TCP Self-Clocking and Bandwidth Sharing

Allen B. Downey

Abstract—
We propose a simple queueing model for TCP transfers sharing a bottleneck link and examine its behavior when the buffer at the bottleneck is large compared to the bandwidth-delay product. This model explains some behaviors of TCP that have already been observed, and predicts other behaviors that are new. We present measurements that demonstrate these behaviors in the current Internet.

I. Introduction

Researchers have proposed many models of TCP performance, most of which have focused on the steady-state behavior of long TCP transfers [1][2][3][4][5][6][7][8][9][10][11][12][13][14][15]. Most of this work has been based, sometimes implicitly, on the assumption that the available buffer capacity in the network is small compared to the bandwidth-delay product (bdp) for most paths.

This paper investigates the behavior of TCP when buffer capacity is large compared to bdp and the exogenous drop rate is low. By simulation and analysis, we find:

- During slow start, the congestion window (cw) increases exponentially only until $cw > bdp$. After that it increases linearly until it exceeds the slow start threshold (ss). After that, it grows as the square root of time. This derivation explains observations of sub-linear growth reported by Altman et al. [11].
- Long transfers can enter a steady state in which cw grows without inducing dropped packets. We call this state SC, for "self-clocking." Self-clocking is a packet transmission pattern, identified by Jacobson [16], in which the send rate is limited by the receipt of ACKs, rather than by the send window. We derive the conditions that make SC possible and describe transitions between slow start, congestion avoidance and SC.
- Long transfers that share a bottleneck can enter a steady state in which they tend to transmit periodically, their congestion windows grow, and the proportion of sharing is independent of the base round trip times of the paths (rtt). We call this state QS, for "queue sharing", and derive the relationships between ss, the send and receive buffers, the period length and the proportion of sharing.

We focus on low degrees of multiplexing on the assumption that for many Internet paths, the bottleneck links are near the edges, not in the core. If the number of hosts behind a bottleneck is low, and the great majority of transfers are short, then the number of long transfers at a given bottleneck at a given time is likely to be low.

To validate the model, we observe long (100–2000 KB) transfers between three client sites and more than 200 Web servers. On the majority of paths, we find at least one transfer that exhibits self-clocking behavior, and on low-bdp paths we find that SC is the most common state. We also observe simultaneous transfers that share a bottleneck, and find many cases that demonstrate queue sharing. We conclude that the conditions required by the model are common in the current Internet, at least on some paths.

The primary contribution of this work is that it expands the scope of existing models to include a domain that is common in the current Internet but previously ignored. In this domain, long TCP transfers demonstrate some undesirable behaviors: they tend to induce persistent queues, and in some cases they respond to a dropped packet by pausing and then resuming at a slightly higher send rate, rather than cutting the send rate. Other behaviors we observed are not necessarily bad, but different; for example, in this domain TCP does not obey the $1/\sqrt{p}$ and $1/rtt$ heuristics that characterize AIMD congestion avoidance. These observations improve our understanding of TCP performance and suggest opportunities for improvement.

Section 2 presents our network model and derives the requirements for a transfer to get into and stay in SC. In Section 3 we extend the model to describe the steady-state behavior of multiple transfers sharing a bottleneck link. Section 4 presents measurements we made to validate the model. In Section 5 we discuss some implications of our findings for TCP performance.

We use the following notation:

cw	congestion window (packets)
ss	slow start threshold (packets)
rtt	round trip time (seconds)
tt	transfer time per packet (seconds)
bw	bottleneck bandwidth (packets/s)
bdp	bandwidth delay product (packets)
sb	size of receive buffer (packets)
rb	size of send buffer (packets)
lb	limiting buffer = $\min(sb, rb)$
bb	bottleneck buffer (packets)
k	packets between drops
p	drop rate (1/k)
th	effective throughput (packets/s)
mss	maximum segment size (bytes)

In common use, "round trip time" and "bandwidth-delay product" are ambiguous; they may or may not include queue delays. In this paper, rtt is the minimum round trip time of a path, sometimes called "base rtt". Similarly, the "delay" in bdp is the round trip time excluding queue delays. By these definitions, rtt and bdp are fixed for a given path and do not depend on current traffic.

II. Self-clocking

In this section we define self-clocking and use a simple network model to demonstrate the transition from slow start to a self-clocking steady state (SC). In turn, we consider the effect on a connection in SC of the slow start threshold, ss, and the limiting send/receive buffer, lb; delayed ACKs, dropped packets, and sharing with short transfers. In Section III, we explore the behavior of multiple long transfers sharing a bottleneck.

Olin College of Engineering, Needham, MA 02492, email: downey@allendowney.com, web: www.allendowney.com

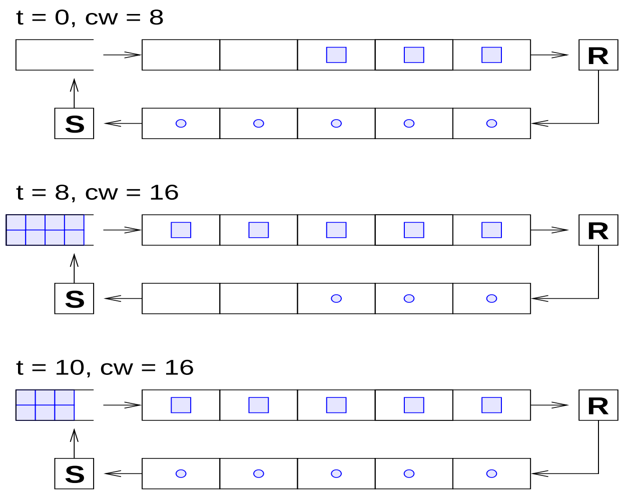


Fig. 1. Transition from slow start to self-clocking.

A. Transition to SC

Figure 1 shows a hypothetical network path, represented by a pipeline with bandwidth-delay product $bdp = 10$ packets and a queue that drains 1 packet per time step. The queue represents the router before the bottleneck link. This example is simplified by putting the bottleneck close to the sender; we relax this assumption below.

Given bdp and the initial congestion window, we define cw^* as the largest slow start congestion window smaller than bdp. In the example, $cw^* = 8$ packets. After the sender transmits these 8 packets, we define as $t = 0$ the moment before the first ACK reaches the sender.

As each ACK arrives, the sender increases cw and transmits two packets, one to replace the just-acknowledged data and one to fill the just-expanded congestion window. Thus, by $t = 8$, cw is 16 and 8 packets have accumulated in queue. During the next two time steps, the queue drains slightly, and then the connection reaches self-clocking. During each time step, the congestion window increases by 1, the sender transmits 2 packets, the receiver gets 1 packet, and the queue grows by 1 packet. During each rtt, cw increases by bdp. Even though the sender is in slow start, the congestion window grows linearly.

The same analysis applies when the bottleneck doesn't happen to be near the sender. Figure 2 shows a more general model of a network path. The box labeled S is the sender; R is the receiver. The e_i are the minimum latencies of a packet on the indicated subpath and tt is the transmission time of a packet on the bottleneck link. The queue at the bottleneck is modeled explicitly; other queue delays are modeled as random variables q_i (see Section III-A). These random delays are meant to capture the effect of short connections (connections that don't leave slow start).

We have written a simulator that implements this network model and a model of TCP that includes slow start, `ssthresh` and fast recovery (with window inflation). It does not implement delayed ACKs; the receiver acknowledges every packet. It doesn't implement a retransmit timer, so it is not accurate for simulations with long delays or high drop rates. We assume that the other links are significantly faster than the bottleneck, so

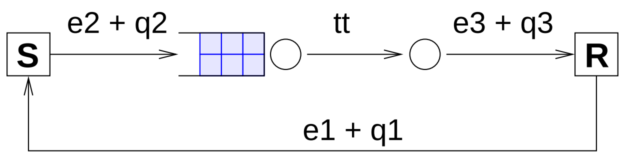


Fig. 2. A simple model of a network path.

that the transmission time on other links is small (1 μs). Since we are considering long transfers, data packets are presumed to be mss bytes. ACKs are presumed to be small. The simulator is available from `allendowney.com/research/tcp`. In addition to our own simulator, we also use the network simulator `ns-2` [17] to validate our simulations and to investigate the effects of delayed ACKs (see Sections II-D and III-B).

Figure 3 shows the result of a simulation with $tt = 20$ ms, $e_1 = 100$ ms, $e_2 = 30$ ms, and $e_3 = 50$ ms. Since tt is 20 ms and rtt is 200 ms, bdp is 10 packets. The line labeled "data" shows the total data received versus time, "cw" shows the congestion window, and "queue" shows the queue, all measured in packets. The marks at the top of the figure show when packets are sent (if two more more packets are sent back-to-back, the marks are stacked vertically). We show total data rather than send rate because in a window-based system, send rate is not well defined. We could plot the average send rate over an interval, but this averaging would obscure important features that are clearly visible if we plot total data.

During slow start, packets arrive in discrete rounds, with the number of packets per round increasing geometrically. After $cw > bdp$, packets arrive at the bottleneck rate, so the receiver ACKs data at the bottleneck rate and cw grows only linearly.

At this point, the transfer is in a self-clocking state we abbreviate SC. **The definitive characteristic of SC is that $\boldsymbol{cw > bdp}$.** The effect of SC is that the send rate is limited by the receipt of ACKs, which reflect the bottleneck bandwidth, rather than by the send window.

B. The effect of ss and lb

In the previous section, we ignore the effect of ss and lb. In this section we ignore their effect; we assume that $bdp < ss < lb$, and then relax this assumption below.

We have seen that when cw exceeds bdp, the transfer enters SC. If no packets are dropped, cw grows linearly until $cw > ss$. At that point, the sender switches from slow start to congestion avoidance.

To derive the behavior of the congestion window in SC after $cw > ss$, we define a variable, x, that counts arriving ACKs, and a function $cw(x)$ that represents the congestion window (in packets) as a function of x. For each ACK, the congestion window grows by $1/cw$, so we can write:

$$\frac{d}{dt}cw(x) = \frac{1}{cw} \quad (1)$$

Then we can solve for $cw(x)$ with initial condition $cw(0) = a$.

$$cw(x) = (2x + a^2)^{1/2} \quad (2)$$

So cw grows as the square root of x. We validate these equations in Section II-D.

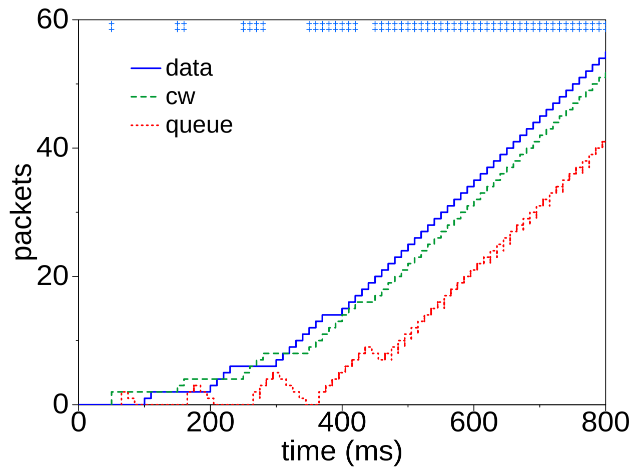


Fig. 3. Simulation of a self-clocking transfer. Although the sender is in slow start, cw and the queue grow linearly.

