Understanding the Performance of TCP Pacing

Amit Aggarwal
amit@cs.washington.edu
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195 USA

Abstract

Many researchers have observed that TCP's congestion control mechanisms can lead to bursty traffic flows on modern high-speed networks, with a negative impact on overall network efficiency. A proposed solution to this problem is to evenly space, or "pace", data sent into the network over an entire round-trip time, so that data is not sent in a burst. In this paper, we quantitatively evaluate this approach. We show that pacing offers better fairness, throughput, and lower drop rates in some situations. However, contrary to our initial intuition, pacing often has significantly worse throughput than regular TCP because it is susceptible to synchronized losses. We propose and evaluate approaches for eliminating this effect.

1 Introduction

The Transmission Control Protocol (TCP) is the de-facto standard for reliable, unicast, best-effort communication on the Internet. One reason for TCP's success is that its behavior has proven to be extremely robust across a wide variety of environments. As the Internet has grown, TCP has evolved to adapt to new operating conditions and performance demands, adding congestion control [16], adaptive timers [17], and small packet avoidance [25]. More recently, a number of extensions such as larger initial windows [1, 2], selective acknowledgements [22], and others [3, 4, 5, 10, 26, 31] have been proposed to improve performance and cope with the problems posed by changing Internet conditions.

One such problem is the interaction between TCP and modern high bandwidth and highly multiplexed networks. Because TCP uses incoming acknowledgments to clock out new data, called "ack-clocking", TCP tends to produce very bursty packet flows. In theory, when a burst arrives at a bottleneck router, the bottleneck has the effect of spacing out traffic on the downstream link. Well-spaced acknowledgments then trigger well-spaced new data transmissions. In practice, while ack-clocking works well for low-bandwidth links used by a small number of TCP flows, when many flows are sharing a high-bandwidth link, each flow's data packets progress together through the bottleneck causing clumped acknowledgments, in turn causing clumped new data transmissions.

Since burstiness is well-known from queuing theory to produce higher queueing delays, more packet losses, and lower throughput [19], many researchers have proposed replacing TCP's window-based ack-clocking mechanism with explicit control over the rate that data is sent into the network [6, 8, 18, 24]. Instead of sending new data into the network only when old data is acknowledged to have departed the network, these approaches send packets at a pre-determined rate. Unfortunately, rate control has its own problems. Depending on the exact mechanism used to set the target rate, rate control can be less responsive to rapid increases in congestion. In contrast, TCP senders will immediately stop sending when the network becomes congested – if no acknowledgments are being returned, no new data will be clocked into the network.

This suggests a hybrid approach Zhang et al. call "pacing" [32]: use acknowledgments to trigger new data to be sent into the network (so that the system will be quickly responsive to increased congestion), but evenly space the actual transmission of new packets at the rate determined by the current window size and round trip time estimate. Relative to TCP, this "slows down" data transmissions – new packets are delayed for some amount of time instead of being sent immediately after the acknowledgment is received. Nevertheless, by more evenly spacing packets and therefore reducing burstiness, pacing has the potential of reducing both packet losses and queueing delays and therefore improving performance. Further, since pacing only delays packets without changing any other aspect of the TCP standard, it can be implemented at the sender (to work with unmodified receivers), at the receiver (by de-clustering acknowledgments, a receiver can cause an unmodified sender to pace data sends), or inside the network (by delaying data or acks, to work with unmodified senders and receivers).

Obviously, pacing is not a new idea. Zhang et al. initially suggested pacing to correct the compression of acknowledgments due to cross traffic. Other researchers have suggesting using pacing when acknowledgments are not available to use for clocking, for example, to avoid TCP slow start at the beginning of a connection [3, 26], after a packet loss [14, 23], or when an idle connection resumes [31]. Similarly, pacing can be used to avoid burstiness in asymmetric networks caused by batching acknowledgments [5]. More recently, researchers have suggested using pacing across the entire lifetime of the connection. Partridge argues that pacing can address problems in TCP performance on long-latency, high bandwidth satellite links [27] while the Berkeley WebTP group has combined pacing, receiver-driven congestion control, and application-level framing into a transport protocol specialized for web traffic [13]. A commercial company, Packeteer, sells a router that conditions traffic by pacing acknowledgments [20]. Despite this widespread interest, however, we know of no systematic evaluation of the impact of pacing on the Internet. The WebTP work reports the result of some initial studies, but it is difficult to determine whether the performance impact is due to pacing or some other aspect of the WebTP protocol.

Our goal in this paper is to quantify the benefits and limitations of using pacing in TCP. Using simulation, we examine the impact of pacing on throughput, fairness, queue size, and drop rate as a function of load, bandwidth-delay product, buffer size, and variable round-trip times. We also examine the impact of sharing a bottleneck link between paced and normal TCP. For the topologies, workloads, and router scheduling policies that we examined, pacing leads to a more fair allocation of bandwidth between flows and in a few cases, pacing improves the performance of unmodified TCP. For example, we show that pacing is significantly better than ack-clocking for single flows using high-bandwidth links with limited buffering; in this case, TCP tends to fall out of its initial slow start phase too early, taking a long time to eventually reach the bottleneck rate.

Our intuition starting this study was that pacing would have the right incentive structure to be adopted by the Internet – it has the potential to be better for both the individual (because packets are less likely to be dropped if they aren't clumped) *and* the network (because competing flows will see less queueing delay and burst losses).

Unfortunately, our intuition proved too simplistic and was wrong on both counts, at least in some cases. First, we show that pacing sometimes has much worse performance than TCP. By evenly spreading traffic out, pacing is "too good" at reducing queueing delays, delaying the point at which congestion is detected. Since in TCP all senders continually increase their traffic in the absence of congestion, pacing can cause "synchronized drops", where flows through the bottleneck experience simultaneous losses. This causes senders to quickly reduce their rate, oscillating the bottleneck between being over-subscribed and under-subscribed. In essence, pacing can have the opposite effect of Random Early Detection (RED) gateways [12], which randomly drop packets to throttle back some sources before congestion becomes extreme. We suggest some approaches to eliminating this effect.

Second, we show that in many cases, pacing performs poorly when competing with unmodified TCP.

