An empirical model of TCP transfers

Allen B. Downey

January 14, 2003

Abstract

We propose a model of TCP performance that captures the behavior of a diverse set of network paths. The model uses more parameters than previous efforts, but we show that each feature of the model describes an effect that is important for at least some paths. We show that the model is sufficient to describe the datasets we collected with acceptable accuracy. Finally, we show that the model's parameters can be estimated using simple, application-level measurements.

1 Introduction

The goals of this paper are:

- To build a network measurement tool that measures the characteristics of a network path necessary to predict TCP performance.
- To build a model of TCP behavior that predicts TCP performance as a function of given network path characteristics.

These two problems are complementary. Thus our ultimate goal is to identify a set of parameters that can be estimated with minimal cost and that characterize the TCP performance of a path with maximal accuracy.

Our approach is empirical in the sense that we are looking for a model that works over the range of characteristics that is typical of current networks. Thus, we are willing to ignore characteristics that might affect TCP performance in theory, but which seem to have little effect in the current Internet. Conversely, we are forced to deal with the full diversity of Internet paths and hosts, including some unexpected phenomena; for example, we find that the growth of the congestion window during slow start is nondeterministic, at least for some sender-receiver pairs.

1.1 Related Work

A number of models have been proposed that relate TCP performance to various path characteristics, including round trip time (rtt), drop rate, bottleneck bandwidth, etc. Most of these models focus on the steady state behavior of long transfers [14] [15] [20] [21] [28] [22] [30] [29] [25] [11] .

These models focus on congestion avoidance as the primary steady state behavior of TCP. Recent work considers cases where buffers (rather than congestion) limit the size of the send window [2][13]. Our model is less detailed than these, but it includes a high-level description of three steady-state behaviors: congestion avoidance, buffer-limited, and self-clocking.

One limitation of many previous models is that they treat the drop rate as an exogenous variable; that is, a characteristic of the network that is independent of the behavior of TCP. One exception is [24], which recognizes that the dropped packets observed by a TCP transfer are at least partly endogenous; that is, caused by the transfer itself. As an example, during slow start it is common for the congestion window to exceed the delay-bandwidth product of the nework, causing the sender to saturate the bottleneck link, and possibly causing packets to be dropped even in the absence of cross traffic. The exogenous drop rate doesn't appear in our model explicity. Instead, we estimate the probability of dropping out of slow start as a function of the current window size. This probability is determined partly by the drop rate, but (in most cases) primarily by the window size and capacity of the path.

Some models of short TCP transfers have been proposed [9] [23] [4] [34] [33]. These models identify two sources of variability in transfer times: variability in rtts and dropped packets. Our datasets suggest a third source of variability: nondeterminism in the growth of the congestion window during slow start. In some cases, this nondeterminism is the primary source of variability. We incorporate this behavior in our model.

Although there are models of short and long transfers, most work has ignored the transition from slow start to steady state. For many paths in the current Internet, this transition happens in the size range from 10–100KB, which happens to be the size range of many TCP transfers. In traces of Web downloads at Boston University, this range contained 15–20 % of the transfers (based on our analysis of datasets from [5]). Our datasets are intended to observe this transition, and we try to describe it in our model.

Of course, the price of realism is complexity. Our model has more parameters than its predecessors, and although it provides some insight into TCP performance, it is not amenable to analysis. Also, it is based on details of a specific path, which requires a measurement infrastructure. But we show that this complexity is necessary; that is, each feature of the model describes an effect that is important for at least some paths. We also show that the model is sufficient to describe the behavior of the datasets we collected, with one exception. Finally, we show that the network characteristics the model uses can be estimated from simple application-level measurements.

1.2 Applications

Why is it useful to predict the performance of TCP transfers? One reason is to improve user interfaces. For example, many web browsers display information on the progress of long transfers and make simple estimates of the remaining transfer times. In theory, these estimates are useful to users, but in current practice they are so innaccurate that many users have learned to ignore them.

Another reason is to automate selection among mirror sites. In many cases, data that is available on the Web or by ftp is available from multiple sites. Often users are asked to choose one of these sites based on geographic proximity, without information about the expected performance of the various paths.

Finally, for distributed applications, these predictions are useful for both resource selection and scheduling.

For users, predictions are probably only useful for transfers that take more than a few seconds. But for applications it may also be important to characterize the performance of short transfers. For example, the performance of a distributed application with fine-grain interaction would likely depend on the transfer time (and variability) of small messages.

Because network performance is so variable, it is usually impossible to predict the duration of a single transfer with any accuracy. Usually the best we can do it to provide a range of likely values, or a distribution of values. Thus, all of our predictions are stochastic, in the sense that we predict the distribution of a set of values, and we evaluate the quality of prediction by the agreement of the predicted and actual distributions.

Different applications might use these predictions differently. For a user interface, it might be best to present the median of the distribution, or a range that is likely to contain the true value. For resource selection, it might be appropriate to choose the resource with the minimum expected value, or the minimum maximal value. Finally, for real-time scheduling, the goal might be to minimize the probability of missing a deadline.

2 Measurement

There are three general approaches to network measurement. One is to collect packet-level information somewhere in a network path. Another is to collect kernel-level information at either the sender or the receiver. The third is to collect information at the application level. The packet and kernel-level approaches provide the most detailed information and the most accurate timings. The advantages of application-level measurement are that it is easy to implement, and the resulting tools are portable.

We start at the application level, to see what is possible with a quick-and-dirty approach, and then evaluate the limitations we encounter. We find that, for the most part, the information we need is available from application-level measurements, and that the benefits of dropping down a level are probably not worth the difficulties.

