters `slow start`.

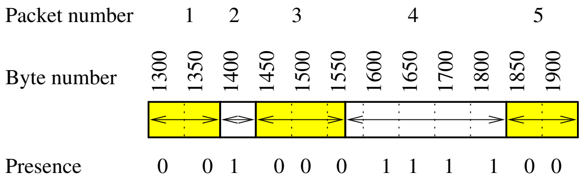



**Figure 6. ACK window transmitted from receiver to sender. Packets 1, 3 and 5 are lost.**

[1] TCP-SACK is generally limited by the TCP options field to reporting only three unique segments of continuous data within a window.

### 3.2.3 Retransmission Policy

Our retransmission strategy is motivated by such evidence as the Internet trace reports by Lin and Kung, which show that 85% of TCP timeouts are due to “non-trigger” [12]. Non-trigger occurs when a packet is retransmitted by the sender without previous attempts, *i.e.,* when three duplicate ACKs fail to arrive at the sender and therefore TCP’s fast retransmission mechanism never happens. In this case, no retransmissions can occur until there is a timeout at the source. Therefore, a mechanism to quickly recover losses without necessarily waiting for three duplicate ACKs from the receiver is needed.

Given that TCP Santa Cruz has a much tighter estimate of the RTT time per packet and that the TCP Santa Cruz sender receives precise information on each packet correctly received (via the ACK Window), TCP Santa Cruz can determine when a packet has been dropped without waiting for TCP’s `Fast Retransmit` algorithm. TCP Santa Cruz can quickly retransmit and recover a lost packet once any ACK for a subsequently transmitted packet is ‘received and a time constraint is met. Any lost packet $y$, initially transmitted at time $t_i$ is marked as a hole in the ACK window. Packet $y$ can be retransmitted once the following constraint is met: as soon as an ACK arrives for any packet transmitted at time $t_x$ (where $t_x > t_i$), and $t_{current} - t_i > RTT$, where $t_{current}$ is the current time and $RTT$ is the estimated round-trip time of the connection. Therefore, any packet marked as unreceived in the ACK window can be a candidate for early retransmission.

## 3.3 Proposed Implementation

TCP Santa Cruz can be implemented as a TCP option containing the fields depicted in Table 1. The TCP Santa Cruz option can vary in size from 11 to 40 bytes, depending on the size of the ACK window (see Section 3.2.2).

| Field | Size (bytes) | Description |
|---|---|---|
| Kind | 1 | kind of protocol |
| Length | 1 | length field |
| Data.copy | 4 (bits) | retrans. number of data pkt |
| ACK.copy | 4 (bits) | retrans. number of data pkt gen. ACK |
| ACK.sn | 4 | SN of data pkt generating ACK |
| Timestamp | 4 | arr. time of data pkt generating ACK |
| ACK Window Granularity | 1 | num. bytes represented by each bit |
| ACK Window | 0 – 18 | holes in the receive stream |

**Table 1. TCP Santa Cruz options field description**

# 4 Performance Results

In this section we examine the performance of TCP Santa Cruz compared to TCP Vegas [3] and TCP Reno [19]. We first show performance results for a basic configuration with a single source and a bottleneck link, then a single source with cross-traffic on the reverse path, and finally performance over asymmetric links. We have measured performance for TCP Santa Cruz through simulations using the “ns” network simulator [16]. The simulator contains implementations of TCP Reno and TCP Vegas. TCP Santa Cruz was implemented by modifying the existing TCP-Reno source code to include the new congestion avoidance and error-recovery schemes. Unless stated otherwise, data packets are of size 1Kbyte, the maximum window size, $cwnd\_max$ for every TCP connection is 64 packets and the initial `ssthresh` is equal to $\frac{1}{2} * cwnd\_max$. All simulations are an FTP transfer with a source that always has data to send; simulations are run for 10 seconds. In addition, the TCP clock granularity is $100ms$ for all protocols.

## 4.1 Basic Bottleneck Configuration

Our first experiment shows protocol performance over a simple network, depicted in Figure 7, consisting of a TCP source sending 1Kbyte data packets to a receiver via two intermediate routers connected by a 1.5Mbps bottleneck link. The bandwidth delay product (BWDP) of this configuration is equal to 16.3Kbytes; therefore, in order to accommodate one windowful of data, the routers are set to hold 17 packets.