Again, the problem is that pacing is "too good" at spreading traffic out. A single paced connection, sharing a congested link with bursty connections, is more likely to have at least one of its packets encounter severe congestion; by contrast, the bursty connections will either be lucky enough to miss other bursts, or unlucky and suffer multiple losses in a burst.

The rest of the paper discusses these issues in more detail. Section 2 briefly discusses some TCP congestion control algorithms and some sources of burstiness in TCP. Section 3 describes pacing. Section 4 describes the simulation methodology and section 5 describes the simulation results. Finally, we conclude in section 6.

2 Background

In this section, we sketch the TCP mechanisms relevant to our discussion (see [15, 16, 28] for further details). We then outline why these mechanisms can cause burstiness. For the purposes of this discussion, we assume the TCP layer picks the size of its transfer unit to avoid fragmentation at the IP layer, so that we can loosely refer to the TCP transfer unit as a packet.

2.1 TCP Mechanisms

TCP is a sliding window-based protocol. The effective window used by a TCP sender is the minimum of the congestion window (`cwnd` – set to match the bandwidth of the network) and the receiver's buffer size. Since the window is the number of bytes the sender is allowed to send without an acknowledgment, the average rate at which traffic enters the network is governed by the window size divided by the round trip time (RTT). The sender uses the incoming acknowledgments to determine when to send new data, a mechanism referred to as ack-clocking [16]. In theory, the network bottleneck will cause any clumped packets to spread out (spaced at the rate the bottleneck serves packets), resulting in well-spaced traffic during the next round trip. However, under a wide variety of conditions, packets from the same connection have been observed to cluster together [32]; we discuss reasons for this in the next subsection.

The congestion window adjustment algorithm has two phases. In the *slow start* phase [16, 28], the sender increases the congestion window rapidly in order to quickly identify the bottleneck rate while at the same time theoretically establishing a stream of well-spaced acknowledgments. The sender typically starts with a window of one packet; each acknowledgment increases the window by 1, effectively doubling the window every round trip time. Assuming the sender is not limited by the receiver's buffer space, the sender increases its congestion window until it detects that a packet loss has occurred; the loss is taken as a signal that the sender is transmitting packets faster than the network can handle. At this point the window is cut in half, and the sender enters the *congestion avoidance* phase. The sender then increases the window by `1/cwnd` on every acknowledgment, effectively increasing it by 1 packet every round trip time. Again, the sender increases the window until it detects a packet loss; the window is cut by half, and the sender resumes increasing the window by 1 packet per round trip.

As a result, the sender's congestion window (controlling the rate at which packets are sent) at first increases along an exponential curve during slow start, then over time repeats a saw-tooth pattern of a factor of two decrease and a subsequent slow, linear increase. Since the receipt of acknowledgments governs the rate of increase in the congestion window, connections with longer round trip times have windows that grow at a slower rate (both slow start and the sawtooth have a proportionately lower slope).

Each TCP acknowledgment reports the highest-numbered packet (more precisely, byte) that had been received along with all earlier packets. Assuming there are no losses, the information in each acknowledgment therefore makes all earlier acknowledgments redundant. As an optimization, TCP receivers try to reduce the number of acknowledgments by delaying returning an acknowledgment for a single received packet (assuming all prior packets had been correctly received) until either another packet arrives or a timer fires. Because half as many acknowledgments are received, and since acknowledgments trigger increases in the congestion window, the principal effect of delayed acknowledgments is to slow the rate at the congestion window is increased: during slow start from a factor of 2 to a factor of 1.5 per round trip and during congestion avoidance from 1 packet to 0.5 packets per round trip.

To simplify our presentation, we ignore the effect of delayed acknowledgments in the remainder of our discussion *and* in our simulations.

2.2 Burstiness in TCP

While ack-clocking in theory causes the sender to spread data out, a number of factors can cause either short or long term burstiness in the behavior of a TCP flow. For this discussion, consider a TCP sender operating over a path with bottleneck bandwidth of b. We define one time unit as the time it takes to transmit one data segment over the bottleneck link. More concretely, $1\ time\ unit = TCP\ packet\ size/b$.

2.2.1 Slow Start.

During slow start, every successfully acknowledged packet increases the window size by one packet. Thus, the sender transmits two packets for every new acknowledgment. Since the acknowledgments are generated at the bottleneck rate, this implies that the sender is bursting data at twice the bottleneck rate, leading to the formation of a queue at the bottleneck link [27]. When the sender window is $W/2$ packets, the sender gets $W/2$ acknowledgements with a spacing of one time unit between consecutive acknowledgments. In response, the sender transmits two packets for each acknowledgment or a total of W packets in $W/2$ time units. Since the bottleneck can forward only $W/2$ packets in $W/2$ time units, the other half of the packets will be queued. Therefore, during slow start, a sender window size of W builds up a queue of size $W/2$ at the bottleneck router. (The queue would be $W/3$ with delayed acknowledgments.)

Since the buffer size at the bottleneck router is necessarily limited, this bursty behavior will cause the router to drop packets when the window size exceeds the router buffer size by some small constant factor. For high bandwidth connections, this may be prior to the point where the sender has reached the bottleneck bandwidth. Ideally, the first loss should not occur in slow start until the window reaches the product of the end-to-end round trip time times the bandwidth, called the delay-bandwidth product. If the router buffer size is much less than this product, the sender will encounter a loss too early and fall out of slow start; it can then take the sender many round-trips of congestion avoidance, increasing the window by only one packet per round trip, to finally reach the bottleneck bandwidth.

2.2.2 Losses

When a lost packet is successfully retransmitted, its acknowledgment may trigger a burst of traffic [11]. At the receiver, the retransmitted packet will typically fill a “hole” in the sequence space, enabling the receiver to acknowledge not just the lost packet but other packets that had been successfully received in the window. When the acknowledgment arrives at the sender, the sender will suddenly be able to send a burst of traffic. Although TCP can use duplicate acknowledgments (acknowledgments for packets delivered after the missing packet) to recover ack clocking, at best the sender transmits new data only during the second half of the recovery period, thereby bursting a window of data in one half of the round trip time [14, 23].

2.2.3 Ack Compression

In the presence of two-way TCP traffic, ack-clocking can be disrupted due to a phenomenon called ack-compression [32]. Acknowledgments from the receiver, generated at the bottleneck rate, can get queued behind data packets on the reverse path. Assuming routers service packets in FIFO order, this can cause the acknowledgments to lose their spacing and reach the sender in a burst. This in turn causes bursty transmissions at the sender.