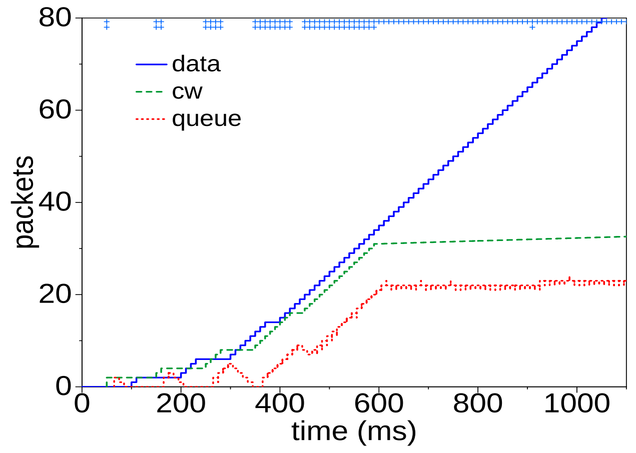


Fig. 4. The effect of ss. After $cw > ss$, the queue and cw grow as the square root of time.

If a transfer is in SC, ACKs arrive at the sender at intervals of roughly tt. In that case, real time t is approximately $x \cdot tt$, which means that cw grows as the square root of time. This relationship does not generally hold for transfers in congestion avoidance, because in that case ACKs don't arrive at regular intervals. But it does hold for large values of cw, because when $cw > bdp$ the ACK arrival rate is determined by the bottleneck bandwidth, not cw.

This result explains some previous observations; for example, Altman et al. report that they observe "paths where TCP window growth is sub-linear" [11]. They suggest that this behavior is caused by increasing queue delays. Appenzeller et al. also note this sub-linearity, but they also attribute it to increasing queue delays [18]. Our model suggests that this behavior is a result of self-clocking, that the sub-linear function is specifically the square root, and that this behavior occurs even if the queue delay is not increasing.

Figure 4 shows a simulation in which ss is 30 packets, so the sender switches to congestion avoidance after 30 ACKs. At that point, the send rate drops from $2bw$ to a little more than bw. Afterward, cw and the queue grow slowly.

Eventually cw may exceed the limiting buffer, lb, which is the smaller of the send and receive buffers. At that point the number of packets in flight is limited by lb, so the queue stops growing, but cw continues to grow.

Many TCP implementations impose an upper bound cw, sometimes called `cwnd_max`. This is a system-level parameter that applies to all connections for a given host, so it is generally larger than lb.

To summarize, we identify four phases a TCP connection can go through:

Phase	Condition	cw growth	Queue growth
1	$cw < bdp$	exponential	exponential
2	$bdp < cw < ss$	linear	linear
3	$ss < cw < lb$	square root	square root
4	$cw > lb$	square root	no growth

During Phases 1 and 2, the sender is in slow start; during Phases 3 and 4 the sender is in congestion avoidance. SC can occur during Phases 2, 3, and 4.

A connection only passes through all phases if $bdp < ss < lb$, which is not always the case. In many TCP implementations, the default value of ss is lb [19], so Phase 3 may be rare.

If $bdp > ss$, the sender switches to congestion avoidance before the transfer reaches SC. It is still possible for the transfer to reach SC, but it may take longer. We see some examples of this in our measurements. If $bdp > lb$, SC is possible in the sense that $cw > bdp$, but the average send rate is limited by lb/rtt.

Discussion of TCP has tended to focus on exponential growth in Phase 1, neglecting the effect of self-clocking. For example, Brakmo and Peterson claim that TCP "*needs* to create losses to find the available bandwidth.[20]" Allman and Paxson write "For TCP, this estimate is currently made by exponentially increasing the sending rate until experiencing packet loss" [21]. Barakat and Altman write "Due to the fast window increase, [slow start] overloads the network and causes many losses" [22]. A casual reader might conclude that long TCP connections inevitably induce drops. Fortunately for the Internet, TCP connections have several opportunities to slow down before inducing a dropped packet.

C. *Conditions for SC*

A transfer can only reach SC if the bottleneck buffer, bb, is large enough to absorb the excess packets. At the beginning of Phase 2, when $cw = bdp$, there are bdp packets in flight. If the packets are equally spaced along the path, there might be only one packet in the bottleneck queue, but in the worst case $bb = bdp$ may be necessary for a transfer to reach SC.

Similarly at the beginning of Phase 3, when $cw = ss$, there are ss packets in flight. In the worst case, $bb = ss$ may be necessary, but if packets are equally spaced, Phase 3 is possible with as little as $bb = ss - bdp$. If $bb > ss$, Phase 3 can continue for a long time. For example, if $ss = 20$ packets and $bb = 100$ packets, the sender can transmit 4800 packets, or 6.9 MB, in Phase 3 (from Equation 2).

Finally, to reach Phase 4, a transfer might need $bb = lb$ in the worst case, or $bb = lb - bdp$ in the best case. Thus if $lb = bdp$, it is possible for a transfer to stay in SC indefinitely with only a minimal bottleneck buffer.

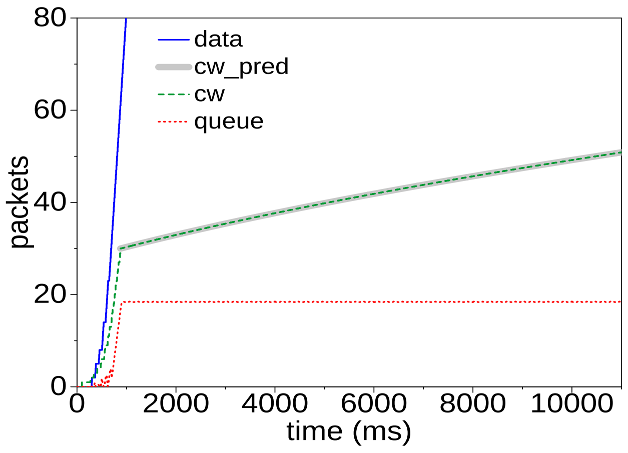


Fig. 5. SC with delayed ACKs (ns-2 simulation). The gray line shows the growth of cw predicted by Equation 2.

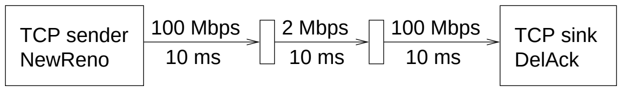


Fig. 6. Configuration for ns-2 simulation.

In Section IV we present measurements that demonstrate that the conditions for SC are common on at least some paths in the current Internet. We conclude that awareness of SC is important for our understanding of TCP performance.

These observations suggests a new category of transfer size: in addition to "mice", which never leave slow start, and "elephants", which are governed by congestion avoidance, we suggest the name "capybara" to describe transfers that are big enough to get into SC but small enough to terminate before overflowing the queue. The capybara is the largest known rodent, a mouse so big it looks like an elephant.

D. *The effect of delayed ACKs*

So far our simulations have been based on a simplified model of the network and TCP behavior. To validate these simplifications, we run a more detailed simulation using ns-2. Figure 6 shows the configuration for this simulation: there are four nodes and three links. The latency for each link is 10 ms, so rtt is 60 ms. The bottleneck bandwidth is 2 Mbps, so bdp is 15,000 B or 10 packets. The receive buffer, rb, is 30 packets. By default, ns-2 sets the initial value of ss to rb, as do many implementations [19]. The bottleneck buffer, bb is 30 packets, which satisfies the requirement for SC that $bb > lb - bdp$. The sender simulates the behavior of TCP NewReno; the sink simulates delayed ACKs.

Figure 5 shows the result of this simulation. At $t = 620$ ms, cw exceeds bdp and the transfer makes the transition to SC without inducing a dropped packet. At $t = 1000$ ms, cw reaches ss and the sender switches to congestion avoidance. After that, cw grows as the square root of time. The gray line labeled "cw_pred" shows the values of cw predicted by Equation 2. The maximum difference between the predicted values and simulated values is 0.06%, which shows that the continuous differential equation (Equation 1) is a good approximation

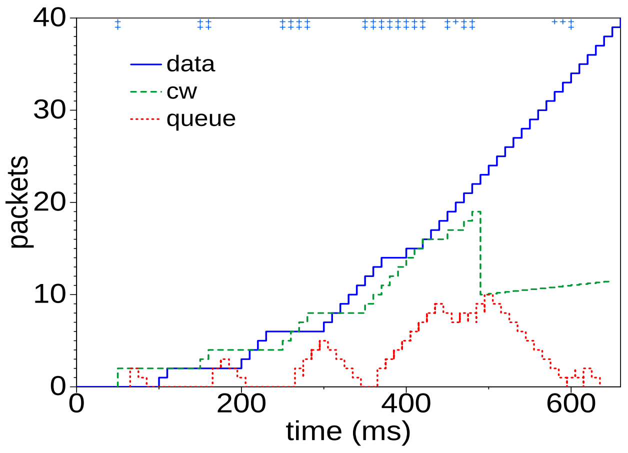


Fig. 7. Self-clocking after a drop. SC can continue if there are enough packets in queue.

of this packet-based system. It also shows that our simulator is consistent with ns, which suggests that our simplified model is sufficient to describe the behaviors we are studying, and that our simulator is a correct implementation of our model.

After $t = 1000$ ms, the queue does not grow because the number of packets in flight is limited by lb, not cw. Therefore this transfer will never induce a drop, and SC can continue indefinitely.

E. *The effect of dropped packets*

To understand the effect of dropped packets, it is useful to distinguish between endogenous and exogenous drops. Endogenous drops are caused by the transfer itself, usually by filling the bottleneck buffer. Exogenous drops are caused by cross traffic or transmission errors.

To reach SC, a transfer needs enough buffer space to avoid endogenous drops (see Section II-C) and enough luck to get through Phase 1 without an exogenous drop. It takes bdp packets without a drop to reach self-clocking, so if the drop rate is p and exogenous drops are independent, the chance of reaching SC is $(1-p)^{bdp}$. For example, if $bdp = 10$ packets and the drop rate is 5%, the chance of getting into SC is 60%. If drops come in bursts, the probability of SC is higher—more transfers suffer multiple drops, but more are spared altogether.

Generally, a transfer can stay in SC as long as cw doesn't fall below bdp long enough to drain the queue. Thus, the longer a transfer has been in SC, and the higher the value of ss, the more likely it is to withstand a small number of dropped packets.

Figure 7 shows a transfer that drops the 15th packet, which is the earliest drop that allow the transfer to stay in SC. When the third duplicate ACK reaches the sender, cw is cut from 19 to 9.5 packets, but there are enough packets in flight to allow cw to reach bdp again before the queue drains.