Most Web browsers already contain code to monitor the progress of HTTP transfers. With a few small changes, we instrumented a version of `wget` to record the arrival time of each chunk of data as presented to the application level. The following pseudocode shows the structure of the instrumentation:

```
set the timer
connect (socket)
record elapsed time
write (request)
while (more data) {
    select (socket)
    record elapsed time
    read (buffer)
    record amount of data
}
```

The `connect` system call returns when the connection is established, so the first elapsed time is the

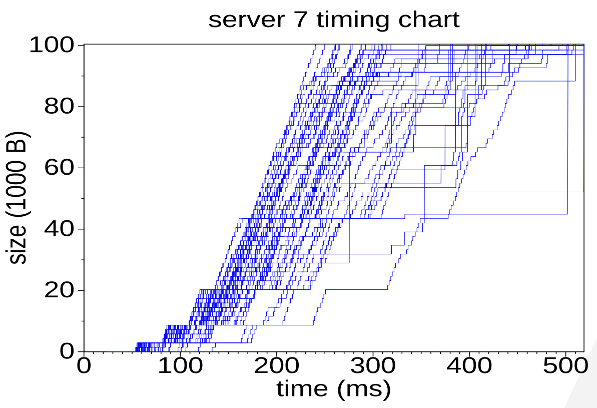


Figure 1: Timing charts for 30 HTTP transfers.

time to send a SYN packet and receive a SYN-ACK packet, which is approximately one rtt.

The `select` system call returns when data from the socket is available. The second elapsed time is the time to send a request and receive the first byte of the reply, which is also approximately one rtt.

For subsequent reads, the elapsed time depends on the arrival of packets and the ability of the OS to present packets to the application layer. In most cases, the spacing of packets is large enough that each `read` corresponds to one packet arrival.

For each transfer, the modified version of `wget` produces two vectors: t_i is the time in ms when the ith read started, and s_i is the total number of bytes read when the ith read completed. Figure 1 plots 30 transfers of a particular file from a Web server, showing s versus t.

At the beginning of each transfer, the slow start mechanism is apparent. Increasing bursts of packets arrive at regular intervals. The first two packets arrive after a little more than 50ms, the next four packets arrive 25ms later, and the next 8 arrive 25ms after that. We can quickly infer that the rtt is 25ms and the initial congestion window (icw) is 2. Looking at the number of packets in each burst, which we call the apparent window size (aws), we can infer that the congestion window at the sender doubles after each round, from 2 to 4 to 8.

In the next round, we expect the congestion window to be 16, and in some cases there is a clear break after the 16th packet, but in many cases there is no apparent break, and packets arrive continuously at regular, small intervals. These cases demonstrate successful TCP self-clocking (see Section 2.1).

A few transfers show evidence of packet loss. When a packet is dropped, the OS can't present additional data to the application layer until the retransmission arrives, so a drop appears as a long horizontal line. When the retransmission arrives, it is made available along with all the data that arrived in the interim; this mass "arrival" appears as a long vertical line. This path shows a significant number of drops, but in most cases they have little effect on performance. At the top of the figure, there are several transfers that suffer long delays because one of the last packets in the transfer was dropped. In this case, there are not enough duplicate ACKs to trigger Fast Retransmission; instead the sender waits for a timeout, with a significant impact on performance.

This figure is promising, because it suggests that simple application-level measurements can provide a lot of information about a network path. Just looking at charts like Figure 1, we can make qualitative assessments of the following characteristics:

- The size and variability of the rtt.
- The initial and subsequent congestion windows.
- The bottleneck bandwidth.
- The buffer size at the sender.
- The frequency and performance impact of dropped packets.
- The characteristics of steady-state behavior (see Section 2.2).

These are exactly the characteristics we need to predict TCP performance. Timing diagrams for all of our datasets are available at `www.caida.org/performance/tcp/timing`, along with our characterization of each.

2.1 TCP Self-clocking

Conventional wisdom holds that the TCP slow start mechanism inevitably ends when the congestion window exceeds the bdp of the path and the sender induces one or more drops at the bottleneck link. As motivation for TCP Vegas, Brakmo and Peterson claim that TCP "*needs* to create losses to find the available bandwidth," and that, "if the threshold window is set too large, the congestion window will grow until the available bandwidth is exceeded, resulting in losses..." [8].

Similar claims are common in discussions of TCP performance. For example, Hoe writes "... the sender usually ends up outputting too many packets too quickly and thus losing multiple packets in the same window" [16]. Allman and Paxson claim "For TCP, this estimate is currently made by exponentially increasing the sending rate until experiencing packet loss" [1]. Barakat and Altman write "Due to the fast window increase, [slow start] overloads the network and causes many losses" [4].

These claims contradict Jacobson and Karels, who suggested that in steady-state TCP is self-clocking [17]. The self-clocking mechanism is based on the observation that if a stream of packets with length L arrive at a bottleneck link with bandwidth B, they will leave the bottleneck link with interpacket spacing L/B, and arrive at the receiver with roughly the same spacing. If the receiver issues ACKs promptly, the ACKs tend to arrive at the sender with the same spacing, which induces the sender to transmit at a rate that approximates B.

When self-clocking works, the transfer induces minimal contention at the bottleneck link, and so is unlikely to cause a packet drop by itself (an endogenous drop). If there are no exogenous drops, the transfer can continue transmitting at rate B indefinitely. During this time, the congestion window continues to grow, but only linearly, as the number of packets per rtt is no longer growing. In this state, the congestion window may be much larger than the bdp.

This observation contradicts an assumption that underlies many models of TCP performance, which is that the send rate is most often determined by the congestion window. In many cases, the congestion window is irrelevant, and congestion control is achieved entirely by the flow control mechanism.

But this leaves us with an empirical question: how often (and in what circumstances) does the TCP self-clocking mechanism succeed? Unfortunately, our observations of self-clocking are limited; in 10 of our 13 datasets, the congestion window never reaches the bdp, either because it is limited by the sender's buffer or because the transfer ends in slow start. But the other three datasets (Servers 7, 9 and 10) show the transition from slow start to self-clocking steady state. For servers 7 and 10, the vast majority of transfers make the transition with no indication of dropped packets. Server 9 is harder to characterize. About half of the transfers enter a self-clocking steady state; the other half seem to enter congestion avoidance.