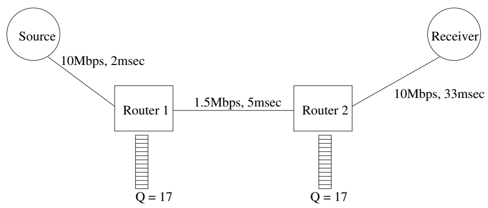


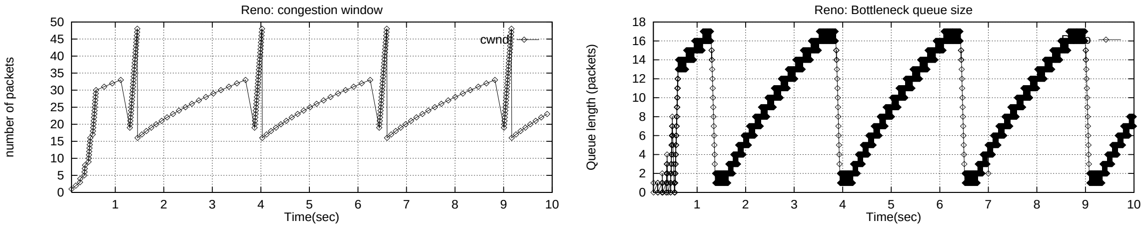

**Figure 7. Basic bottleneck configuration**

Figures 8 (a) and (b) show the growth of TCP Reno’s congestion window and the queue buildup at the bottleneck link. Once the congestion window grows beyond 17 packets (the BWDP of the connection) the bit pipe is full and the queue begins to fill. The routers begin to drop packets once the queue is full; eventually Reno notices the loss, retransmits, and cuts the congestion window in half. This produces see-saw oscillations in both the window size and the bottleneck queue length. These oscillations greatly increase not only delay, but also delay variance for the application. It is increasingly important for real-time and interactive applications to keep delay and delay variance to a minimum.

In contrast, Figures 9 (a) and (b) show the evolution of the sender’s congestion window and the queue buildup at the bottleneck for TCP Santa Cruz. These figures demonstrate the main strength of TCP Santa Cruz: adaptation of the congestion control algorithm to transmit at the bandwidth of the connection without congesting the network and without overflowing the bottleneck queues. In this example the threshold value of $N_{op}$, the desired additional number of packets in the network beyond the BWDP, is set to $N_{op} = 1.5$. Figure 9(b) shows the queue length at the bottleneck link for TCP Santa Cruz reaches a steady-state value between 1 and 2 packets. We also see that the congestion window, depicted in Figure 9(a) reaches a peak value of 18 packets, which is the sum of the BWDP (16.5) and $N_{op}$. The algorithm maintains this steady-state value for the duration of the connection.

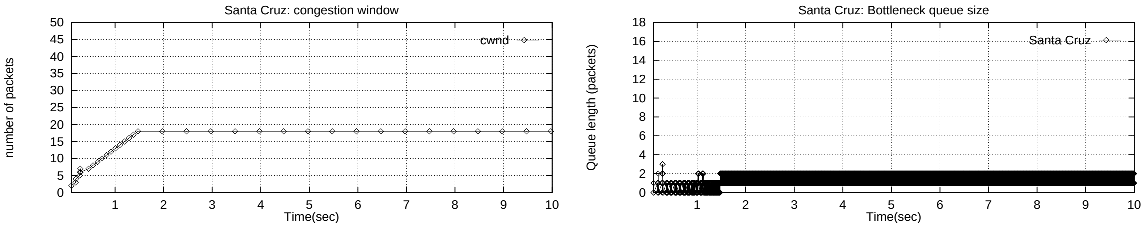
Table 2 compares the throughput, average delay and delay variance for Reno, Vegas and Santa Cruz. For TCP Santa Cruz we vary the amount of queueing tolerated in the network from $N_{op}$ = 1 to 5 packets. All protocols achieve similar throughput, with Santa Cruz $n = 5$ performing slightly better than Reno. The reason Reno’s throughput does not suffer is that most of the time the congestion window is well above the BWDP of the network so that packets are always queued up at the bottleneck and therefore available for transmission. What does suffer, however, is the delay experienced by packets transmitted through the network.