As this example shows, a transfer in SC does not display the behavior expected in AIMD congestion avoidance. Instead of reducing its send rate when a drop is detected, a self-clocking transfer only pauses long enough to reduce the queue, and then resumes at the same send rate (actually slightly higher).

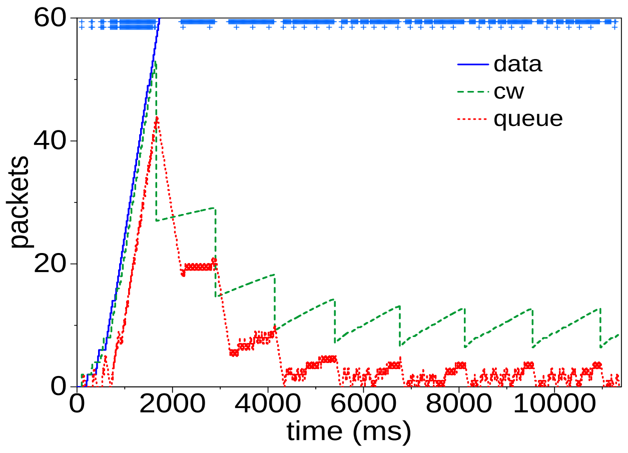


Fig. 8. The effect of periodic drops.

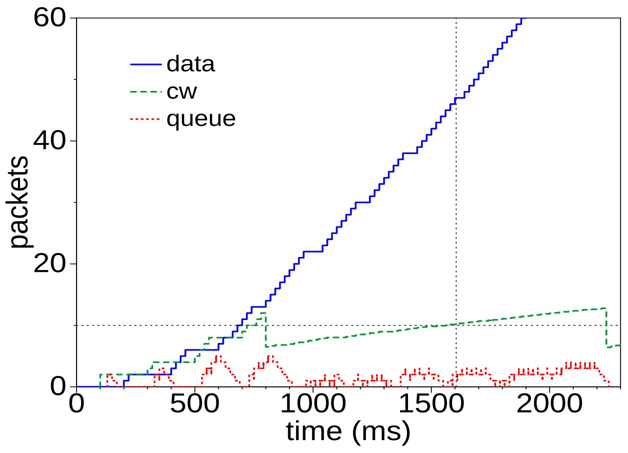


Fig. 9. The transition from congestion avoidance to self-clocking after a drop.

F. The effect of periodic drops

As long as a transfer stays in SC, throughput is strictly bandwidth-limited. But there is a wide range of conditions in which transfers switch back and forth between congestion avoidance and SC; to understand average throughput, we have to take both into account. Two questions we can ask are

- What is the maximum drop rate that keeps a transfer in SC?
- When does bandwidth (as opposed to drop rate) have a limiting effect on throughput?

We start by considering a path that drops every kth packet (in the next section, we consider random drops). Given k, we can find the minimum value of cw in steady state, which we call a—this is the same a that appears as the initial condition in Equation 1. Then we can compute the time it takes for $cw(x)$ to recover from a dropped packet; that is, the time to climb from $cw = a$ to $cw = 2a$. Solving Equation 2 with $cw = 2a$ yields

$$x = 3a^2/2 \tag{3}$$

In equilibrium, $x = k$ and

$$a = \sqrt{2k/3} \tag{4}$$

Figure 8 shows how a transfer converges on this equilibrium. The simulator drops the 30th packet and every 62nd packet thereafter. The analysis predicts that the transfer will be in equilibrium when $a = (2k/3)^{1/2} = 6.43$ packets. In the simulation, the transfer converges on $a = 6.40$.

k_{min} is the minimum value of k that keeps the transfer in SC, which happens when $a \geq bdp$. We find k_{min} by solving $\sqrt{2k/3} = bdp$, which yields $k_{min} = 3bdp^2/2$. This is bad news, because it means that as bdp increases, the drop rate required for SC decreases quickly. For example, if $bdp = 10$ packets, $k_{min} = 150$ packets and the maximum drop rate is $p_{max} = 1/k_{min} = 0.67\%$.

However, as long as $2a > bdp$, the congestion window will sometimes exceed bdp, and so the transfer will sometimes be in SC. We can use this observation to compute steady state throughput as a function of drop rate.

It is well known that in congestion avoidance, the average steady-state throughput is proportional to $1/\sqrt{p}$ [1][5][8], but this result is based on the assumption that cw never exceeds bdp. This assumption is only true when $k < 3bdp^2/8$. For larger values of k, we have to take into account bandwidth limitation when $cw > bdp$. Thus, in the range $3bdp^2/8 \leq k \leq 3bdp^2/2$, the average throughput depends on both the drop rate and the bottleneck bandwidth.

To estimate this throughput, we start by considering a single interval between drops. Given the initial congestion window cw_0 and the number of packets until the next drop, k, we can compute the average throughput during the interval, th.

First, we need to know when the transition from congestion avoidance to SC will occur; that is, when cw reaches bdp. Given cw_0, we solve $cw(x) = bdp$ for k_{ca} which is the number of ACKs the sender gets during the congestion avoidance phase.

$$k_{ca} = (bdp^2 - cw_0^2)/2 \tag{5}$$

Next we need to know how much time, t_{ca}, passes before the sender gets k_{ca} ACKs. To do that we estimate average throughput while cw grows from cw_0 to bdp. In general, as cw grows from cw_0 to a given value cw_1, its average value is

$$cw_{avg} = \frac{1}{k}\int_0^k cw(x)dx = \frac{2(cw_1^3 - cw_0^3)}{3(cw_1^2 - cw_0^2)} \tag{6}$$

Since we know that the transfer is in congestion avoidance during this part of the interval, we expect the average throughput to track cw_{avg}, with one adjustment: since cw is an upper bound on the data in flight, and TCP tries to send maximum-size packets, the actual data in flight may be up to one packet less than cw, and on average we expect it to be one-half packet less. Therefore,

$$t_{ca} = bdp * k_{ca}/(cw_{avg} - 1/2) \tag{7}$$

where the unit of t_{ca} is the transmission time of a packet at the bottleneck[1].

After the first k_{ca} packets, the transfer is back in SC, so the remaining time, $t_{sc} = k - k_{ca}$. During the entire interval, $k - 1$

[1]Since we measure time in units of tt, it is convenient to equate expressions of time and packets without multiplying the units explicitly. Strictly, the time to transmit k packets at the bottleneck rate is $t = k$ packets multiplied by (1 transmission time / packet); for simplicity we write $t = k$.

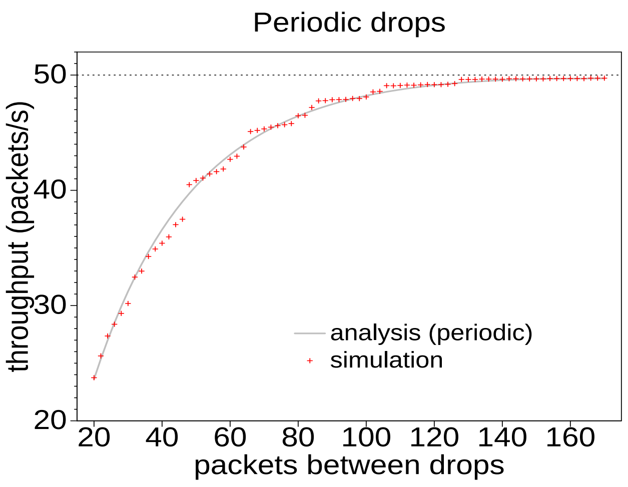


Fig. 10. The effect of periodic drops on throughput.

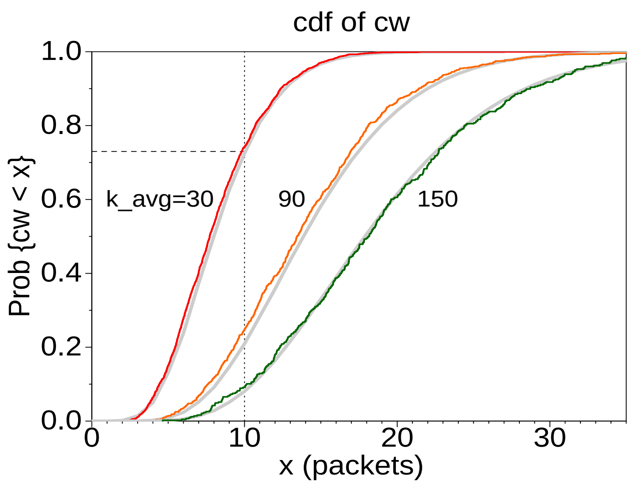


Fig. 11. Distribution of cw as a function of k_{avg}. The analysis and simulation show good agreement.

packets are transmitted in total time $t = t_{ca} + t_{sc}$. So the average throughput in packets per rtt is

$$th_{avg} = bw * (k - 1)/(t_{ca} + t_{sc}) \tag{8}$$

Figure 9 shows a transfer with $k = 62$. It starts by dropping the 9th packet, chosen for purposes of illustration because it puts the transfer into equilibrium immediately. The horizontal line is at $bdp = 10$ packets; the vertical line shows the time when the transfer gets back to SC as computed by Equation 7, so this example is consistent with our analysis.

Since k_{ca} is 29 packets and cw_{avg} is 8.3 packets per rtt, t_{ca} is 37.4 (in multiples of tt). The time in SC, t_{sc}, is $62 - 29 = 33$ time units, so th_{avg} is 43.3 packets/s. In the simulation, the steady-state throughput is 43.0 packets/s, so this example is also consistent with Equation 8.

Figure 10 shows that this analysis agrees with the simulation over the relevant range of k, from $3bdp^2/8 = 37.5$ to $3bdp^2/2 = 150$, which corresponds to drop rates from 2.7% to 0.7%. The plateaus in the simulation results are due to the discreteness of packets. Each simulation runs for 2000 packets and computes the average throughput in steady state (from the fifth drop until the end).

G. The effect of random drops

So far, we have been assuming periodic drops. In this section, we extend our analysis to random drops and derive the relationship between throughput and drop rate in the domain where transfers vacillate between SC and congestion avoidance.

Again, it is useful to think of a transfer as a series of intervals between dropped packets. If the drop rate is p and we assume that drops are uncorrelated, we can compute the distribution of the interval lengths. For a given interval, if we know the initial congestion window cw_0 and the interval length, we can compute the final congestion window cw_1. This suggests an iterative process for approximating the distribution of cw, as observed at the time of a dropped packet.

Given an estimate of $pdf(cw)$, we can generate an improved estimate by the following procedure:

1. Choose cw_0 from pdf.
2. Choose k, the interval between drops, from a geometric distribution with parameter p and $k_{avg} = 1/p$.
3. Compute $cw_1 = (2k + cw_0^2)^{1/2}$.

Repeating these steps, the distribution of the computed cw_1 forms an improved estimate of pdf.