These observations suggest that in many cases the self-clocking mechanism is effective, and that concerns about congestion induced by slow start may be overstated.

2.2 TCP Steady State

Self-clocking is one of three steady-state behaviors we see in our data sets. Another is the buffer-limited

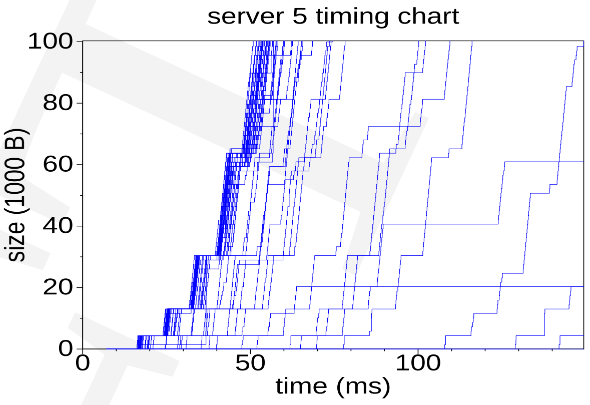


Figure 2: Timing charts for 30 HTTP transfers.

state, in which the performance of the transfer is limited by the buffer size at the sender. The third is congestion avoidance, in which the congestion window grows linearly until a drop occurs and then halves. A transfer may enter congestion avoidance when a dropped packet is detected or when the congestion window exceeds *ssthresh*. Congestion avoidance appears in only a few of our datasets, and only for a small number of transfers.

Figure 2 shows a set of transfers that are primarily buffer-limited. After 2 rounds of slow start, the apparent window size reaches 12 packets and stays there for the next four rounds. Since the window size doesn't increase, we infer that the limitation is the buffer size at the sender, not the congestion window. In fact, the congestion window can grow indefinitely during a buffer-limited transfer.

3 The Datasets

Of course, a model of TCP performance should apply to the widest range of network conditions. To develop our model, we wanted to collect datasets from network paths with a variety of characteristics.

The ubiquity and accessibility of Web servers makes them a convenient tool for network measurements. We started with a few sites that gave us explicit permission to perform experiments. This approach kept us out of trouble, but it is not clear that the paths we looked at are representative of the Internet.

Later, we looked for a more diverse collection of servers. We obtained one day of traces from the IRCache Project (`http://www.ircache.net/`) and looked for frequently-accessed files that were at least 100,000 bytes. Starting with the most popular and working our way down (skipping one that was obviously pornographic), we made measurements of the

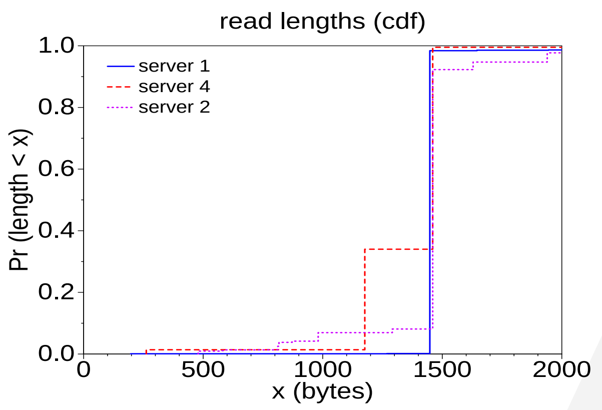


Figure 3: Cdf of read sizes for three servers.

first 11 files we were able to download successfully. We used the HTTP/1.1 Range header to get only the first 100,000 bytes of data (plus roughly 350 bytes of header). Two servers ignored the Range header and sent the whole file.

We chose a file size of 100,000 bytes because we expected that most transfers of that size would leave slow start. In fact, we underestimated; in 6 of the datasets, most transfers never leave slow start.

Each dataset includes 100 transfers, spaced out with an average of 100 seconds between them (exponential distribution). Thus, the duration of the measurements is 3–4 hours. Our intention was to impose a minimally-noticeable load on the servers. This attempt seems to have been successful; if anyone noticed, they didn't contact us. However, since we didn't have permission to make these measurements, we do not identify the servers in this paper, except by numbers that we assigned arbitrarily.

In general, the only information we have about the servers is what we can infer from the URL, but the set appears to include diverse locations, content, and network characteristics, and may be representative of servers in the Internet. Our client is located at Boston University, which is a multi-hosted instititionwith relatively low-traffic, high-bandwidth network connections. So the client end of these paths may not be typical of Internet users.

The next few sections present the steps we used to process these datasets and estimate the characteristics of the network paths we observed.

3.1 Packet size

If the application reads data as soon as it is available, then the receive buffer will usually be empty and the data read will usually correspond to single incoming packet. We expect that most applications can keep up with most network connections, despite the vagaries of the local scheduler.

To test this claim, we plotted the distribution of read sizes for each dataset. Figure 3 shows cdfs of the read sizes for a typical server (Server 1), and two servers that we chose because they are unusual. In the typical case, the vast majority of reads are 1448 or 1460 bytes, which correspond to the maximum segment size (mss) of the path, minus L2 and L3 headers. We conclude that in most cases, the application is reading data fast enough to get a single packet per read.

For Servers 4 and 2, the most common read size is 1460 bytes, but a significant number of packets are smaller. For Server 4, more than 30% of the packets are 1176 bytes. For Server 2, packets span the range from 300 to 2000 bytes. The reason for this diversity is that the initial and subsequent congestion windows on this server are not integer multiples of the mss.

Three of the servers we looked at send a first packet that is smaller than the rest. The sizes of these initial packets were 250–350 bytes, which is roughly the range of sizes for HTTP headers. We conclude that some servers send HTTP headers as a separate segment. Most implementations of TCP buffer outgoing data until a full packet is available or until a timer expires. So if there is a delay after the server writes the HTTP header and before it retrieves the data, the header might be sent as a separate packet. Alternatively, these servers might be flushing the socket after writing the header, although we don't know a reason for doing that.