The minimum forward delay through the network is equal to 40 msec propagation delay plus 6.9 msec packet forwarding time,




(a) (b)

**Figure 8. TCP Reno: (a) congestion window (b) bottleneck queue**




(a) (b)

**Figure 9. TCP Santa Cruz: (a) congestion window (b) bottleneck queue**

yielding a total minimum forward delay of approximately 47 msec. Reno is the clear loser in this case with not only the highest average delay, but also a high delay variation. Santa Cruz with $N_{op} = 1.5$ provides the same average delay as Vegas, but a with lower delay deviation. As $N_{op}$ increases, the delay in Santa Cruz also increases because more packets are allowed to sit in the bottleneck queue. Also, throughput is seen to grow with $N_{op}$ because not only does the `slowstart` period last longer, but the peak window size is reached earlier in the connection, leading to a faster transmission rate earlier in the transfer. In addition, with a larger $N_{op}$ it is more likely that a packet is available in the queue awaiting transmission.

| Comparison of Throughput and Average Delay | | | | |
|---|---|---|---|---|
| Protocol | Throughput (Mbps) | Utilization | Ave. delay (msec) | Delay variance (msec) |
| Reno | 1.45 | 0.97 | 99.4 | 2.06 |
| SC n=1.5 | 1.42 | 0.94 | 55.1 | 0.0041 |
| SC n=3 | 1.45 | 0.97 | 60.6 | 0.0063 |
| SC n=5 | 1.47 | 0.98 | 79.2 | 0.0073 |
| Vegas (1,3) | 1.40 | 0.94 | 55.2 | 0.0077 |

**Table 2. Throughput, delay and delay variance comparisons for Reno, Vegas and Santa Cruz for basic bottleneck configuration.**

## 4.2 Traffic on Reverse Link

This section looks at how data flow on the forward path is affected by traffic and congestion on the reverse path. Figure 10 shows that, in addition to a TCP source from A to the Receiver, there is also a TCP Reno source from B to Router 1 in order to cause congestion on the reverse link.

Figure 11 (a) shows that the congestion window growth for Reno source A is considerably slower compared to the case when no reverse traffic is present in Figure 8(a). Because Reno grows its window based on ACK counting, lost and delayed ACKs on the reverse path prevent Source A from filling the bit pipe as fast as it could normally do, resulting in low link utilization on the forward

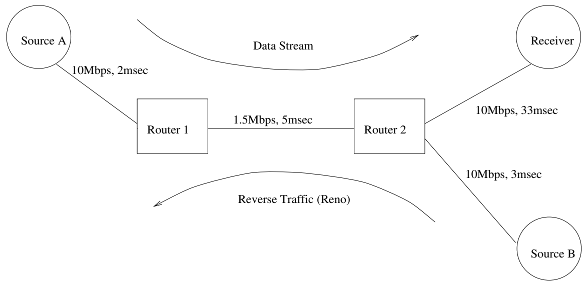


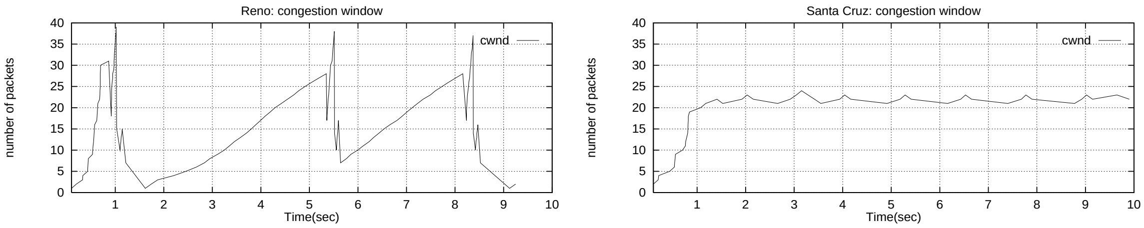

**Figure 10. Traffic on the reverse link**

path. In addition, ACK losses also delay the detection of packet losses at the source, which is waiting for three duplicate ACKs to perform a retransmission. In contrast, Figure 11 (b) shows that the congestion window for TCP Santa Cruz ($N_{op}$ = 5) is relatively unaffected by the Reno traffic on the reverse path and reaches the optimal window size of around 22 packets, demonstrating TCP Santa Cruz's ability to maintain a full data pipe along the forward path in the presence of congestion on the reverse path.