This process can be made more efficient by treating the values of pdf as weights to be distributed over the improved estimate pdf' according to the distribution of k. Furthermore, we can compute the distribution of k implicitly by repeatedly dividing $pdf(cw)$ by p. Here is the algorithm:

```
update_pdf (p):
    pdf' = 0
    for i in domain(pdf):
        w = pdf[i]
        cw0 = i / 2
        apply_weight (w, cw0, p)
```

Given the initial estimate pdf, `update_pdf` forms an improved estimate, pdf', based on the drop rate p. The function apply_weight distributes the weight w over `pdf'`, given cw_0 and the distribution of k implied by p.

```
apply_weight (w, cw0, p):
    for k in domain(pdf):
        j = (int)(sqrt (2*k + cw0*cw0))
        d = p * w
        pdf'[j] += d
        w -= d
```

To generate an initial estimate for the distribution of cw, we can use Equation 6 to compute the average congestion window for a given drop rate, $\mu = (14/9)\sqrt{2k_{avg}/3}$. Then we use a normal distribution with parameters μ and $\sigma = \mu/3$ as the initial estimate for pdf. Over the relevant range of drop rates, the update algorithm converges after 2 iterations. Each iteration takes time proportional to n^2, where n is the number of discrete values in the domain of pdf.

Figure 11 shows the computed distributions of cw for a range of k_{avg}, along with the distribution of cw from a simulation of a long transfer (64000 packets). In general there is good agreement.

The vertical line in the figure is at $bdp = 10$ packets. Even when $k = 30$ ($p = 3\%$), $cw_0 > bdp$ more than a quarter of the time, as indicated by the dashed line. **Thus, bandwidth limitation is relevant to TCP performance even when the drop rate is relatively high.**

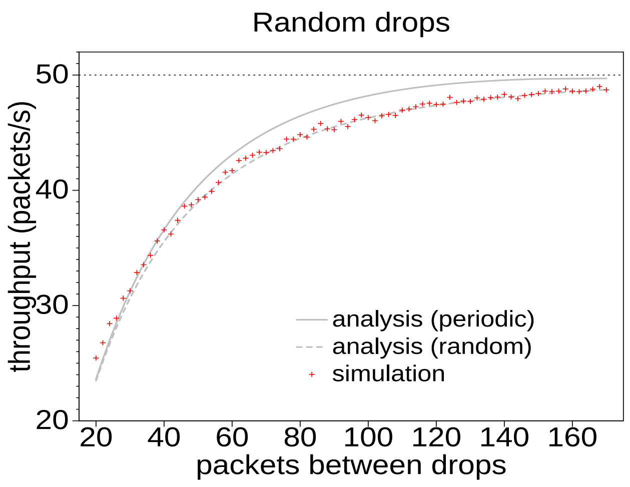


Fig. 12. The effect of random drops on throughput.

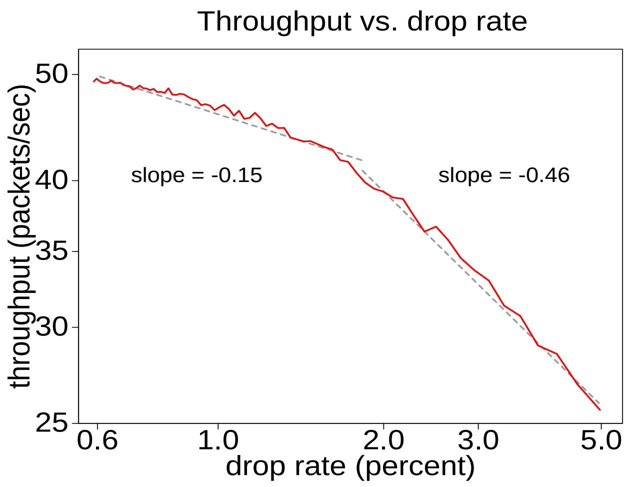


Fig. 13. Throughput and drop rate on a log-log scale. For low drop rates, $th \sim p^{-0.15}$.

Next we compute the average throughput of a transfer with a given distribution of cw. In an interval between drops, if we know the initial congestion window, cw_0, and the number of packets before the next drop, k, we can compute the duration of the interval and average throughput during the interval (see Section II-F). Then, since we know the distributions of cw_0 and k, we can estimate the average throughput of a long transfer by Monte Carlo simulation:

1. Choose cw_0 from $pdf(cw)$.
2. Choose k from the distribution of interval lengths.
3. Compute th and t using the technique in Section II-F.

The computed values of th give us the distribution of throughputs in each interval. In order to compute the time average, we have to weight each th by the duration of the interval, t. This process can be made more efficient by the same algorithm we used to estimate $pdf(cw)$. It runs in time proportional to nm, where n is the number of discrete values for cw and m is the number of discrete values for th.

We believe that this algorithm is equivalent to formulating a Markov model for the congestion window and solving the resulting system numerically.

Figure 12 shows the computed throughput for a range of values of k_{avg} along with the measured throughput of a simulated long transfer (64000 packets). Throughout the range, there is good agreement between the analysis and simulation.

This analysis doesn't yield a simple relationship between throughput and drop rate, but we can look for an empirical relationship. Previous results suggest that $th \sim p^{\beta}$, with $\beta = -0.5$. To estimate β, we took the results from Figure 12 and plotted $\log(th)$ versus $\log(p)$. Figure 13 shows the result with a piecewise linear fit estimated by least squares.

In the range $p =$ 1.8% to 5% ($k_{avg} =$ 20–56), the curve fits a line with $\beta = -0.46$ ($R^2 = 0.99$), which is consistent with the $1/\sqrt{p}$ heuristic. For lower drop rates, $p =$ 0.6% to 1.8% ($k_{avg} =$ 56–160), throughput is limited by bw and the curve flattens, with $\beta = -0.15$ ($R^2 = 0.95$). Below $p =$ 0.6%, the throughput is essentially bw. This relationship is qualitatively similar to the one derived by Padhye et al. [10].

H. *The effect of delays*

So far we have been ignoring the effect of cross traffic (except in the form of exogenous drops). In this section, we consider the effect of cross traffic at the non-bottleneck links. In the next section we look at interactions among transfers that share a bottleneck link.

In the model in Figure 2, queue delays at non-bottleneck links appear as random variables: q_1 is the total queue delay incurred by ACKs on the path from receiver to sender; q_2 is the delay incurred by data packets between the sender and the bottleneck, and q_3 is the delay between the bottleneck and the receiver. At each point in the path, FIFO order is maintained. If a packet is delayed, subsequent packets are delayed enough to maintain order. We assume that delays are uncorrelated (except at the bottleneck, which is modeled explicitly), and that the queue delay of ACKs on the return path are unrelated to the forward-path queue at the bottleneck.

Figure 14 shows the effect of q_3, a delay between the bottleneck and the receiver. In this example, the 25th packet is delayed by 300 ms. A delay at this point in the path halts the data stream immediately; after a lag of $rtt/2$, it halts the growth of the congestion window, so the sender stops sending. After a longer lag, the queue at the bottleneck starts to decline.

After the delayed packet, subsequent packets arrive in rapid succession (limited by the bandwidth of the slowest link between the site of the delay and the receiver). The receiver issues a flurry of ACKs, which causes the sender to issue a burst of packets, which arrive in succession at the bottleneck. If this burst arrives before the queue is depleted, then self-clocking can resume without any loss of throughput.

A delay in the ACK stream has a similar effect, except that the arrival of packets at the receiver is not disrupted. A delay between the sender and the bottleneck only disrupts the growth of the queue.

I. *Sharing with short transfers*

Transfers with large ss and lb can keep a lot of packets in queue, which makes them more robust in the presence of delays and exogenous drops, and increases their share of the bottleneck

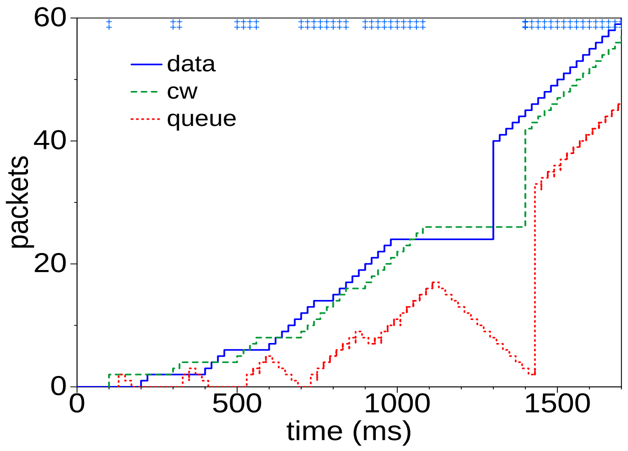


Fig. 14. The effect of a queue delay between the bottleneck and the receiver (q_3).

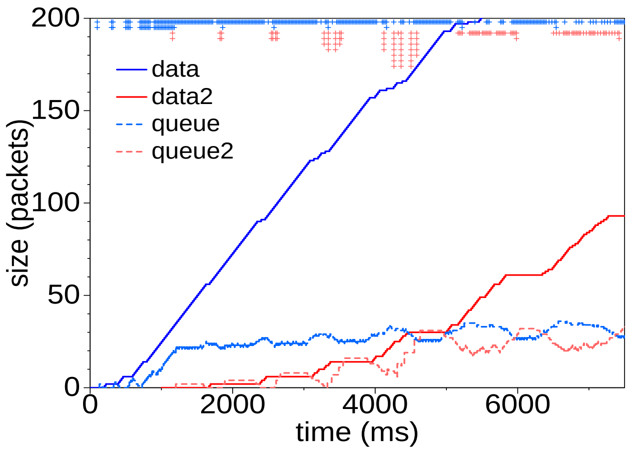


Fig. 15. Transition from slow start to queue sharing. The second transfer sees long and increasing queue delays.

bandwidth. This is good for long transfers, as long as enough buffer space is available, but it is bad for short transfers.

Persistent queues increase the effective latency of the network, which increases the time for a transfer to get through slow start. They also decrease the available buffer space, which makes it less likely that a competing transfer will make the transition to SC without inducing a drop.

Figure 15 shows an example with two transfers, one with $rtt = 200$ ms, the other with $rtt = 300$ ms, both with $ss_i = 30$. The second transfer begins at $t = 1000$ ms, after the first transfer reaches SC and establishes a persistent queue.

For the second transfer, the initial latency is 668 ms, which implies that there were 18 packets in queue when the initial flight arrived at the bottleneck. As cw_2 expands, the queue increases quickly, so the effective latencies of the next four rounds are 720, 740, 840 and 1040 ms.

The transfers reach steady state when $cw_2 = ss_2 = 30$. By then, cw_1 has grown to 37, so the first period in steady state is $20(30 + 37) = 1340$ ms. After that, the period grows by 40 ms per period. Initially, the proportion of bw for the second transfer is $30/(30 + 37) = 45\%$.