In general, small packets degrade performance by increasing the overhead of transmitting headers and switching packets. However, in the few cases we saw where servers are sending sub-maximal packets, the performance impact is probably small.

3.2 Bottleneck bandwidth

The idea of using packet spacing to estimate bottleneck bandwidth was proposed by Keshav [18] and has been implemented in various network measurement tools [7] [10] [32] [19] [12].

Brakmo and Peterson implemented a version of TCP Vegas that uses packet-pair bandwidth estimates to choose the value of ssthresh [8]. Similar techniques have been proposed and evaluated elsewhere [16] [3] [1]. Partridge et al. have implemented a version of FreeBSD that uses packet pair estimates to accelerate slow start [31].

To implement packet-pair bandwidth estimation using our measurements, we compute the first differences of the vectors t and s, yielding dt, which is

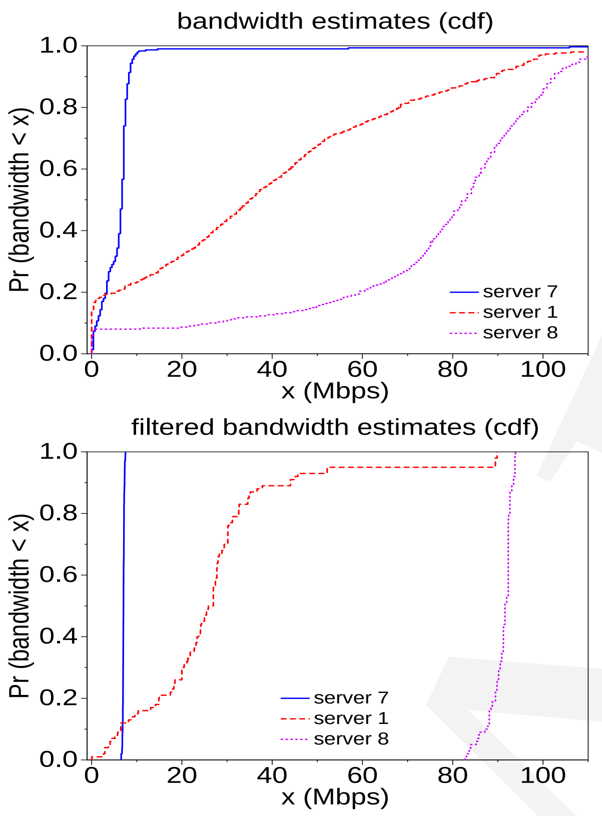


Figure 4: Bandwidth estimates for three servers, with and without filtering.

the interpacket spacing, and ds, which contains the packet sizes. For each packet, we compute the instantaneous bandwidth $bw_i = ds_i/dt_i$. If packets leave the bottleneck link back to back, and their spacing isn't perturbed by cross traffic, bw estimates the bottleneck bandwidth of the path.

Previous packet pair measurements have been based on the assumption that packets often arrive at the destination with unperturbed packet spacing, and that the correct bottleneck bandwidth is the mode of the distribution of estimates. Dovrolis et al. warn that under some traffic conditions, the global mode is determined by cross traffic and not bottleneck bandwidth, but they still expect a local mode at the correct value [12].

For continuous distributions, the notion of a mode is awkward to define, and methods for identifying modes tend to be ad hoc. Furthermore, many of our datasets exhibit no clear modes. Figure 4 (top) shows distributions of bw estimates from servers with slow, medium and high bottleneck bandwidths. Server 7 shows a strong mode around 7 Mbps, but the other two cases are less promising. Server 8 shows a mode near 90 Mbps, but the distribution is nearly uniform from 70 to 100 Mbps. Similarly, the "mode" for Server 1 spans the range from 0 to 100 Mbps. It seems impossible to generate a precise bandwidth estimate from these distributions.

Fortunately, we have more information to work with. Looking at timing charts like Figure 1, we see that interpacket spacing is highly variable, but there are many linear segments where unperturbed packets arrive at what appears to be the bottleneck bandwidth. The processing power of the human eye makes it seem trivial to identify these parallel line segments and estimate their slopes. The trick is to accomplish the same task statistically.

Again, we start by computing, for each chart, the vector of bandwidth estimates bw_i. For each of these vectors, we look at each subsequence of k bandwidth estimates, and compute the deviation of the jth subsequence, $\sigma_j = 1/k \sum_{i=j}^{j+k} |bw_i - m|$, where m is the median of the estimates in the subsequence. The subsequences with the lowest deviation correspond to the straightest line segments in the timing chart.

During processing, we keep only the n subsequences with the lowest deviation, on the assumption that their interpacket spacing represents the true bottleneck bandwidth. We filter out the subsequences with higher deviation, on the assumption that their interpacket spacing has been perturbed. Figure 4 (top) shows distributions of the estimates that remain after filtering with $k = 8$ and $n = 100$. In all three cases, the range of the estimates has been greatly reduced. Furthermore, in all of our datasets, the mode of the distribution is at or near the median; thus, we use the median as our bandwidth estimate and the interquartile distance as an indicator of its precision. This filtering technique works well with a range of values for k and n.

For most of our datasets, we don't know the actual bottleneck bandwidths of the path, so we can't evaluate the accuracy of these estimates. But by dividing our datasets into subsets, we can evaluate their repeatability. For each dataset, we generate 5 subsets with 20 randomly-chosed timing charts in each. We generate a bandwidth estimate for each subset, and compute the range and interquartile distance of the five estimates. Table 3.2 shows the results for all 13 datasets.

In most cases, the range of estimates is small, which suggests that they are actually measuring the capacity of a link in the path. However, one weakness of our technique is that it might be fooled by what Dovrolis et al. call a "post-narrow capacity mode."