Table 3 shows the throughput and delay obtained for Reno, Santa Cruz and Vegas. Santa Cruz achieves up to a 68% improvement in throughput compared to Reno and a 78% improvement over Vegas. Because of the nearly constant window size, the variation delay with our algorithm is considerably lower than Reno. Vegas suffers from low throughput in this case because its algorithm is unable to maintain a good throughput estimate because of high variation in RTT measurements. Vegas exhibits low delay primarily due to its low utilization of the bottleneck link; this insures that packets are never queued at the bottleneck and therefore do not incur any additional queueing delay from source to destination.

## 4.3 Asymmetric Links

In this section we investigate performance over networks that exhibit asymmetry, *e.g.,* ADSL, HFC or combination networks, which




(a) (b)

**Figure 11. Comparison of congestion window growth when TCP-Reno traffic is present on the reverse path: (a) Reno (b) TCP Santa Cruz n=5**

| Comparison of Throughput and Average Delay | | | | |
|---|---|---|---|---|
| Protocol | Throughput (Mbps) | Utilization | Ave. delay (msec) | Delay variance (msec) |
| Reno | 0.823 | 0.55 | 134.3 | 56.2 |
| SC n=1.5 | 1.213 | 0.81 | 54.8 | 0.0034 |
| SC n=3 | 1.312 | 0.87 | 60.6 | 0.0057 |
| SC n=5 | 1.390 | 0.92 | 73.4 | 0.0080 |
| Vegas (1,3) | 0.778 | 0.52 | 49.5 | 0.0016 |

**Table 3. Throughput, delay and delay variance comparisons with traffic on the reverse link.**

may have a high bandwidth cable downstream link and a slower telephone upstream link. TCP has been shown to perform poorly over asymmetric links [11] primarily because of ACK loss, which causes burstiness at the source (the size of the bursts are proportional to the degree of asymmetry) and leads to buffer overflow along the higher bandwidth forward path; and also reduced throughput because of slow window growth at the source due to lost ACK packets. Lakshman et. al. [11] define the normalized asymmetry $k$ of a path as the ratio of the transmission capacity of data packets on the forward path to ACK packets on the reverse path. This is an important measurement because it means the source puts out $k$ times as many data packets as the reverse link has capacity. Once the queues in the reverse path fill, only one ACK out of $k$ will make it back to the receiver. Each ACK that does arrive at the source then generates a burst of $k$ packets in the forward path. In addition, during congestion avoidance, the window growth will be slowed by $1/k$ as compared to a symmetric connection.

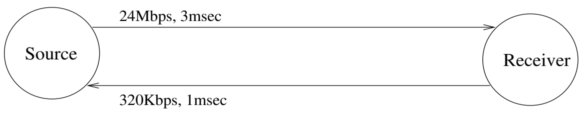



**Figure 12. Simulation configuration for asymmetric links**

The simulation configuration depicted in Figure 12 has been studied by Lakshman et. al. in detail and is used here to examine performance. In this configuration the forward buffer, $B_f = 9$ packets. Using 1 Kbyte data packets this results in a normalized asymmetry factor $k = 3$.

Figure 13 (a) shows the congestion window growth for Reno. Because of the burstiness of the connection due to ACK loss, there are several lost data packets per window of data, causing Reno to suffer timeouts every cycle (that is why the congestion window reduces to 1 packet). Figure 13 (b) shows the development of the window with TCP Santa Cruz. In this case, the congestion window settles a few packets above the BWDP (equal to 31 packets) of the connection.[2] During `slow start` there is an initial overshoot of the window size during one round-trip time delay, *i.e.,* the final round before the algorithm picks up the growing queue, a burst of packets is sent, which ultimately overflows the buffer.