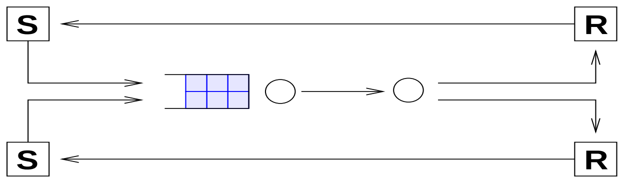


Fig. 16. Two transfers sharing a bottleneck.

This simulation also shows the effect short transfers have on longer transfers. During slow start, a new transfer imposes periodic delays on the existing transfer; in other words, the effect of short transfers is the same as the effect of q_3, a random delay between the bottleneck and the receiver (see Section II-H).

III. Bandwidth Sharing

In this section we analyze the interactions of long transfers sharing a bottleneck. Many TCP models assume that the mechanism of bandwidth sharing is the AIMD congestion avoidance algorithm, and that transfers interact with each other by inducing dropped packets. But when the bottleneck buffer is large enough, transfers share bandwidth by sharing buffer space. Without inducing dropped packets, transfers can reach an equilibrium in which the sum of their send rates is bw, the queue at the bottleneck is constant, and the sum of their congestion windows exceeds bdp indefinitely.

To investigate this behavior, we extend the model shown in Figure 2 to include two (or more) transfers sharing a bottleneck, as shown in Figure 16. We use the notation cw_i, ss_i and lb_i to refer to the congestion window, slow start threshold, and limiting buffer of the ith transfer, and rtt_i and bdp_i to refer to the round trip time and bandwidth-delay product of the ith path.

Figure 17a shows a simulation of two transfers that start at the same time, with $ss_1 = ss_2 = 80$ packets. The lines labeled "data" and "data2" show the total data received by each transfer, in units of packets. The lines labeled "queue" and "queue2" show the number of packets each transfer has in the bottleneck buffer. The marks at the top of the figure show when packets are sent; there are two rows, one for each sender. For clarity, the congestion windows are not shown.

During Phase 1, the transfers tend to repel each other, sending packets in alternating bursts rather than interleaving; this is the packet clustering described by Zhang et al. [2]. Transfers can be divided into a series of intervals where each transfer sends cw_i packets per interval. Thus, the duration of each interval is $tt(cw_1 + cw_2)$.

When $cw_i > ss_i$, the senders switch to congestion avoidance. During each subsequent interval, each congestion window grows by 1 packet. By the same analysis as in Section II-B, we can show that the congestion windows and queue are bounded by the square root of time, as in Phase 3 (see Section II-B). When $cw_i > lb_i$ for all transfers, the queue stops growing, but the cw_i continue to grow as in Phase 4.

These observations lead us to define a transfer state called QS, for "queue sharing." **The definitive characteristic of QS is that two or more transfers share a bottleneck and the sum of their congestion windows exceeds *bdp*.**

A. Unequal sharing

Previously we have used rtt to denote the base round trip time for a path, excluding queue delays. In this section we introduce $ertt$ to denote the effective rtt including queue delays, and use $brtt$ for base rtt.

When transfers share a bottleneck through AIMD congestion avoidance, they share unfairly. Specifically, each transfer gets a share of the bandwidth in proportion to $1/ertt$. Floyd and Jacobson show that the cause of this bias is the additive increase algorithm; after a dropped packet, the long-rtt transfer is slower to claim its share of the available bandwidth [23].

But this heuristic only applies when throughput is congestion-limited. In QS, throughput is bandwidth-limited and the proportion of sharing is determined by ss and lb, and depends only weakly on $ertt$ and p.

To demonstrate this effect, Figure 17b shows a transfer with $brtt = 100$ ms sharing a bottleneck with a transfer with $brtt = 200$ ms. The values of ss_i are chosen so that in steady state the two transfers have the same number of packets in queue. At $t = 4000$, the queue delay at the bottleneck is 400 ms, so the $ertt_i$ are 500 ms and 600 ms, respectively. By the $1/ertt$ heuristic, the long-rtt transfer should get 45% of the bottleneck bandwidth. By the queue sharing heuristic, we expect the transfers to share equally, and they do.

On the other hand, if one transfer has a higher ss, it tends to keep more packets in queue and get a higher proportion of bw. Figure 17c shows two transfers with the same $brtt$; by the $1/ertt$ heuristic, they should share equally. But in this case the values of ss_i are chosen so that the first transfer keeps more packets in queue. By the queue sharing heuristic, the transfer with larger ss should get 57% of the bottleneck bandwidth; in the simulation it gets 58%.

This analysis also applies when more than two transfers share a bottleneck, and in the presence of random delays. Figure 17d shows three transfers with the same rtt and $ss_i = 30$, 15, and 5 packets. This example includes random delays chosen from a lognormal distribution with parameters $\zeta = 4$ and $\sigma = 0.6$. The delays cause the transfers to interleave rather than alternate, but the effective throughput of the transfers is as expected. Our analysis predicts that the transfer with $ss_i = 30$ should get $30/(30 + 15 + 5) = 60\%$ of bw; in the simulation it gets 56%.

In Phase 4, the number of packets in flight is limited by lb_i, so queue residency is between $lb_i - bdp_i$ and lb_i, depending on packet spacing. Again, we can use the expected queue residency to predict the proportion of queueing.

Another characteristic of QS is that transfers are fast to discover available capacity. Under AIMD congestion avoidance, transfers increase their send rate linearly, so when a transfer through a bottleneck ends, it can take many rtts for the remaining transfers to claim the available capacity. In QS, the remaining transfers use the available capacity immediately, as shown in Figures 17b and 17c.

B. The effect of delayed ACKs

In this section we use ns to validate our model by comparison with a more detailed simulation and to investigate the effect of delayed ACKs. We extend the configuration in Figure 6

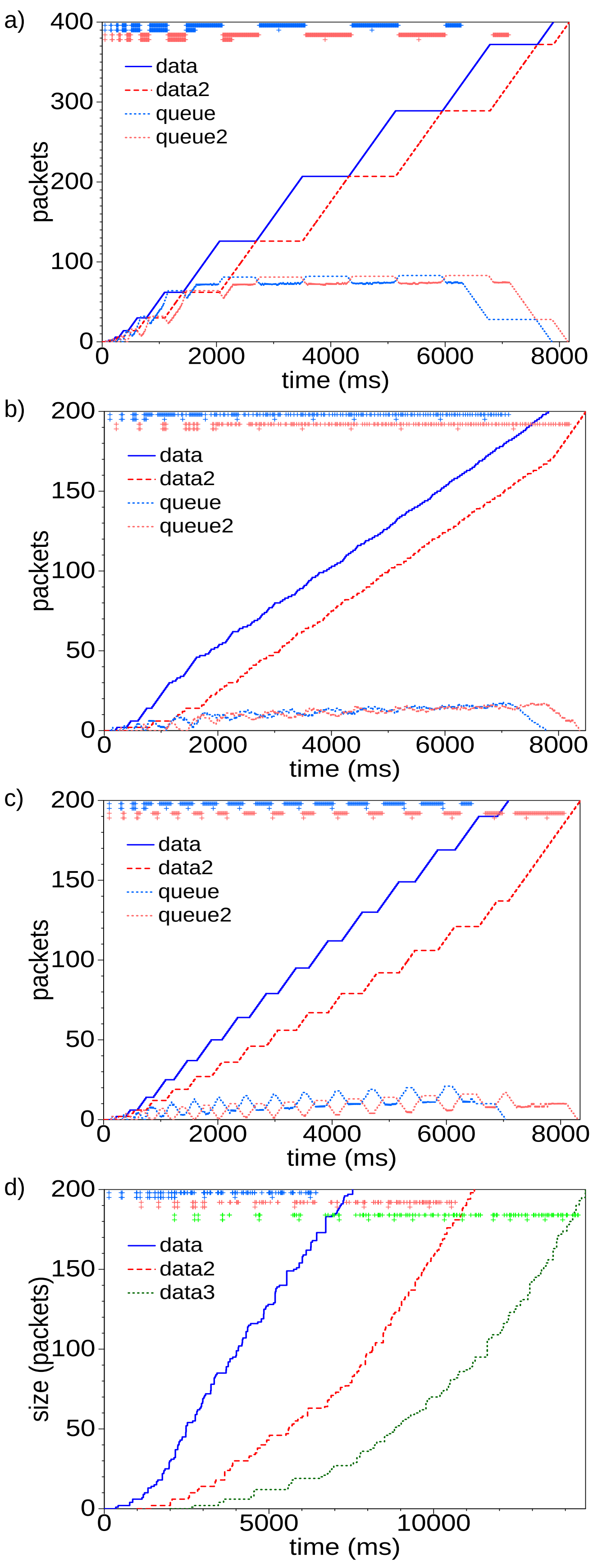


Fig. 17. Self-clocking with multiple transfers: (a) queue sharing, (b) transfers with different rtt, (c) transfers with different ss, (d) transfers with different ss.

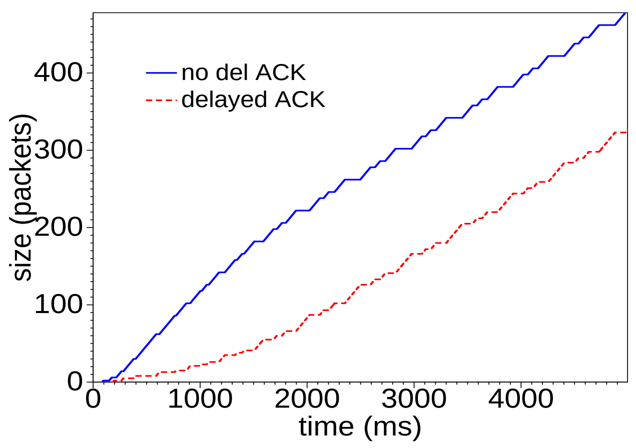


Fig. 18. A transfer that implements delayed ACKs sharing a bottleneck with one that doesn't (ns simulation).

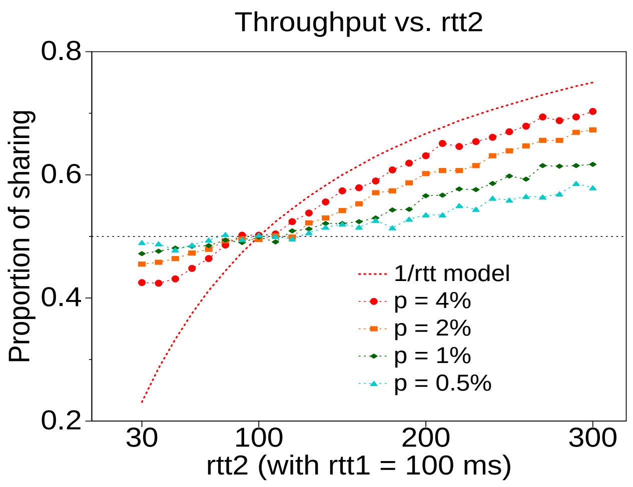


Fig. 19. With exogenous drops, throughput depends on rtt, but only indirectly.

to include n TCP senders sharing the bottleneck link, with rtts varying from 40 – 120 ms.