For Server 6, the range of estimates is quite wide. On further investigation, we found that this dataset actually contains three paths with different charac-

Server	Est bw	Range	Interquart
1	24.908	(24.908, 28.014)	1.147%
2	63.656	(63.656, 86.860)	0.701%
3	89.040	(88.996, 90.051)	0.513%
4	92.710	(91.975, 92.712)	0.269%
5	90.677	(89.628, 91.937)	0.687%
6	63.870	(41.590, 84.870)	11.974%
7	6.982	(6.914, 7.075)	0.212%
8	91.261	(91.261, 92.313)	0.190%
9	0.331	(0.331, 0.513)	5.845%
10	9.376	(9.356, 9.412)	0.126%
11	89.474	(89.444, 90.866)	0.669%
12	22.694	(21.129, 33.811)	15.300%
13	88.775	(88.775, 90.142)	0.386%

Table 1: Bandwidth estimates for each dataset. Est. bw is based on all 100 timing charts. Range contains the highest and lowest estimates from each subset of 20 charts. Interquart is one-half the interquartile distance, written as a percentage of the estimated bw.

teristics. Subsequent requests for the same URL are actually being handled by different servers, due to changes in DNS information caused by distributed content delivery mechanisms like those used by Akamai Technologies and Speedera Networks.

The filtering technique we use here can be integrated into existing bandwidth measurement tools. As future work, we plan to evaluate this technique more rigorously and compare it with existing tools.

3.3 Congestion windows

The duration of short TCP transfers tends to be a multiple of the round trip time, where the multiplier depends on the behavior of the congestion window at the sender. Thus, in order to predict TCP performance for a given server, we have to measure its initial and subsequent congestion windows. In order to do that, we have to be able to identify the end of each round of packets.

To separate arriving packets into rounds, we look at the vector of interpacket spacing, ds, and identify intervals that seem to be due to congestion control rather than queue delays. To do that, it helps to know the rtt and the interpacket spacing at the bottleneck. As a coarse estimate of the rtt, we collect the measured rtts of the SYN-ACK and request-reply rounds and compute the 5th percentile of their values. To get the interpacket spacing at the bottleneck, we use the bandwidth estimation technique in the previous section.

For each packet arrival, we compute the inter-

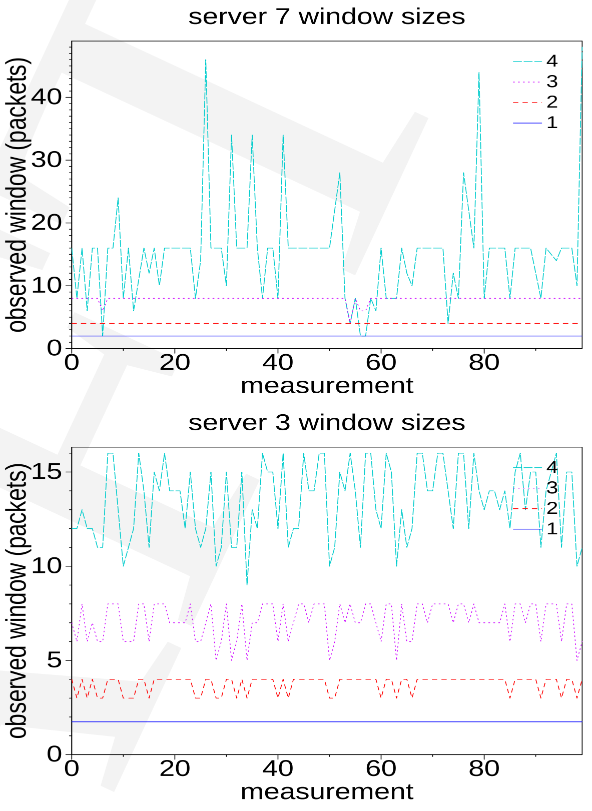


Figure 5: Window sizes for the first four rounds of slow start, for 100 HTTP connections.

packet spacing we expect at the bottleneck bandwidth, $inter = ds_i/bbw$. Then we compute a logarithmic transformation of the interarrival time, $dt'_i = f(dt_i)$, scaled so that $dt'_i = 0$ if $dt_i = inter$ and $dt'_i = 1$ if $dt_i = rtt$. This transformation gives us a criterion for breaking a timing chart into rounds; if $dt'_i > 0.5$, we consider the ith packet to be the beginning of a new round. During slow start, the breaks between rounds are obvious and the choice of this threshold has little effect. As the congestion window approaches the bdp, it becomes impossible (and meaningless) to indentify rounds. In the next few sections, we will limit the discussion to cases where the breaks between rounds are unambiguous. Later, we consider ways to characterize steady state behavior.

Figure 5(top) shows a server with the kind of slow start behavior we expect. The first round is always 2 packets, the second is always 4, and the third is usually 8, except in a few cases where, it seems, a packet drop causes TCP to switch to congestion avoidance. By the fourth round, the congestion window has reached the bdp, which is about 15 packets, and it is no longer possible to identify the breaks between rounds accurately.

Although this behavior is comprehensible, it is not typical. In most of our datasets, the behavior of the congestion windows turns out to be nondeterministic. Figure 5(bottom) is typical. The initial congestion window is consistently 2524 B, or 1.74 packets. But the second round is sometimes 3 and sometimes 4 packets. The third round is usually twice the second, but again, it sometimes falls short by a packet. The same thing happens in the next round; the congestion window either doubles or falls short by a packet, seemingly at random.

This behavior is not an abberration. Of our 13 datasets, 10 show significant nondeterminism starting in the second or third round and continuing in subsequent rounds. The window sizes for all datasets are available from `www.caida.org/performance/tcp/wins`.