2.2.4 Multiplexing

Perhaps most importantly, ack-clocking will only spread data at the bottleneck rate. For high-bandwidth links shared by a large number of connections, this can result in each connection transmitting in a burst and remaining idle for the rest of the round trip time. As a connection starts up, its data packets are spaced apart at the bottleneck by one time unit (the rate the bottleneck services packets). Each round trip increases the number of packets, but all of the packets remain clustered together, each spaced by one time unit. Unless two connections happen to send their cluster of packets so that they overlap at the bottleneck, the packets for each connection will tend stay clustered. When two clusters do overlap at the bottleneck (for example, because of slow start increases or if the connections have different round trip times), the result is a larger burst than either would have generated on their own.

3 Pacing

The goal of pacing is to evenly spread the transmission of a window of packets across the entire duration of the round trip time. This can be implemented either by the sender or the receiver. At the sender, instead of transmitting packets immediately upon receipt of an acknowledgment, the sender can delay transmitting packets to spread them out at the rate defined by the congestion control algorithm – the window size divided by the estimated round-trip time. Alternatively, a receiver can delay acknowledgments to spread them across the round trip time, so that when they arrive at the sender, they will trigger spaced data packets. Of course, receiver pacing is less effective, since as we discussed earlier, acknowledgments arriving at the sender can trigger multiple data sends; with receiver pacing, these packets will be sent in a burst. Further, receiver pacing is susceptible to ack compression. Therefore, we only simulate sender-based pacing in this paper.

As a traditional window based protocol, TCP uses a window to determine the number of packets that can be sent and uses the receipt of acknowledgments to trigger the sending of packets. Pure rate based schemes, on the other hand, use rates to determine both *how much* and *when* to send. Pacing is a hybrid between these two approaches – it uses the TCP window to determine *how much* to send but uses rates instead of acknowledgments to determine *when* to send. The idea of pacing has been proposed or used in a number of different contexts [3, 5, 13, 14, 20, 23, 26, 27, 31, 32], but none of these quantify the impact of incorporating pacing into the TCP congestion control algorithm.

3.1 Implementation

We have incorporated pacing into the `ns` [30] simulation code for TCP Reno. The `ns` Reno code closely models the congestion control behavior of most of the TCP implementations in widespread use.

Our implementation of pacing uses a variant of the leaky bucket algorithm [29]. In effect, the algorithm divides the RTT into intervals of duration `RTT/window` and at most one packet is transmitted during an interval. This ensures that packet transmissions are spread across the whole duration of the RTT. As new data is acknowledged (altering the window size) or the RTT estimate changes, the duration of the current and subsequent intervals is suitably altered to adjust to the new rate. We compute the RTT estimate using

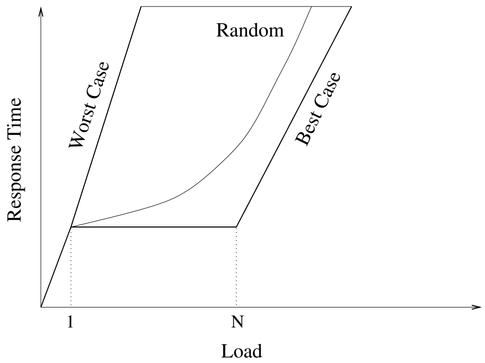

Figure 1: Best and worst case bounds on delay for a queuing system.

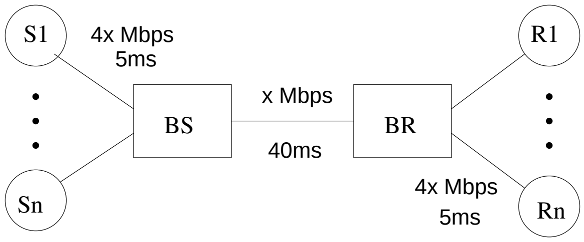

Figure 2: The network topology used for the simulation experiments.

an exponential weighted moving average (EWMA). The TCP timestamp option is used to get accurate RTT samples. Our RTT estimate is separate from the one maintained by TCP flow control algorithm, since we assume a fine-grained timer. Further, we use fine grained timers for sending data at the appropriate rate.

3.2 Why Pacing (Should) Help

One way to understand the impact of pacing is to consider the router from a queueing theory perspective (figure 1). With bursty traffic, packets arrive all at once. As a result, queueing delay grows linearly with load, even when the load is below capacity. TCP, due to its burstiness, can be thought of as being close to this worst case curve. With pacing, traffic is evenly spaced out; so there is minimal queueing until the load matches the bottleneck capacity. The queueing delay increases linearly once the bottleneck is saturated. This represents the best possible time for a queueing system and is theoretically even better than the curve for a random traffic source. However, we argue in the paper that contrary to intuition, delaying queueing until the link is oversubscribed has a negative performance impact on the performance of pacing. TCP uses feedback from the network to detect congestion and adjust to it. With pacing, this feedback is delayed until there is actual congestion, making it difficult for senders to ”avoid” overwhelming the network.

4 Simulation Setup and Methodology

We present performance results using our implementation of pacing in the `ns` network simulator [30]. The topology used for the simulation experiments is shown in figure 2. One or more TCP connections are established between a set of senders and receivers through a single bottleneck link. The bottleneck link router uses FIFO scheduling and drop tail buffer management, unless otherwise specified. Further, by default, the delay on the bottleneck link and the side links is set to $40ms$ and $5ms$ respectively. The bottleneck bandwidth is varied in our experiments and the bandwidth of the side links is set to four times the bottleneck bandwidth. The buffer size at the sender and receiver routers is set to a value much higher than the bandwidth-delay product while that at the bottleneck routers is varied. Similarly, the maximum advertised window size of a connection is set to a high enough value so that the actual window is not limited by it. We use a maximum segment size of 576 bytes.

For simulations, we use the `ns` implementation of *TCP Reno* (which contains slow start, congestion avoidance, fast retransmit and fast recovery) and a paced version of TCP Reno (which we henceforth refer to as *Paced Reno*). The receiver acknowledges every packet.