Simulating this model with and without delayed ACKs, we see no effect on the qualitative results from the previous section:

- Relative $ertt$ for different senders has only a weak effect on the proportion of sharing.
- In Phase 3, a sender with a larger ss gets a larger share of bw.
- In Phase 4, a sender-receiver pair with a larger lb gets a larger share of bw.

`ns` also allows us to simulate a sender that implements delayed ACKs sharing a bottleneck with one that doesn't. Figure 18 shows two transfers with $brtt = 60$ ms, $bdp = 10$ packets, and $ss = lb = 30$ packets. One of the receivers implements delayed ACKs; the other doesn't. As expected, the transfer with delayed ACKs takes longer to get through Phase 1, but in QS the transfers share bw equally, at least approximately. Interestingly, the periodic structure is more complex; instead of strict alternation, there is more interleaving of packets.

We conclude that delayed ACKs have no substantial effect on the qualitative behavior of QS, but in some cases they modify the pattern of interleaved transmission.

C. The effect of random drops

In this section we examine the effect of exogenous drops on bandwidth sharing. Figure 19 shows the result of a simulation with $brtt_1 = 100$ ms and values of $brtt_2$ from 30–300 ms. The figure plots the proportion of sharing, $th_1/(th_1 + th_2)$, versus rtt_2 for several values of p, and compares the result to the proportion of sharing predicted by the $1/ertt$ heuristic. We observe:

- For high drop rates, a transfer with a short $ertt$ gets a greater share of bw, in accordance with the $1/ertt$ heuristic.
- In a range of moderate drop rates (0.5 – 4% in the example), TCP shares more fairly than the $1/ertt$ model predicts, especially in the left side of the figure where bdp_2 is small.
- At low drop rates (less than 0.5% in the example), bandwidth sharing is nearly independent of $ertt$.

In summary, when the drop rate and bdp are low enough to allow QS, the proportion of sharing depends primarily on queue residency, and only weakly on $ertt$. In turn, queue residency depends strongly on ss and lb. In Section IV-B we show measurements that demonstrate these effects in the Internet.

IV. Measurements

Our analysis and simulations are based on the assumption that the bottleneck buffer is large enough, and the drop rate low enough, to allow SC. How often is this true in the current Internet?

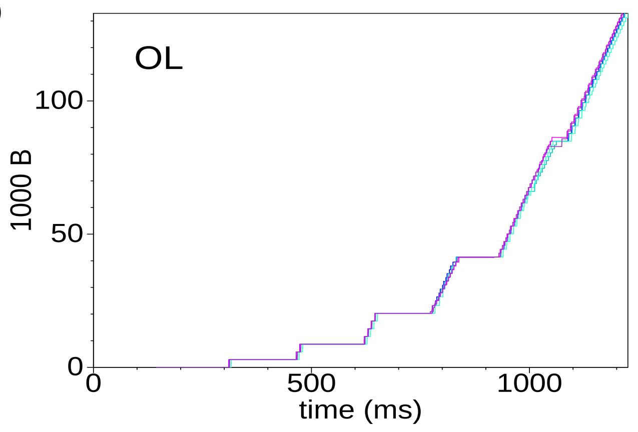
To answer that question, we need a sample of Internet paths. Our first sample includes paths from a single client (Client 1) to a set of geographically-distributed web servers. To generate this set, we used traces from the IRCache Project (`http://www.ircache.net/`) to find large files available from servers busy enough not to notice our measurements. Looking at one day of traces from 10 proxy servers, we identified 83 frequently-accessed files that were at least 100,000 bytes. We downloaded each file 10 times with an average of 100 seconds (exponentially distributed) between them. The receive buffer was set to 1 MB, so we expect the limiting buffer to be at the sender.

Next we classified each path according to which state most transfers were in. We did this by a combination of statistical techniques and visual examination. These techniques are admittedly ad hoc, but our goal is not to create an automated process for classifying TCP connections (see Section IV-C), but to demonstrate the existence and estimate the prevalence of the phenomena predicted by our model.

We use statistical heuristics similar to those in T-RAT [24] to identify flights of packets and estimate rtt. We have several measurements for each path, and each measurement yields

Client	Median rtt, ms	Median bw, Mbps	Median bdp, KB	90%ile bdp, KB
1	86.3	2.77	21.6	57.0
2	82.9	7.95	56.3	154
3	70.3	23.2	141	572

TABLE I
ESTIMATED PATH PARAMETERS.

a)


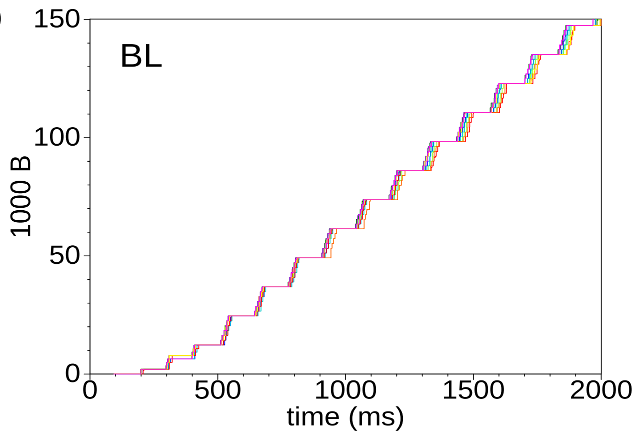
b)


Fig. 20. Timing charts for paths with different steady-state behavior.

several estimates of rtt, so we use the minimum of all measurements to estimate the base rtt of the path.

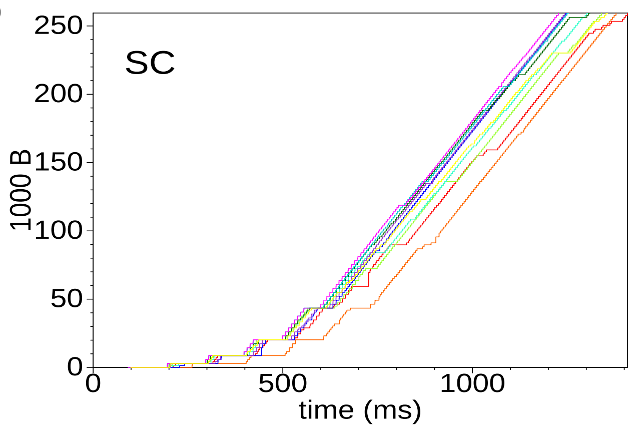
To estimate bottleneck bandwidth we use basic packet pair techniques [25]. In general these techniques are not highly reliable [26], but because we observe several long transfers on each path, we are able to filter aggressively, which improves the consistency of the results, and may improve the accuracy [27]. In any case, we only need a coarse estimate for our purposes.

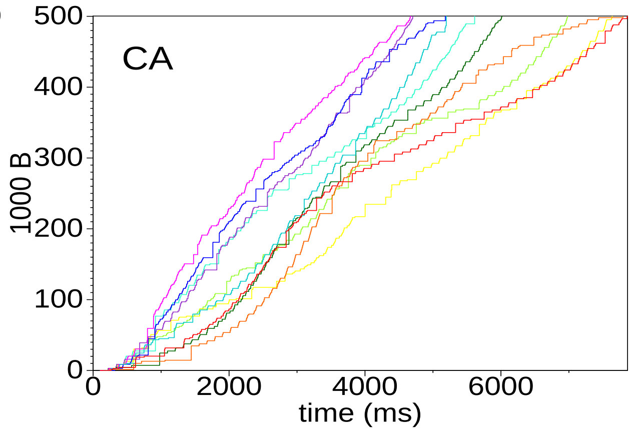
The first line of Table I summarizes these estimates for the 83 paths we observed from Client 1. The estimated bdp is the product of the estimated rtt and bw. This client is connected to the Internet by 2 T1 lines with a total effective bandwidth of roughly 2.8 Mbps. In many cases the T1s are the bottleneck, so this sample of paths is skewed toward relatively low bw and bdp.

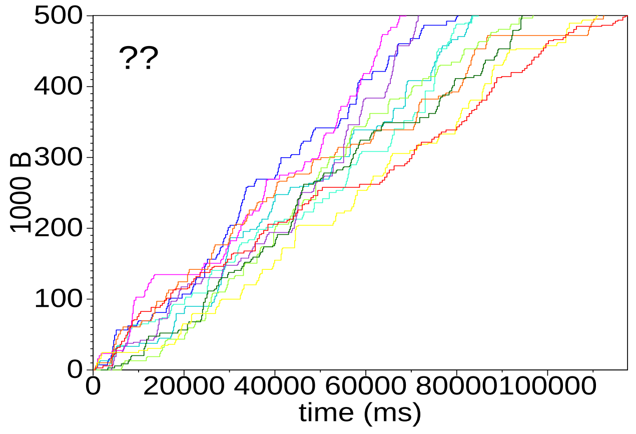
With these estimates in hand, we examine the timing charts and classify each transfer using the following criteria:

1. If packets arrive in identifiable flights and the number of packets in successive flights roughly doubles, we conclude that a transfer is in Phase 1. If the transfer ends in Phase 1, we classify it as opportunity-limited (OL), which means that cw never had the opportunity to exceed bdp. Figure 20a shows an example in which all ten transfers are OL.

2. If a transfer leaves Phase 1, and the rest of the packets arrive in identifiable flights, and the size of the flights is roughly constant, we classify the transfer as buffer-limited (BL), which means that the number of packets in flight is limited by the send or receive buffer. Figure 20b shows an example in which all ten transfers are BL.

a)


b)


c)


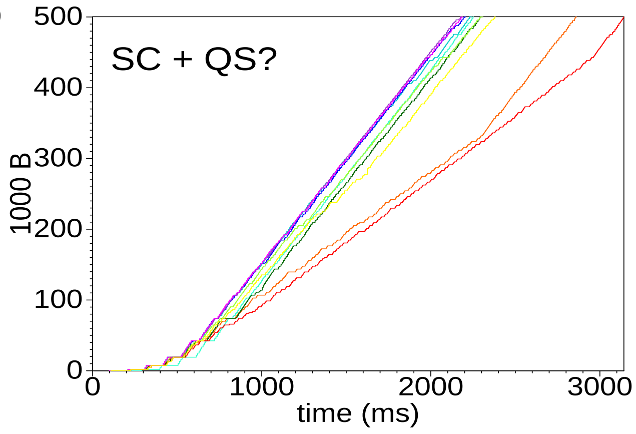
d)


Fig. 21. Timing charts for paths with different steady-state behavior.

3. If a transfer leaves Phase 1 and the rest of the packets arrive at roughly equal intervals without identifiable flights, and if the modal interarrival time is within a factor of two of a packet transfer time at the estimated bw, we classify the transfer as SC (self-clocking). Figure 21a shows a path where all transfers reach SC after 4 rounds of slow start. We confirm that these transfers are not OL because the last "flight" is more than double its predecessor, and much larger than the estimated bdp.