The most likely explanation of this behavior is an interaction between the delayed ACK mechanism at the receiver and the growth of the congestion window at the sender [27]. In steady state, most receivers send one ACK for every other packet. Other ACKs are delayed until the next packet arrives or until a timer expires.

During slow start, many senders increase the congestion window by one mss for each new ACK that arrives. If the receiver acknowledges every other packet, the congestion window tends to grow by a factor of 1.5 per round, rather than the canonical factor of 2. To avoid this effect, many receivers modify the delayed ACK mechanism during slow start. In general, though, the receiver does not know whether the sender is in slow start and must use heuristics.

In accordance with RFC1122, most receivers use a timer to bound the time an ACK is delayed. We suspect that this timer is the source of nondeterminism in window sizes. For our datasets, the receiver was running Red Hat Linux 7.3 (kernel version 2.4.18-3). On this system, the duration of the delayed ACK timer depends on the estimated rtt, with minimum and maximum values of 40 and 200 ms.

3.4 Round trip time

The performance of short TCP transfers depends on the distribution of rtts. In general it is trivial to measure the rtt of a network path. Tools like `ping` can generate packets of various types and sizes, and measure the time until a reply arrives. So it is no surprise that we can use TCP to estimate rtt, or that by making repeated measurements we can estimate the distribution of rtts.

A TCP connection takes at least two rtts, one for the SYN-ACK round and one for the request-reply round. If the transfer size is greater than the initial congestion window (icw), additional data rounds are required. In our datasets, we can measure the SYN-ACK and request-reply rounds directly, and after segmenting the timing chart we can estimate the rtts of the next three data rounds reliably.

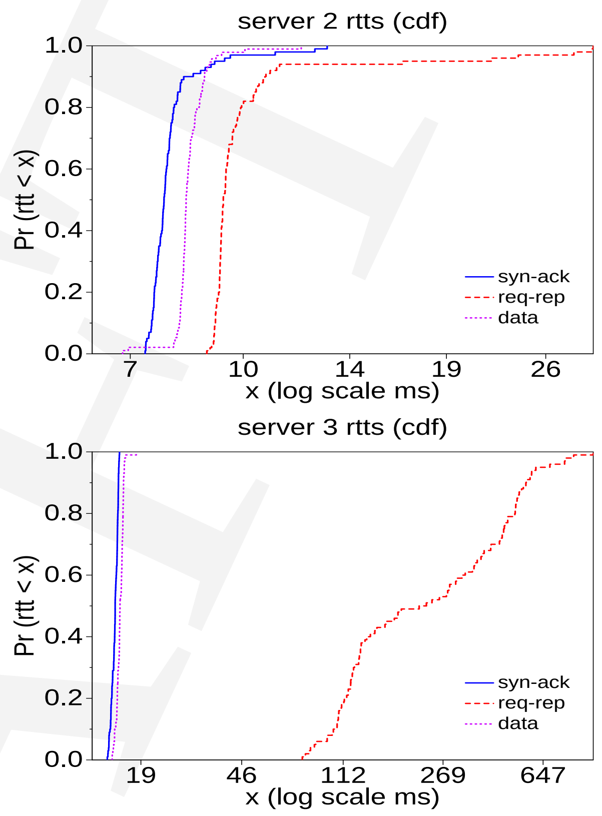


Figure 6: Distributions of round trip times for the SYN-ACK round, reply-request round, and next three data rounds of 100 HTTP connections.

Figure 6 shows distributions of rtt for SYN-ACK, reply-request round, and data rounds. Server 2 (top) is typical. The SYN-ACK round sees the shortest rtts because the packet sizes are minimal and there is no application-level processing at the server. The request-reply round takes the longest, and has the highest variability, because the application-level processing at the server is synchronous and may require disk access.

In two of our datasets, the rtts for the request-reply round are 2–10 times longer than the network rtt. Server 3 (Figure 6 bottom) is an example. Clearly for this kind of application, a model of TCP performance needs to include a model of application-level performance. After the request-reply round, the servers we observed seem to keep up with the network. Even in high-bandwidth, low-rtt paths, the

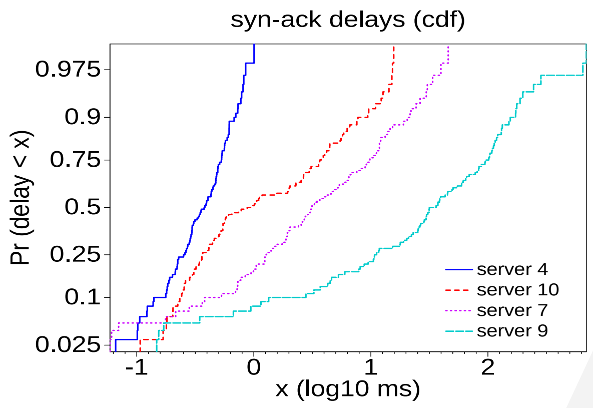


Figure 7: Distributions of delays for the SYN-ACK round, for four servers. The axes are transformed so that a lognormal distribution appears as a straight line.

subsequent data rounds happen at network speed.

Server 3 is the one exception. After the first 40 packets each transfer is delayed by roughly 50 ms (a few are shorter and a few are as long as 400 ms). Since the distribution of these delays is nothing like the distribution of rtts, we conclude that they are caused by the server. In general, though, our datasets do not distinguish between network and server delays. Barford and Crovella address this problem more successfully using network-level traces [6].

There are two ways to characterize the variability we see in rtt from one connection to the next. One is simply to record the empirical cdf of rtts. The other is to summarize the distribution of rtts using moments or other summary statistics. In general, the distribution of network delays is heavily skewed. Therefore, moments computed from samples do a poor job of characterizing the shape of the distribution. In many cases, the distribution of rtts is well-described by a three-parameter lognormal model; in other words, the distribution is characterized by a minimal value, which is the latency of the network path, and a distribution of delays, which tends to be normal on a log axis.