We evaluate the impact of pacing on the aggregate throughput for all flows as well as individual flow throughputs, drop rate and the average queue size at the bottleneck. We also measure the fairness using a modified version of Jain's fairness index [7]. Jain's fairness index is defined as follows: if there are n concurrent connections in the network and the throughput achieved by connection i is equal to x_i, then

$$f = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$$

Since the rate of increase of a TCP sender's window is dependent on its RTT, we define the fair share of a flow to be inversely proportional to its RTT. Based on this notion of fairness, we compute the normalized fairness ratio for flows with different RTTs as follows:

$$f = \frac{(\sum_{i=1}^{n} x_i.RTT_i)^2}{n.\sum_{i=1}^{n} (x_i.RTT_i)^2}$$

5 Results

We present the simulation results in five parts: section 5.1 compares the performance of TCP Reno and Paced Reno for a single flow. Sections 5.2 and 5.3 present results for multiple flows starting at the same time, with the same as well as with different round trip times. In section 5.4, we describe the results of a more realistic model consisting of multiple senders, with each sender periodically initiating a fixed size flow. In section 5.5, we evaluate the interaction between TCP Reno and Paced Reno. Finally, in section 5.6, we evaluate the effect of the queueing discipline at the bottleneck on the performance of pacing.

5.1 Single Flow

Figure 3 plots the cumulative throughput for a single flow as a function of time. The buffer size at the bottleneck is limited to approximately one-fourth the delay-bandwidth product. In the initial period, pacing achieves much better performance than Reno. This is because TCP Reno, due to its burstiness in slow start, overwhelms the limited buffer at the bottleneck router and incurs a loss earlier and with a lower window size than Paced Reno. For TCP Reno, the first loss occurs when the window size reaches twice the bottleneck buffer size, while Paced Reno gets a loss only after it saturates the pipe. As a result Reno takes a long time in congestion avoidance to ramp up to the bottleneck bandwidth (figure 4). In steady state, however, both Reno and Paced Reno display the same saw-tooth behavior and achieve similar throughput.

In order to study how various factors like buffer size and bandwidth affect the difference in performance of Reno and Paced Reno, we fixed the length of the flow to the duration of one saw-tooth.[1]. Figure 5 plots the difference in the bandwidth achieved by Paced Reno and TCP Reno as we vary the bottleneck bandwidth and buffer size. Two points emerge from the graph. First, the difference in performance increases with the delay-bandwidth product for limited buffer sizes. As networks grow faster, the penalty of falling out of slow start early also increases. Second, the difference disappears when the buffer size grows beyond half the

[1] We compute the cycle time of a saw-tooth wave using the formula $T_{cycle} = RTT^2.BW/packet\_size$ proposed in [21].

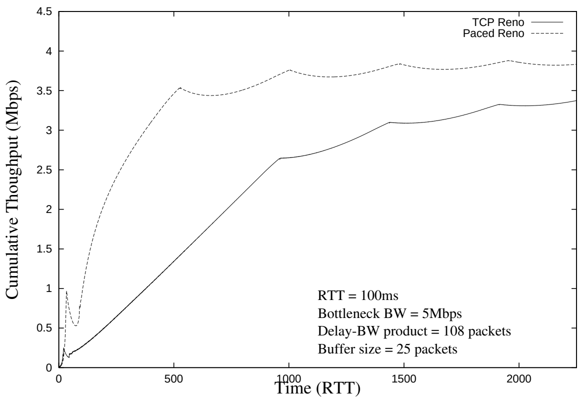

Figure 3: Cumulative throughput for a single flow as a function of time. Note that unlike cumulative data transferred, cumulative throughput decreases if time elapses without forward progress.

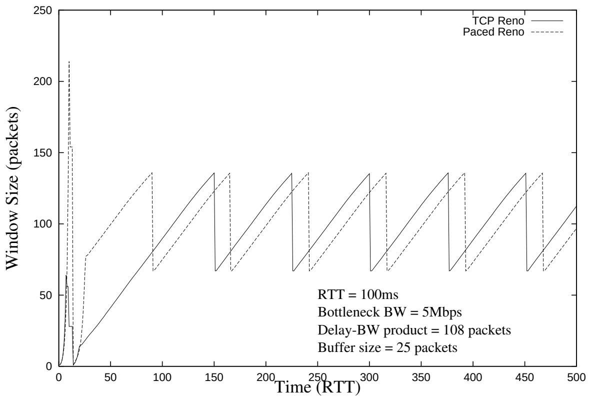

Figure 4: Window size for a single flow as a function of time.

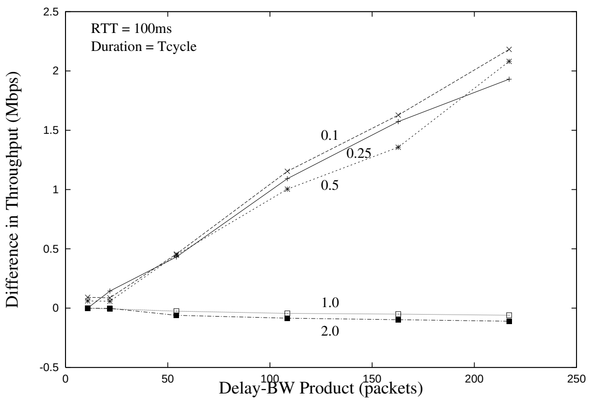

Figure 5: Difference in throughput between Paced Reno and TCP Reno for a single flow as a function of delay-bandwidth product. The different lines are for different ratios between the buffer size and delay bandwidth product.

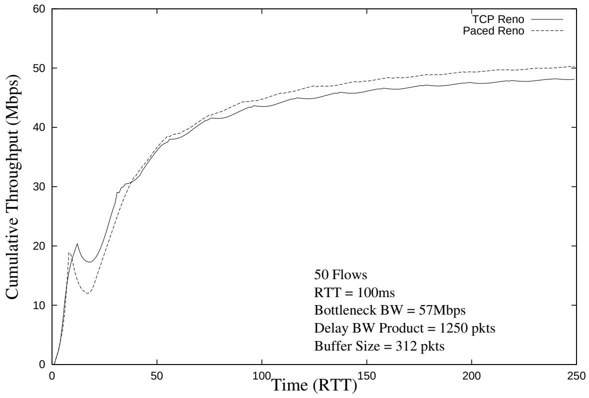

Figure 6: Cumulative throughput for 50 flows as a function of time.