A comparison of the overall throughput and delay obtained by a Reno, Vegas and Santa Cruz ($N_{op} = 1.5$, $N_{op} = 3$ and $N_{op} = 5$) sources is shown below in Table 4. This table shows that Reno and Vegas are unable to achieve link utilization above 52%. Because of the burstiness of the data traffic, Santa Cruz needs an operating point of at least $N_{op} = 3$ in order to achieve high throughput. For $N_{op} = 3$ and $N_{op} = 5$ Santa Cruz is able to achieve 99% link utilization. The end-to-end delays for Reno are around twice that of Santa Cruz and the delay variance is seven orders of magnitude greater than Santa Cruz. Because Vegas has such low link utilization the queues are generally empty, thus there is a very low delay and no appreciable delay variance.

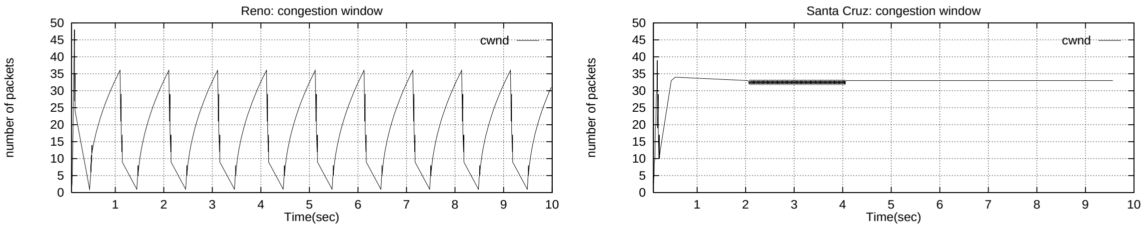
| Comparison of Throughput and Average Delay | | | | |
|---|---|---|---|---|
| Protocol | Throughput (Mbps) | Utilization | Ave. delay (msec) | Delay variance ($\mu$sec) |
| Reno | 1.253 | 0.52 | 8.4 | 1400 |
| SC n=1.5 | 1.275 | 0.53 | 3.5 | 0.0004 |
| SC n=3 | 23.72 | 0.99 | 4.6 | 0.0003 |
| SC n=5 | 23.73 | 0.99 | 4.8 | 0.0003 |
| Vegas (1,3) | 0.799 | 0.33 | 3.3 | 0.0000 |

**Table 4. Throughput, delay and delay variance over asymmetric links.**

## 5 Conclusion

We have presented TCP Santa Cruz, which implements a new approach to end-to-end congestion control and reliability, and that can be implemented as a TCP option. TCP Santa Cruz makes use of a simple timestamp returned from the receiver to estimate the level of queueing in the bottleneck link of a connection. The protocol successfully isolates the forward throughput of the connection from events on the reverse link by considering the changes in delay along the forward link only. We successfully decouple the growth of the congestion window from the number of returned ACKs (the approach taken by TCP), which makes the protocol resilient to ACK loss. The protocol provides quick and efficient error-recovery by identifying losses via an ACK window without waiting for three

[2] See Lakshman et. al. [11] for a detailed analysis of the calculation of the BWDP.




(a) (b)

**Figure 13. Comparison of congestion window growth: (a) Reno (b) TCP Santa Cruz**

duplicate ACKs. An RTT estimate for every packet transmitted (including retransmissions) allows the protocol to recover from lost retransmissions without using timer-backoff strategies, eliminating the need for Karn's algorithm.

Simulation results show that TCP Santa Cruz provides high throughput and low end-to-end delay and delay variance over networks with a simple bottleneck link, networks with congestion in the reverse path of the connection, and networks which exhibit path asymmetry. We have shown that TCP Santa Cruz eliminates the oscillations in the congestion window, but still maintains high link utilization. As a result, it provides much lower delays than current TCP implementations. For the simple bottleneck configuration our protocol provides a 20% – 45% improvement in end-to-end delay (depending on the value of $N_{op}$) and a delay variance three orders of magnitude lower than Reno. For experiments with congestion on the reverse path, TCP Santa Cruz provides an improvement in throughput of at least 47% – 67% over both Reno and Vegas, as well as an improvement in end-to-end delay of 45% – 59% over Reno with a reduction in delay variance of three orders of magnitude. When we examine networks with path asymmetry, Reno and Vegas achieve link utilization of only 52% and 33%, respectively, whereas Santa Cruz achieves 99% utilization. End-to-end delays for this configuration are also reduced by 42% – 58% over Reno.