For a given set of measured rtt_i, we estimate the minimal value, $\theta = \min rtt_i$, and then compute the distribution of delays, $delay_i = rtt_i - \theta$. By plotting the distribution of delays, we can see whether the lognormal model is appropriate. Figure 7 shows the distribution of delays for the SYN-ACK round for four servers with a range of variability. The axes have been transformed so that a lognormal distribution appears as a straight line. The actual distributions are only approximately straight, but they are close enough that we think the lognormal model summarizes them well.

Server	θ ms	ζ log_{10} ms	σ log_{10} ms	E[delay] ms
1	271.564	1.198	0.545	3.842
2	7.351	-0.379	0.424	0.749
3	14.247	-0.114	0.292	0.931
4	75.239	-0.547	0.316	0.608
5	6.648	-0.253	0.126	0.783
6	6.611	0.087	0.746	1.441
7	24.471	0.505	0.703	2.120
8	87.689	-0.414	0.598	0.790
9	37.593	1.452	0.833	6.038
10	50.942	0.009	0.662	1.256
11	227.368	1.218	0.870	4.937
12	239.564	1.960	0.238	7.303
13	50.225	-0.541	0.301	0.609

Table 2: Estimated parameters for the distribution of rtts in the SYN-ACK round for 13 servers.

Table 2 shows the estimated parameters for the SYN-ACK round for each server. The expected value of delay is computed as $E[delay] = exp(\zeta + \sigma^2/2)$.

3.5 Correlations

The duration of short TCP transfers is the sum of a series of consecutive rtts. Therefore, correlation between successive rtts will affect the distribution of transfer times. In general the strength of correlation depends on the interval between packets. For modeling TCP performance, the relevant interval is the rtt of the path.

Bolot characterized the relationship between the rtts of successive packets and found that correlations diminish as the timescale increases, and disappear when the interval between packets exceeds 500ms [7]. Moon et al. estimate the autocorrelation function for series of RTP packets and find strong correlations that diminish over larger intervals, again becoming insignificant beyond 500ms [26].

By breaking our observations of slow start into a series of rtts, we can use our measurements to estimate correlations between successive rounds. For each round, we compute the cdf of all rtts seen during that round. Then for each timing chart, we find the rank of each observed rtt in the cdf for its round. Next we transform the ranks using the inverse of the normal distribution, and then compute Pearson's correlation. Table 3 shows the estimated correlations for the first four rounds, for each server. Values in parentheses are statistically insignificant at 90% confidence.

Server	syn-ack req-rep	req-rep data 1	data 1 data 2	data 2 data 3
1	0.203	0.180	0.220	0.364
2	(-0.113)	(0.126)	(0.076)	(-0.108)
3	(-0.082)	(-0.131)	0.702	0.770
4	(0.096)	0.265	(-0.034)	(-0.008)
5	(0.116)	(-0.121)	(0.088)	(0.092)
6	0.671	0.640	0.736	0.713
7	0.396	0.334	0.310	0.326
8	(0.044)	(-0.020)	(-0.185)	(-0.269)
9	0.721	0.682	0.657	0.638
10	0.184	(0.081)	(0.157)	(-0.144)
11	0.811	0.745	0.874	0.915
12	0.631	0.625	0.875	0.895
13	(-0.040)	(0.101)	(-0.047)	(-0.043)

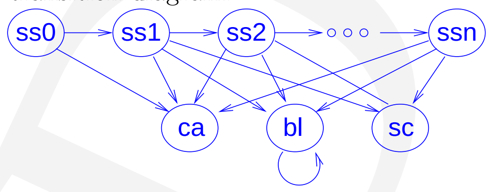
Table 3: Correlations in rtt for the first four rounds of TCP connections.

About half of the servers show significant correlations; these correlations are usually consistent from round to round. For Server 3, the correlation between data rounds is above 0.7, but the correlations in the first two rounds are insignificant because the duration of the req-rep round is limited by the server and unrelated to network conditions.

Based on prior work, we might expect higher correlations on paths with shorter rtts, but that is not the case. There is no apparent relationship between rtt and the degree of correlation. On the other hand, paths with high variability (see E[delay] in Table 2) tend to have high correlations. This result makes sense, since paths with longer delays are more likely to have queues that persist long enough to induce correlations over longer timescales.

4 Performance model

Finally we are ready to assemble a model of TCP performance. The model is based on the following state transition diagram:



The states labeled $ss0$ through ssn are slow start states; the states labeled ca, sb and sc represent congestion avoidance, server bound, and self-clocking states. By looking at the timing chart for each transfer, we try to identify the sequence of states the transfer went through.

For each transfer, we have a series of window sizes, w_i, computed as in Section3.3. All transfers start in $ss0$. If w_0 is less than 5.0, we move to $ss1$; a larger window probably indicates a dropped packet, so we move to ca.

For subsequent rounds, we compute the ratio of successive window sizes, w_i/w_{i-1}. If this ratio is between 1.5 and 2.0, we move to the next slow start state. If it falls short of 1.5, we assume that a dropped packet caused the congestion window to shrink and we move to ca. If we see evidence of a dropped packet (see Section 2), we also move to ca. Finally, if the window exceeds bdp, we move to sc.

For each slow start state, we keep track of the distribution of window sizes in that state.

Once a transfer leaves slow start, we compute the average throughput between the end of the last round of slow start and the end of the timing chart. For each of the states ca, sb and sc, we keep track of the distribution of throughputs.

Thus, for each timing chart, we compute a series of states that starts in $ss0$ and ends in one of ca, sb or sc. By counting the state transitions in these paths, we estimate the probability of each state transition.

This model now contains all the information we need to compute the distribution of transfer times for a given transfer size. By making a random walk through the state transition diagram, we can generate a single estimate of the transfer time. By making repeated walks, we can estimate the distribution. The next section explains this process in more detail.