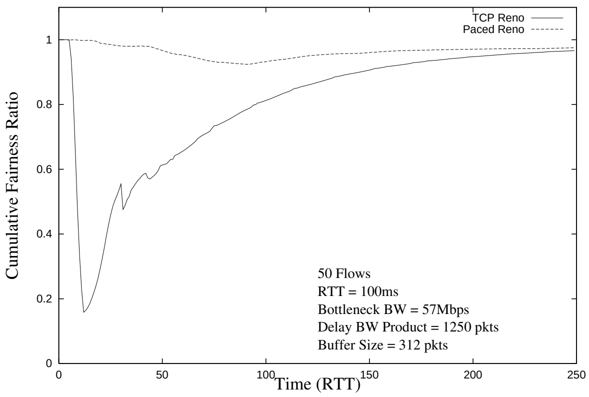

Figure 7: Fairness with respect to cumulative throughput for 50 flows as a function of time.

delay-bandwidth product. In fact, at that point, pacing performs slightly worse, because it lags behind by a round trip time. With large buffer sizes, the bursty behavior of Reno is absorbed by the buffer and it does not get a loss until it reaches the bottleneck bandwidth.

5.2 Multiple Flows

To study how multiplexing affects the performance of pacing, we conducted an experiment in which 50 flows, each with a RTT of $100ms$ and sharing the same bottleneck link, were started at approximately the same time (section 5.4 considers a more realistic model in which start times are not synchronized). The bottleneck bandwidth was set to $57Mbps$, so that the fair share of the delay-bandwidth product of each flow was roughly 25 packets. Figures 6 and 7 plot the cumulative aggregate throughput and the cumulative fairness respectively. With limited buffer size, one would expect Reno flows to experience losses earlier and perform worse than pacing, just as in the single flow case. However, contrary to intuition, during the initial period, TCP Reno achieves better throughput than Paced Reno. Also, Reno is less fair than Paced Reno during this period. On the other hand, during steady state, pacing achieves better throughput than Reno. These two distinct and counter-intuitive phenomena can be explained by the synchronization and de-synchronization effects that pacing has in slow start and steady state respectively.

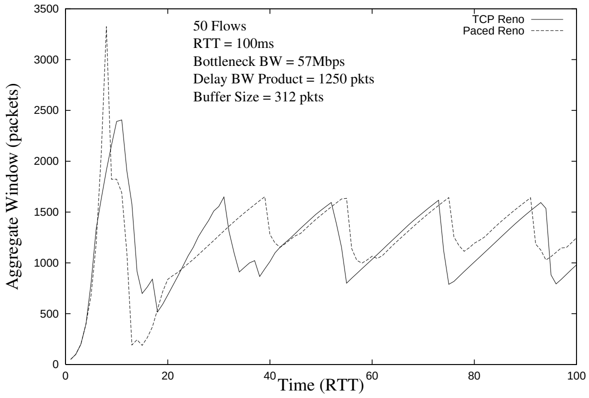

Figure 8: Aggregate window size for 50 flows as a function of time. In slow start, Paced flows synchronize. In steady state, Paced flows have a larger aggregate window owing to de-synchronization.

5.2.1 Synchronization Effect of Pacing

With multiple flows, pacing synchronizes the drops that the various flows experience during slow start. All the flows continue in slow start, till the network is beyond saturation. At this point, everyone drops because of congestion and mixing of flows, thereby making the bottleneck under-utilized. With Reno, flows send bursts of packets in clusters. Due to the limited buffer size, some of the flows drop early (when they "hit" bursts from another flow) and back-off; allowing the other flows to ramp up. Thus, different flows experience drops at different times, thereby utilizing the bottleneck better while sacrificing fairness. Figure 8 illustrates this using the aggregate window size of all the flows. All the paced flows behave as a single flow, reducing their rate at the same time; on the other hand, Reno flows utilize the link better by dropping at different times and spreading out the aggregate window.

There are two main reasons for the synchronization effect in pacing:

1. Late Congestion Signals: Pacing completely spreads out the transmissions of data; as a result, the queue size at the bottleneck router remains negligible until the link is saturated. Therefore, paced flows gets congestion signals only after it is too late causing everyone to drop. With Reno, owing to bursty transmissions, some of the flows experience congestion before the bottleneck is utilized. As the bottleneck buffer size is increased to the delay-bandwidth product, both Reno and Paced Reno experience synchronization; Reno because the bottleneck queue absorbs its burstiness.

2. Mixing of Flows: Data from a single Reno flows is clustered together. Therefore, temporary congestion at the bottleneck does not uniformly affect all the flows. With pacing, traffic from all the flows is thoroughly mixed; therefore, when the bottleneck link is saturated, all the flows experience losses.

5.2.2 De-synchronization Effect of Pacing

In steady state, pacing has a de-synchronizing effect (illustrated by the larger aggregate window in figure 8), leading to slightly higher throughput. Also, this de-synchronization leads to lower fairness over short time periods (duration of one saw tooth); over longer periods the fairness is similar to Reno.

In steady state, each flow increases its window size by 1 packet every RTT. The new packet that is sent every RTT is not ack-clocked in Reno; if the bottleneck queue is full, the packet is dropped. Since, every Reno flows sends at least two packets back-to-back, each flow gets at least one loss when the queue is filled. With pacing, all the packets are spread out and flows are mixed; as a result, there is randomness in the way

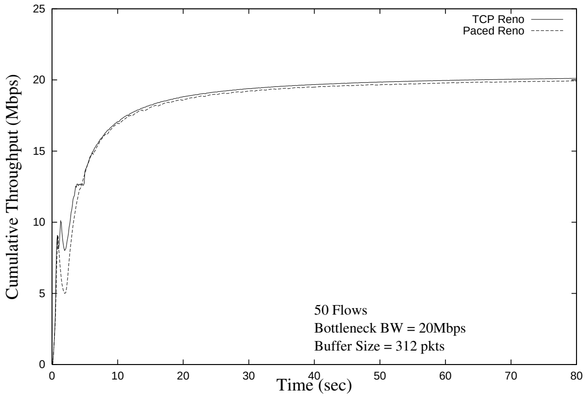

Figure 9: Cumulative throughput for 50 flows. Half of the flows have a RTT of 100ms while the remaining have a RTT of 280ms (these flows have a higher delay on the non-bottleneck links).