4. The received data curve in these figures is the integral of the receive rate, so the AI phase of congestion avoidance appears as a parabola. When a drop occurs, there is usually a visible delay and, when the transfer resumes, the slope is roughly halved. If the data curve shows linear acceleration during additive increase and at least one point of inflection at a multiplicative decrease we classify it as CA (congestion avoidance). Figure 21b shows a path where several transfers demonstrate the characteristic behavior of AIMD congestion avoidance.

Our classifications are deliberately conservative in the sense that we only classify a transfer if it clearly demonstrates characteristic behavior of OL, BL, SC or CA. Transfers that do not fit neatly into these categories are unclassified (labeled `??` in Table II). Unclassified transfers are most likely to be CA or QS, and unlikely to be SC.

Several measurements in this dataset show characteristics of queue sharing. Figure 21c shows an example where arrivals can often be divided into rounds, but there is no consistent pattern in the number of packets per round. Several of these transfers resemble Figure 17d, so they may be examples of QS, but we don't have enough information in these measurements to confirm that. In Figure 21d, most transfers are in SC, but two transfers appear to share the bottleneck with another, unobserved, transfer. In both cases, the transfers revert to SC when the competing transfer completes, but there is no clear-cut periodic behavior, so again we hesitate to call this QS. In the next section, we describe a set of measurements specifically designed to identify queue sharing.

Table II summarizes our classifications. On 61 of the 83 paths (73%), the majority of the 10 transfers are in SC. There are only 7 paths where CA is the most common state. Ten paths are BL, two are OL, and two are unclassified. One path is application-limited (AL); according to the HTML header, the server is running `thttpd`, which limits transfer rates.

We repeated the measurements from Client 2, which is connected to the Internet by cable modem. The throughput for many transfers is higher (near 8 Mbps), so this sample includes paths with higher bdp. We expect the prevalence of SC to be lower, and it is, but not by much. On 54 of the 88 paths (61%) SC is the most common state.

Client	SC	CA	BL	OL	??	AL	Total
1	61	7	10	2	2	1	83
2	54	6	11	9	7	1	88
3	24	24	44	17	24	0	133

TABLE II
PATH CLASSIFICATIONS.

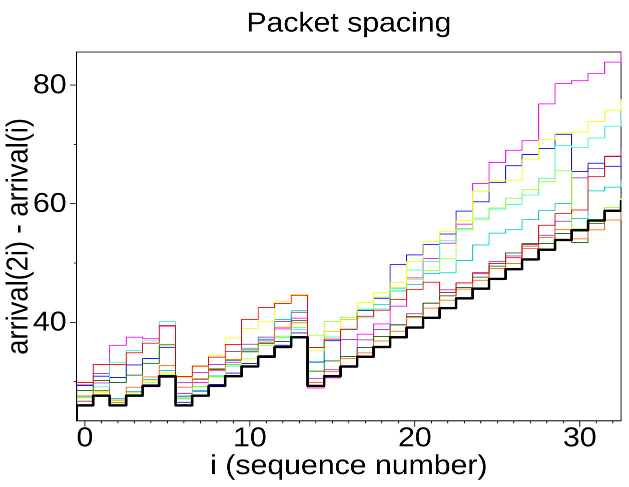


Fig. 22. Packet interarrival times (d_i) for 10 transfers during the transition to SC.

Later we made a set of measurements from Client 3, which is on the campus of a West Coast university with multiple connections to the Internet, including a T1, a DS-3 (45 Mbps) and two Gigabit Ethernets (1 Gbps each). Using new data from IR-Cache, we identified 133 large files available from busy servers. Since we expected many high-bdp paths, we increased the size limit to 2000 KB. In this dataset, we find a lower prevalence of SC and more CA. Still, on 24 out of 133 paths (18%), SC is the most common state.

We conclude that the self-clocking behaviors demonstrated in our model exist in the Internet, and that SC is a common state for long TCP transfers, at least for some clients.

A. *Validation of SC*

Although we don't measure the queue or cw during our measurements, we can confirm that they demonstrate self-clocking by examining the time between packet arrivals more carefully.

If the sender is in slow start, and the receiver ACKs every packet, then the ACK of packet i will cause the sender to transmit packets $2i$ and $2i + 1$. If a_i is the arrival time of packet i, then

$$d_i = a_{2i} - a_i = rtt + q \cdot tt \tag{9}$$

where q is the number of packets in queue when packet i arrives at the bottleneck, and $q \cdot tt$ is the total transmission time of those packets.

In Figure 22, the thin lines show d_i for 10 transfers from the same server. The black line shows the packet spacing we expect if the queue at the bottleneck grows as in our simulation of self-clocking. In some cases the actual interarrival time is longer than predicted, probably due to cross traffic, but it is almost never shorter. This result shows that for transfers that appear to be in SC, the spacing between arrivals is consistent with the behavior of the queue we expect during SC.

B. *Bandwidth sharing*

To investigate the prevalence of queue sharing, we initiated simultaneous transfers from pairs of servers to the same client. We chose Client 1 because we have determined that self-clocking is common for transfers to this client.

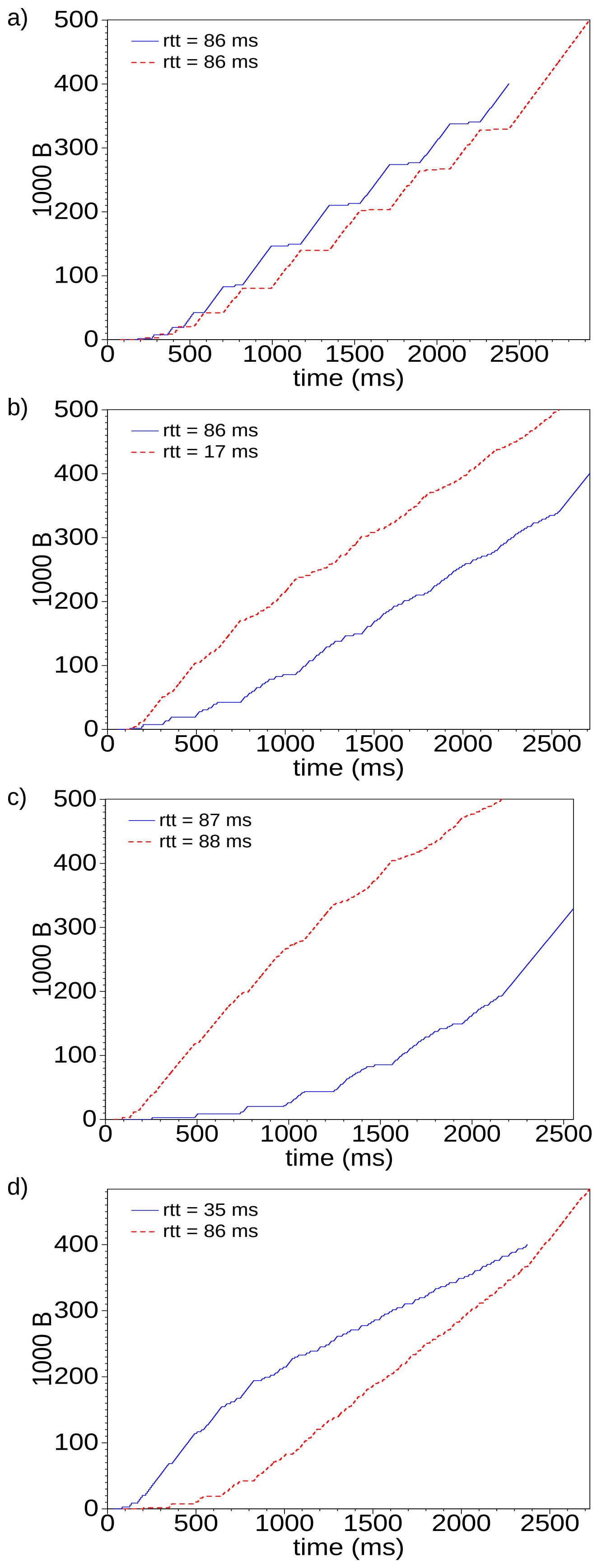


Fig. 23. Measurements showing queue sharing: (a) transfers from different sites with the same rtt, (b) transfers with different rtt, (c) transfers with the same rtt where one transfer is delayed (d) transfers with different send buffers.

We identified 23 web servers with files larger than 300 KB, and generated 100 test cases by choosing two servers at random and initiating two downloads in rapid succession. As in the previous experiment, we use statistical heuristics to identify breaks between flights and estimate rtt, and packet pair techniques to estimate bw and bdp for each path. We then apply the following classification criteria:

1. If the transfers don't overlap because one is delayed, we know that they did not share the bottleneck. This happens in 3 cases.
2. When one transfer ends, if the throughput of the other transfer is unaffected, we conclude that the transfers were not limited by the same bottleneck link. This happens in 6 cases.
3. If packets arrive in alternating flights, the size of the flights increases linearly or stays constant, and when one of the transfers ends the other is able to use the available bandwidth immediately, we classify the measurement as QS.
4. If either transfer shows the characteristic curvature and inflection points of AIMD congestion avoidance, we classify the measurement as CA.

Based on this classification, there are 91 cases where we succeeded in inducing simultaneous transfers that share a bottleneck. In 68 of these cases (75%) the timing charts show behaviors characteristic of queue sharing. Figure 23 shows several examples.

In another 11 cases the periodic structure is not clear, but there appears to be queue sharing at the bottleneck, because when the first transfer ends, the throughput of the second increases immediately. In the remaining 12 cases, one of the transfers is in CA, so that when the first transfer ends, the second accelerates gradually. These results suggest that in network paths where SC is common, QS is a prevalent mode of bandwidth sharing for long transfers.

Looking at specific cases, we see support for several of the behaviors predicted by our model. Figure 23a shows one of the clearest examples of queue sharing between two servers with roughly the same rtt. This measurement is remarkably similar to the simulation output in Figure 17a.

Figure 23b shows two servers with rtts that differ by a factor of 5 (17 ms and 86 ms). The transfer on the long-rtt path takes longer to get through slow start, but in steady state the transfers share bandwidth almost equally, just as in the simulation in Figure 17b.

Conversely, Figure 23c shows two servers with the same rtt. One of the transfers gets a head start and occupies the queue, so the second transfer takes longer to get through Phase 1. When the first transfer ends, the second transfer is able to claim the available bandwidth immediately, as our model predicts.

In many of these measurements, the arrival pattern shows the characteristic scalloped shape that our model predicts when paths with different rtt share a bottleneck, or when one starts later than another.

Figure 23d shows a case where the second transfer to start gets higher throughput in steady state, despite a longer rtt. The most likely explanation is that the queue residency for the first transfer is limited by the send buffer. In our previous measurements, we observed buffer-limited transfers from the same server, which confirms that this server provides relatively small send buffers.

These examples show that under some conditions (low bdp and low p) bandwidth sharing is determined by ss and lb, and depends only weakly on rtt.