Our simulation experiments indicate that our end-to-end approach to congestion control and error recovery is very promising, and our current work focuses on evaluating the fairness of TCP Santa Cruz, its coexistence with other TCP implementations, and its performance over wireless networks.

## 6. References


[1] H. Balakrishnan, S. Seshan, and R. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks*, Dec. 1995.

[2] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. SIGCOMM'94*, pages –, London, UK, Aug./Sept. 1994. ACM.

[3] L. Brakmo and L. Peterson. TCP Vegas: End-to-end congestion avoidance on a global internet. In *IEEE Journal of Selected Areas in Communication*, October, 1995.

[4] D. Clark, M. Lambert, and L. Zhang. NETBLT: A high throughput transfer protocol. In *Proc. SIGCOMM*, pages 353–59, Stowe, Vermont, Aug. 1987. ACM.

[5] K. Fall and S. Floyd. Simulation-based comparisons of tahoe,reno, and SACK TCP. In *Computer Communication Review*, volume 26 No. 3, pages 5 – 21, July, 1996.

[6] S. Floyd and V. Jacobson. Random Early Detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 4(1):397–413, Aug 1993.

[7] R. Jacobson, R. Braden, and D. Borman. Rfc 1323 TCP extensions for high performance. Technical report, May 1992. Technical Report 1323.

[8] V. Jacobson and S. Floyd. TCP and explicit congestion notification. In *Computer Communication Review*, volume 24 No. 5, pages 8 – 23, October, 1994.

[9] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In *Computer Communication Review*, volume 17 No. 5, pages 2 – 7, August 1987.

[10] S. Keshav. Packet-pair flow control. http://www.cs.cornell.edu/skeshav/papers.html.

[11] T. Lakshman, U. Madhow, and B. Suter. Window-based error recovery and flow control with a slow acknowledgement channel: a study of TCP/IP performance. In *Proceedings IEEE INFOCOM '97*, number 3, pages 1199–209, Apr 1997.

[12] D. Lin and H. Kung. TCP fast recovery strategies: Analysis and improvements. In *Proceedings IEEE INFOCOM '98*, Apr. 1998.

[13] L.Kalampoukas, A. Varma, and K. Ramakrishnan. Explicit window adaptation: a method to enhance TCP performance. In *Proceedings IEEE INFOCOM '98*, pages 242 – 251, Apr. 1998.

[14] M. Mathis and J. Mahdavi. Forward acknowledgment: Refining TCP congestion control. In *Proc. SIGCOMM'96*, pages 1–11, Stanford, California, Sept. 1996.

[15] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. Technical report, Oct. 1996. RFC 2018.

[16] S. McCanne and S. Floyd. ns-lbnl network simulator. http://www-nrg.ee.lbl.gov/ns/.

[17] C. Parsa and J. Garcia-Luna-Aceves. Improving TCP performance over wireless networks at the link layer. *ACM Mobile Networks and Applications Journal*, 1999. to appear.

[18] N. Samaraweera and G. Fairhurst. Explicit loss indication and accurate RTO estimation for TCP error recovery using satellite links. In *IEE Proceedings - Communications*, volume 144 No. 1, pages 47 – 53, Feb., 1997.

[19] W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, 1996.

[20] J. Waldby, U. Madhow, and T. Lakshman. Total acknowledgements: a robust feedback mechanism for end-to-end congestion control. In *Sigmetrics '98 Performance Evaluation Review*, volume 26, 1998.

[21] Z. Wang and J. Crowcroft. Eliminating periodic packet losses in the 4.3-Tahoe BSD TCP congestion control algorithm. In *Computer Communication Review*, volume 22 No. 2, pages 9 – 16, April, 1992.

[22] Z. Wang and J. Crowcroft. A new congestion control scheme: Slow start and search (Tri-S). In *Computer Communication Review*, volume 21 No. 1, pages 32 – 43, Jan., 1991.