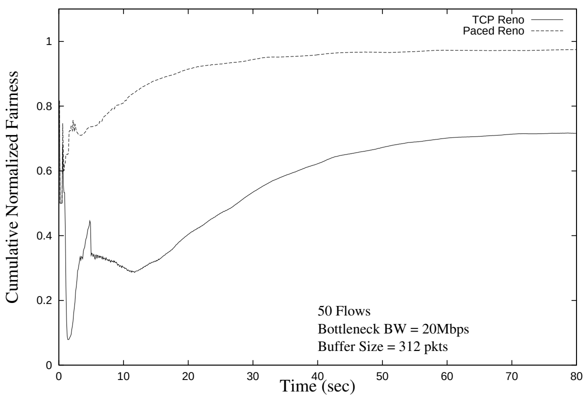

Figure 10: Normalized fairness with respect to cumulative throughput for 50 flows with variable RTTs.

packets are dropped. During a particular phase, some flows might get multiple losses while others might get away without any. This de-synchronizes the windows of different flows, leading to better utilization of the link. This effect persists even when the buffer size is increased to the delay-bandwidth product.

5.3 Multiple Flows - Variable RTT

Up to now, we have considered only flows with the same RTT. This means that even if the individual flows are bursty, the aggregate will tend to be smooth as each flow tries to schedule its own region in the RTT. To eliminate this effect, we simulated 50 flows with different RTTs – half of them have a RTT of $100ms$ while the other half have a RTT of $280ms$. Figures 9, 10 and 11 depict the cumulative throughput, normalized fairness and the drop rate at the bottleneck for this setup.

Figure 10 shows that pacing achieves much better normalized fairness than TCP Reno. This increase in fairness does not come at the cost of performance; as figure 9 shows, both Reno and Paced Reno achieve similar throughput.

As expected, with variable RTTs, the higher burstiness of Reno as a result of overlap of packet clusters from different flows (section 2.2.4) becomes visible. Figure 9 shows that Reno has a higher drop rate (it also forms larger queues at the bottleneck), while achieving similar throughput as pacing. This trend of higher drop rates continues even when the buffer size is increased to the delay-bandwidth product.

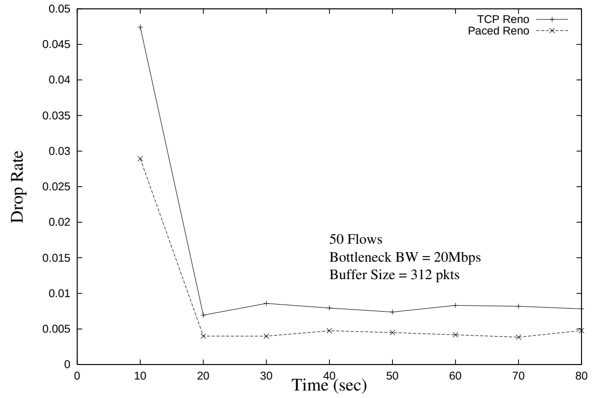

Figure 11: Drop rate for 50 flows with variable RTTs.

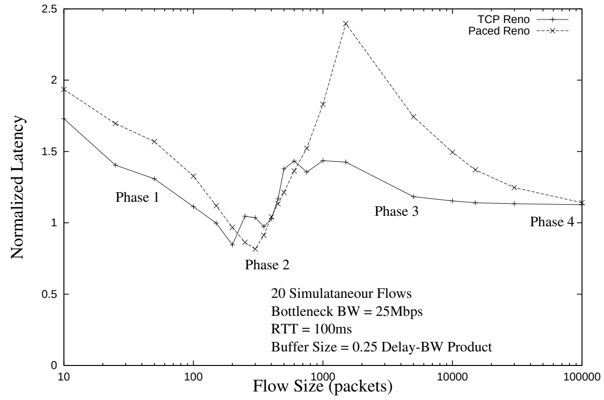

Figure 12: Normalized latency for flows initiated from 20 nodes with a mean think time of 1 sec. The buffer size is one fourth the delay-bandwidth product.

5.4 Variable Length Flows

The scenarios considered so far consisted of flows with unlimited data starting at approximately the same time. In this section, we study a more realistic environment in which new flows enter the system over time. In our experiment, a constant size flow is established between each of 20 sender nodes and the corresponding receiver node. As a particular flow finishes, a new flow is established between the same nodes after a think time that is exponentially distributed with mean $1sec$. Thus, at any time, there are at most 20 flows sharing a single bottleneck link. This model de-synchronizes the start time of various flows and leads to interaction between flows in various stages of completion. We vary the size of the flows in our experiments and use the average latency of completion of flows as the performance metric. In order to compare performance for different flow sizes, we normalize the latency using an estimate of the "ideal latency". We define ideal latency as the latency of a flow that does slow start until it reaches its fair share of the bandwidth and then continues with a constant window. Note that in some case the ideal latency can be worse than the actual latency since a flow can in reality overshoot its fair share; further, ideal latency assumes perfect fairness while in reality, flows can achieve better mean latency at the expense of fairness. Our intent in using ideal latency is only to normalize the latencies to facilitate comparison.

Figure 12 plots the normalized latencies for Reno and Paced Reno as a function of the flow size. The graph can be divided into four distinct phases:

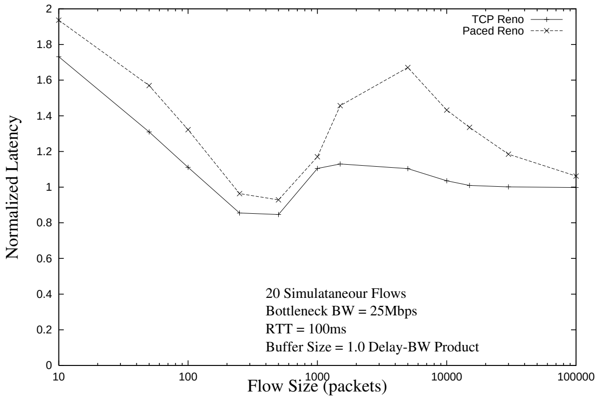

Figure 13: Normalized latency for flows initiated from 20 nodes with a mean think time of 1 sec. The buffer size is equal to the delay-bandwidth product.