C. Related work

Zhang et al. use packet-level information and a tool called T-RAT to identify the limiting factor for a large dataset of TCP connections [24]. They find that a majority of transfers are opportunity-limited; in our measurements, we see a lower percentage because we deliberately induce long transfers. Of the remaining connections, they find that the majority are application-limited; again, we see a lower percentage because our measurements are based on large file transfers, which are less likely to be application-limited.

They characterize many of the remaining connections as congestion-limited or buffer-limited, but this classification is based on statistical heuristics that break transfers into apparent flights. For transfers in SC there is often no apparent break between flights, and apparent breaks might be due to cross-traffic; for transfers in QS, apparent breaks are due to interactions between transfers, and unrelated to rtt. Therefore it is not clear how T-RAT would classify a connection in SC or QS. According to our interpretation of their heuristics, some connections in SC might be misclassified as congestion-limited, and some connections in QS might be misclassified as buffer-limited.

Jaiswal et al. use passive measurements to characterize TCP connections [28]. Their heuristics are also based on the assumption that the packets of a TCP connection can be divided into flights, and that there is no overlap between the arrival of acknowledgments and the transmission of additional data. But this overlap is very common in SC. According to our interpretation of their heuristics, some connections in SC and QS might be misclassified as application-limited.

V. Conclusions

We have used a simple network model to investigate the behavior of TCP when the buffer at the bottleneck link is larger than the bandwidth-delay product (bdp) of the path. We identify three phases that can occur in the transition from slow start to congestion avoidance. Our analysis explains some previous observations, including sub-linear growth of the congestion window, and predicts several new behaviors. We present measurements that show that these behaviors occur in the Internet. Our analysis and observations have several implications:

- Under some conditions (large buffers, low drop rates), TCP connections can enter a steady state, which we call SC, in which the congestion window can exceed bdp indefinitely. Transfers can transition from slow start to SC without inducing a drop.
- We derive the size of the bottleneck buffer, bb, needed to support SC. In the worst case, SC may require $bb > lb$, where lb is the smaller of the send and receive buffers. But in the best case, when $lb = bdp$, SC is possible with minimal bb.
- We derive the range of drop rates in which self-clocking has a significant effect on performance. We can summarize these results by defining a parameter $\beta = p \cdot bdp^2$ that characterizes a network path with drop rate p and bandwidth-delay product bdp (measured in packets). If $\beta < 2/3$, transfers can stay in SC indefinitely. If $2/3 < \beta < 8/3$, transfers will sometimes be in SC. When $\beta > 8/3$, transfers stay in congestion avoidance (CA). These results are based on periodic drops; with random correlated drops, the range where connections vacillate between SC and CA is wider.
- We present an analysis of TCP performance in the domain where transfers are sometimes limited by the congestion window and sometimes by the bottleneck bandwidth, and verify this analysis by comparison with `ns` simulations.
- When multiple transfers share a bottleneck, they can enter a steady state, which we call QS for “queue sharing”. In QS, the sum of the congestion windows can exceed bdp indefinitely without inducing drops.
- Transfers in SC and QS may not respond correctly to a dropped packet; after a pause, they resume sending at a slightly *higher* rate.
- When transfers in QS share bandwidth, the proportion of sharing depends strongly on `ssthresh` and the send/receive buffers, and only weakly on rtt. Under these conditions, TCP does not obey the $1/\sqrt{p}$ and $1/rtt$ heuristics, which are the basis of TCP-friendly algorithms [29][30]. Thus, TCP is not always TCP-friendly.

Some of the behaviors we observed have been described before, but they are often omitted from models of TCP. For example, many models of TCP performance, and heuristics for characterizing TCP behavior, are based on the assumption that most connections send packets in discrete flights, where the time between flights is roughly rtt. For transfers in SC, there are often no breaks between flights, and for transfers in QS, the time between flights depends on the congestion windows and transfer time at the bottleneck, not rtt.

On the other hand, our observations suggest ways to improve TCP. In SC, the congestion window can be much greater than bdp, making it ineffective as an estimate of available capacity. Some implementations address this problem by bounding cw with a system-level parameter (like `cwnd_max`), but if this parameter is too low, it limits performance, and if it is too high it is irrelevant. It may be desirable set this bound dynamically, either by using packet pair techniques to estimate the bottleneck bandwidth or by using sub-linear growth in the congestion window as a signal that $cw > bdp$.

In some conditions the initial value of `ssthresh` and the size of the send and receive buffers have a strong effect on TCP queue behavior in steady state. It may be desirable to adjust these values dynamically. Several projects have tried to measure bdp and set ss accordingly [20][31][32] [21][33].

In future work, we would like to investigate the implications of this model for network provisioning and queue management. A perpetual problem with TCP is that large buffers allow persistent queues, which increase the apparent latency of the network, but smaller buffers reduce the ability of the network to handle bursts, which increases the drop rate. Our analysis complicates this picture, showing that the queue behavior of transfers is different in SC, QS and congestion avoidance. Thus, optimal buffer sizes may depend on the prevalence of SC and QS. Recently Appenzeller et al. revisited buffer sizing for TCP flows [18]. They assume that long flows are in AIMD congestion avoidance, so it is not clear whether their results apply when SC and QS are prevalent.

References

[1] S. Floyd, “Connections with multiple congested gateways in packet-switched networks, Part 1: One-way traffic,” *ACM Computer Communication Review*, vol. 21, no. 5, pp. 30–47, Oct. 1991.

[2] L. Zhang, S. Shenker, and D. D. Clark, “Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic,” in *SIGCOMM*, 1991, pp. 133–147.

[3] T. Lakshman and U. Madhow, “The performance of TCP/IP for networks with high bandwidth-delay products and random loss,” *IEEE/ACM Transactions on Networking*, vol. 5, no. 3, pp. 336–350, July 1997.

[4] T. V. 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 *INFOCOM (3)*, 1997, pp. 1199–1209.

[5] T. Ott, J. Kemperman, and M. Mathis, “The stationary behavior of ideal TCP congestion avoidance,” August 1996, unpublished manuscript.

[6] M. Mathis, J. Semke, and J. Mahdavi, “The macroscopic behavior of the TCP congestion avoidance algorithm,” *Computer Communications Review*, vol. 27, no. 3, 1997.

[7] V. Misra, W. Gong, and D. Towsley, “Stochastic differential equation modeling and analysis of TCP windowsize behavior,” University of Massachusetts, Tech. Rep. ECE-TR-CCS-99-10-01, 1999.

[8] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe, “Modeling TCP throughput: A simple model and its empirical validation,” in *SIGCOMM*, 1998, pp. 303–314.

[9] J. Padhye, V. Firoiu, and D. Towsley, “A stochastic model of TCP Reno congestion avoidance and control,” University of Massachusetts, Tech. Rep. CMPSCI 99-02, 1999.

[10] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe, “Modeling TCP Reno performance: A simple model and its empirical validation,” *IEEE/ACM Transations on Networking*, vol. 8, no. 2, pp. 133–145, April 2000.

[11] E. Altman, K. Avrachenkov, and C. Barakat, “A stochastic model of TCP/IP with stationary random losses,” in *SIGCOMM*, 2000.

[12] ——, “TCP in the presence of bursty losses,” *Performance Evaluation*, vol. 42, pp. 129–147, 2001.

[13] E. Altman, K. Avrachenkov, C. Barakat, and R. N. nez Queija, “TCP modeling in the presence of nonlinear window growth,” INRIA, Tech. Rep. RR-4312, Novemeber 2001.

[14] A. Misra and T. J. Ott, “The window distribution of idealized TCP congestion avoidance with variable packet loss,” in *INFOCOM (3)*, 1999, pp. 1564–1572.

[15] F. Baccelli and D. Hong, “The AIMD model for TCP sessions sharing a common router,” in *Proceedings of the Conference on Communication, Control and Computing*, October 2001.

[16] V. Jacobson, “Congestion avoidance and control,” in *SIGCOMM*, 1988, pp. 314–329.

[17] “The ns-2 network simulator,” http://www.isi.edu/nsnam/ns, 2005.

[18] G. Appenzeller, I. Keslassy, and N. McKeown, “Sizing router buffers,” in *SIGCOMM*, 2004, pp. 281–292.

[19] W. Noureddine and F. Tobagi, “The transmission control protocol,” Stanford University, Tech. Rep., July 2002.

[20] L. S. Brakmo and L. L. Peterson, “TCP Vegas: End to end congestion avoidance on a global internet,” *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1465–1480, 1995. [Online]. Available: citeseer.nj.nec.com/brakmo95tcp.html

[21] M. Allman and V. Paxson, “On estimating end-to-end network path properties,” in *SIGCOMM*, 1999, pp. 263–274. [Online]. Available: citeseer.nj.nec.com/allman99estimating.html

[22] C. Barakat and E. Altman, “Performance of short TCP transfers,” in *NETWORKING*, ser. Lecture Notes in Computer Science, vol. 1815. Springer, 2000, pp. 567–579.

[23] S. Floyd and V. Jacobson, “Traffic phase effects in packet-switched gateways,” *Journal of Internetworking: Practice and Experience*, vol. 3, no. 3, pp. 115–156, September 1992.

[24] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, “On the characteristics and origins of Internet flow rates,” in *SIGCOMM*, 2002.

[25] S. Keshav, “A control-theoretic approach to flow control,” in *SIGCOMM*, 1991, pp. 3–15. [Online]. Available: citeseer.nj.nec.com/keshav91controltheoretic.html

[26] C. Dovrolis, P. Ramanathan, and D. Moore, “Packet dispersion techniques and capacity estimation,” *IEEE/ACM Transactions on Networking*, December 2004. [Online]. Available: citeseer.nj.nec.com/513517.html

[27] A. Downey, “An empirical model of TCP performance,” Olin College, Tech. Rep. TR-2003-001, August 2003.

[28] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, “Inferring TCP connection characteristics through passive measurements,” in *IEEE INFOCOM*, 2004.

[29] J. Mahdavi and S. Floyd, “TCP-friendly unicast rate-based flow control,” http://www.psc.edu/networking/papers/tcp_friendly.html, 1997.

[30] J. Padhye, J. Kurose, D. Towsley, and R. Koodli, “A model-based TCP-friendly rate control protocol,” University of Massachusetts, Tech. Rep. CMPSCI 98-04, 1998.

[31] J. C. Hoe, “Improving the start-up behavior of a congestion control scheme for TCP,” in *SIGCOMM*, 1996, pp. 270–280. [Online]. Available: citeseer.nj.nec.com/hoe96improving.html

[32] M. Aron and P. Druschel, “TCP: Improving startup dynamics by adaptive timers and congestion control,” Dept. of Computer Science, Rice University, Tech. Rep., 1998. [Online]. Available: citeseer.nj.nec.com/aron98tcp.html

[33] C. Partridge, D. Rockwell, M. Allman, R. Krishnan, and J. Sterbenz, “A swifter start for TCP,” BBN, Tech. Rep. 8339, March 2002.