1. In *phase 1*, neither Reno nor Paced Reno flows experience any losses. The latency of Paced Reno flows is slightly higher because they spread packets throughout the RTT and hence lag behind a standard Reno sender.

2. In *phase 2*, with a limited buffer size, Reno experiences more losses in slow start because of its bursty behavior. As a result, more Reno flows are forced to timeout before they complete the data transfer and pacing has some performance advantage over Reno. With large buffer size, this effect disappears.

3. The synchronization effect of pacing becomes visible in *phase 3*. New flows starting in slow start, saturate the network. Due to late congestion signals and mixing, all the flows drop packets, including those that are in congestion avoidance. These synchronized drops happen regularly during the duration of a flow, severely diminishing its performance.

4. In *phase 4*, the length of the flows becomes so large that the synchronizing events have an insignificant impact on the overall performance. New flows start very infrequently during the lifetime of a flow.

Even with a buffer space equal to the delay-bandwidth product, pacing shows synchronization effects which reduce its performance (figure 13). Reno performs better because Reno flows send packets in clusters, a burst from a particular flow in slow start only has a local effect; it does not affect all the flows. Also, with the large buffer size, Reno performs uniformally better than pacing because it does not fall out of slow start earlier, thereby eliminating the effect that occurred in *phase 2* for limited buffer size. In section 5.6, we discuss ways of eliminating the synchronization effect of pacing.

5.5 Interaction of Paced and non-Paced Flows

In this section, we present results related to the interaction between Reno and Paced flows. Figure 14 plots the latency of Reno and Paced flows when the two are mixed together. Each flow has a size of 300 packets and starts at the same time. Since pacing spreads out packets throughout the duration of the RTT, a paced flow is very likely to experience a drop as a result of one of its packets landing in a burst from a Reno flow. Reno flows, on the other hand, are less likely to affected by bursts from other flows, as their packets are clustered. As a result, they have much better latency than paced flows, when both are competing for bandwidth in a mixed flow environment.

Figure 15 plots the ratio between the mean latencies of Reno and Paced flows for different scenarios. With a large flow size of 5000 packets, pacing still performs worse. Our hypothesis is that the paced

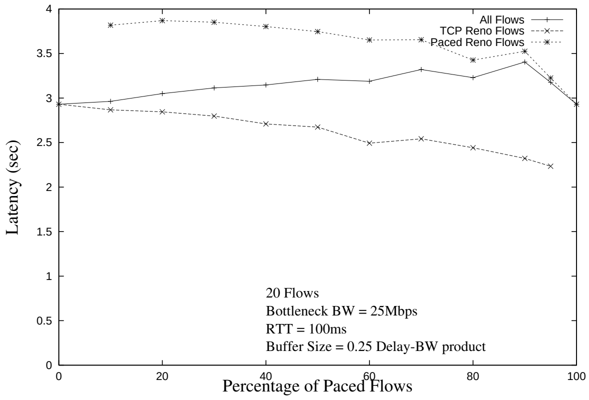

Figure 14: Latency when Reno and Paced flows are mixed. All flows start at the same time and send 300 packets each.

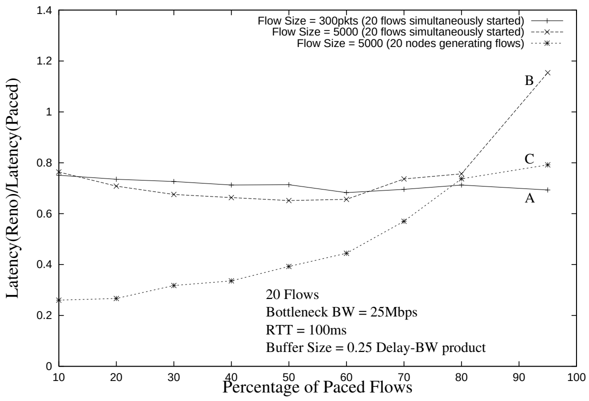

Figure 15: Ratio of the mean latency of Reno and Paced flows when they are mixed. For the lines marked A and B, 20 flows start simultaneously with different flow sizes. For the line marked C, 20 nodes generate flows using the model described in section 5.4. Line A corresponds to figure 14.

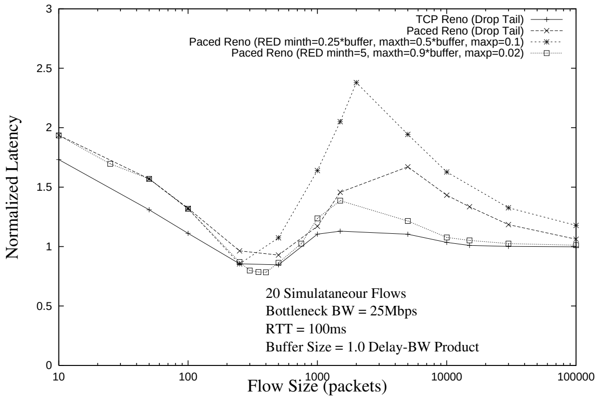

Figure 16: The normalized latency as a function of flow size using different queueing disciplines at the bottleneck.

flows introduce enough spreading out and mixing of packets that in steady state, the flow that gets a loss is randomly distributed and not deterministic. Since Reno flows transmit in bursts, they have a smaller probability of experiencing a drop as opposed to Paced flows which spread their packets uniformally in the whole RTT. The performance of pacing as compared to Reno even deteriorates further when we use the model from the previous section in which new flows start. These new flows in slow start, cause the old paced flows to regularly drop packets, further diminishing its performance.

5.6 Effect of Queueing Discipline

Pacing performs well in a number of scenarios. For links with a high delay-bandwidth product and limited buffering, pacing has the potential for significant performance gains. Even with sufficient buffering, pacing improves fairness and drop rates, especially when the RTTs are variable. However, in some cases, pacing performs significantly worse than the "non-paced" version of TCP. We attribute this degradation to two main characteristics of pacing. First, by evenly spreading traffic out, pacing delays the congestion signals to a point where the network is already over-subscribed. This can have a performance impact, especially if some of the flows are in slow start. Second, owing to mixing of traffic, pacing synchronizes drops, especially with drop tail queues.

Both the above problems can be reduced by intelligent active queue management techniques. Intelligent schemes for choosing the packet to drop coupled with probabilistic dropping of packets (like RED) can alleviate the synchronization problem due to mixing. Further, since, pacing does not build up the queue until it reaches the link capacity, a better trigger for dropping packets might be the link utilization rather than the queue size. BLUE [9] proposes a similar technique; it uses the link idle time and packet losses to manage congestion.

We have evaluated the impact of RED on pacing with different parameter values for RED. Figure 16 shows the normalized latency for the model in the previous section, using RED at the bottleneck buffer. The performance is very sensitive to the choice of parameters. Using high values for the minimum threshold does not perform well. Pacing builds up a queue when the network is already saturated; therefore, with pacing even a small queue size should be interpreted as congestion. In our experience, fixing the min and max thresholds to very low and high values respectively, with a low maximum packet marking probability works best.

6 Conclusions

The idea of pacing is appealing; intuitively, it seems to be better for individual flows as well as the network. In this paper, we have quantitatively evaluated the effect of incorporating it into the TCP congestion control algorithm using extensive simulations. While pacing improves fairness and throughput in some cases, it can have significantly worse performance as compared to Reno in a lot of cases – both with a mixture of paced and non-paced flows and even when all flows are paced. Further, the paper tries to gain intuition into the causes of the counter-intuitive performance of pacing.

Acknowledgments

I would like to thank my advisor, Tom Anderson, for the long hours he put in for this paper and for guiding me at every stage; Stefan Savage for coming up with the idea of performing this study and providing useful suggestions all along and David Wetherall for agreeing to be on my quals committee.

References

[1] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's initial window. RFC 2414, September 1998.

[2] Mark Allman, Chris Hayes, and Shawn Ostermann. An evaluation of TCP with larger initial windows. *ACM Computer Communications Review*, 28(3), July 1998.

[3] Mohit Aron and Peter Druschel. TCP: Improving startup dynamics by adaptive timers and congestion control. Technical Report TR98-318, Rice University, 1998.

[4] H. Balakrishnan and Randy. H. Katz. Explicit Loss Notification and Wireless Web Performance. In *IEEE Globecom Internet Mini-Conference*, November 1998.

[5] H. Balakrishnan, V. Padmanabhan, and Randy. H. Katz. The Effects of Asymmetry in TCP Performance. In *Proceedings of Third ACM/IEEE Mobicom Conference*, September 1997.

[6] F. Bonomi and K. Fendick. The rate based flow control framework for the available bit rate atm service. *IEEE Network Magazine*, pages 25–39, March/April 1995.

[7] D-M. Chiu and R. Jain. Analysis of Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[8] David D. Clark, Mark M. Lambert, and Lixia Zhang. NETBLT: A High Throughput Transport Protocol. In *Proceedings of the ACM SIGCOMM '87 Conference on Communications Architectures and Protocols*, pages 353–359, August 1987.

[9] W. Feng, D. Kandlur, D. Saha, and K. Shin. Blue: A new class of adaptive queue management algorithms. Technical Report CSE-TR-387-99, University of Michigan, April 1999.

[10] S. Floyd and T. Henderson. The newreno modification to tcp's fast recovery algorithm. RFC 2582, April 1999.

[11] Sally Floyd. Simulator tests. Technical Report, July 1995.

[12] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.

[13] Rajarshi Gupta, Mike Chen, Steven McCanne, and Jean Walrand. WebTP: A receiver-driven web transport protocol. Submitted to INFOCOM 99, July 1998.

[14] J. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Masters Thesis, MIT, June 1995.

[15] V. Jacobson. Modified TCP Congestion Avoidance Algorithm. Technical Report, April 1990.

[16] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of the ACM SIGCOMM '88 Conference on Communications Architectures and Protocols*, pages 314–329, August 1988.

[17] P. Karn and C. Partridge. Round trip time estimation. In *Proceedings of the ACM SIGCOMM '87 Conference on Communications Architectures and Protocols*, August 1987.

[18] S. Keshav. A Control Theoretic Approach to Flow Control. In *Proceedings of the ACM SIGCOMM '91 Conference on Communications Architectures and Protocols*, September 1991.

[19] L. Kleinrock. *Queueing Theory*. Wiley, New York, 1975.

[20] Packeteer: Intelligent Bandwidth Management. `http://www.packeteer.com/technology/technology.html`.

[21] M. Mathis, J. Semke, J. Madhavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM Computer Communication Review*, 27(3):67–82, July 1997.

[22] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP selective acknowledgement options. RFC 2018, April 1996.

[23] Matt Mathis, Jeff Semke, Jamshid Madhavi, and Kevin Lahey. The Rate-Halving Algorithm for TCP Congestion Control. Internet Draft, July 1999.

[24] P. P. Mishra and H. R. Kanakia. A hop-by-hop rate-based congestion control scheme. In *Proceedings of the ACM SIGCOMM '92 Conference on Communications Architectures and Protocols*, August 1992.

[25] J. Nagel. Congestion control in tcp/ip internetworks. RFC 896, January 1984.

[26] Venkata N. Padmanabhan and Randy H. Katz. TCP fast start: A technique for speeding up web transfers. In *Proceedings of the IEEE Globecom '98 Internet Mini-Conference*, November 1998.

[27] C. Partridge. ACK Spacing for High Bandwidth-Delay Paths with Insufficient Buffering. Internet Draft draft-rfced-info-partridge-01.txt, September 1998.

[28] W. Stevens, M. Allman, and V. Paxson. TCP congestion control. RFC 2581, April 1999.

[29] Jonathan S. Turner. New Directions in Communications (or which way to the information age?). *IEEE Communications Magazine*, 24(4):8–14, October 1986.

[30] UCB/LBL/VINT. Network Simulator - ns. `http://www-mash.cs.berkeley.edu/ns/`.

[31] V. Visweswaraiah and J. Heidemann. Improving Restart of Idle TCP Connections. Technical Report TR97-661, University of Southern California, November 1997.

[32] L. Zhang, S. Shenker, and David D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two Way Traffic. In *Proceedings of the ACM SIGCOMM '91 Conference on Communications Architectures and Protocols*, pages 133–147, September 1991.