4.1 Estimating transfer times

In previous sections, we have shown how to use a set of timing charts to estimate the parameters of a network path. These parameters are:

- The distribution of rtts for the SYN-ACK round, the request-reply round, and the first data round.
- The correlation in rtts for the first two data rounds.
- The state transition probabilities for n slow start states plus terminal states ca, sb and sc.
- The distribution of window sizes for each slow start state.
- The distribution of throughputs for each terminal state.

These parameters are sufficient to estimate the transfer time for a given transfer size, s. Here is the algorithm in pseudocode:

1. Set s_{total}, the total data received, to 0.
2. Choose rtt_0 from the distribution of SYN-ACK rtts and rtt_1 from the distribution of request-replay rtts. Set t_{total}, the total elapsed time, to $rtt_0 + rtt_1$.
3. Start in state $ss0$.
4. Using the state transition probabilities, choose the next state at random.
5. If the new state is terminal, choose *throughput* at random from the distribution of throughputs. Compute the remaining time $t_{rem} = (s - s_{total})/throughput$ and return the sum $t_{rem} + t_{total}$.
6. Choose a window size, *win*, from the distribution of window sizes for this state. If $s_{total} + win > s$, the transfer completes during this round. Return t_{total}.
7. Update $s_{total} = s_{total} + win$.
8. Choose rtt_i from the distribution of data rtts, and update $t_{total} = t_{total} + rtt_i$.
9. Go to step 4.

By repeating this process, we can estimate the distribution of transfer times.

5 Validation

To test this model, we divided each dataset randomly into two sets of 50 transfers. We used the first subset to estimate the parameters of the model and generate a distribution of transfer times for a range of sizes. Then we compare the predicted times with the measured times from the second subset.

Figure 8 shows the results for four servers we chose to be representative. For Server 1, most transfers end in slow start. For some of the file sizes, the distribution of transfer times is multimodal because some transfers require one or more extra rtts. The model captures this behavior well. For Server 2, most transfers are buffer-limited, so the transfer time is determined by the distribution of rtts. Again, the distributions are often multimodal. For Server 9, many transfers enter congestion avoidance almost immediately, but some are self-clocking. The range of

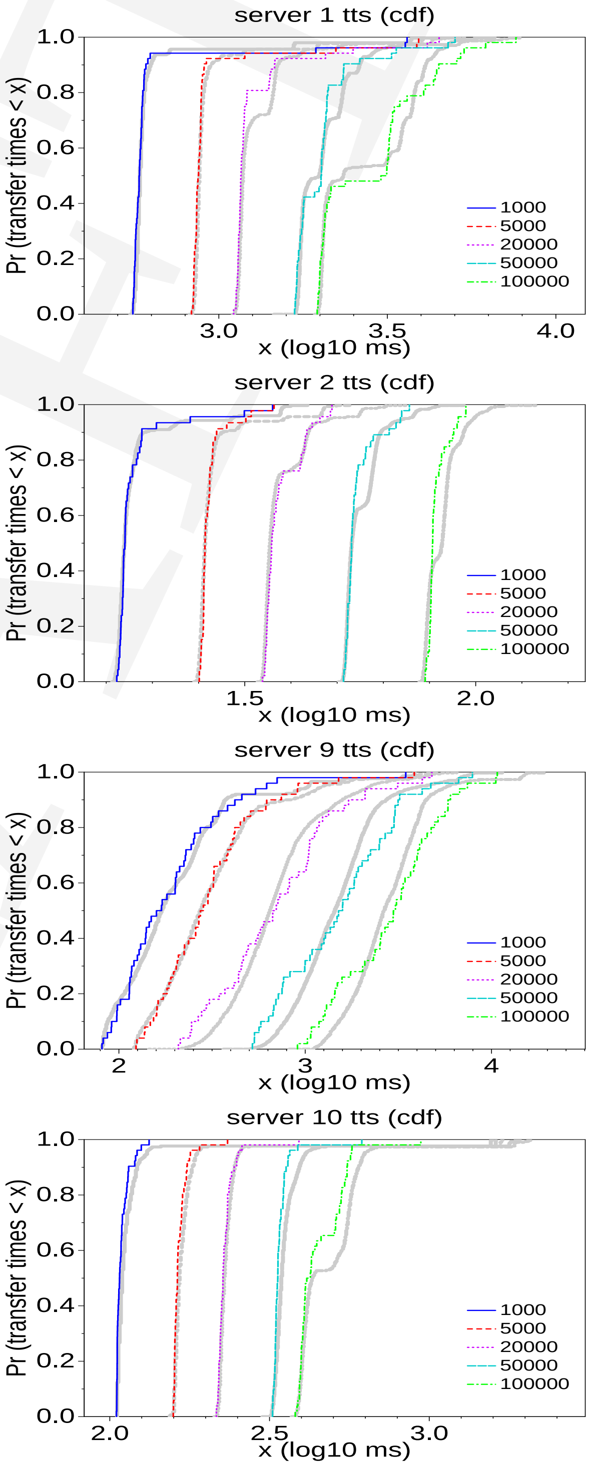


Figure 8: Distributions of predicted (thick, gray lines) and actual transfer times (thin, darker lines).

transfer times is unusually wide, but the model describes the distributions reasonably well, in location and variance if not in shape. For Server 10, most transfers are self-clocking and the range of transfer times is relatively narrow. Similar figures for all 13 datasets are available from `www.caida.org/performance/tcp/aptimes.100000`.

In two cases, the model is not as successful. As we discuss in Section 3.4, transfers from Server 3 see a delay after the first 40 packets. Since this delay isn't included in the model, our preditions for longer transfers are too short. The other problem is Server 6, which is actually not a single server, but three servers with different performance characteristics, as we discuss in Section 3.2. Our model combines the characteristics of the three servers, forming a trimodal distribution of rtts. When we convolve this distribution with itself, it gets smoother, whereas the real distribution of transfer times is sharply trimodal. Although the location and variance of the predictions is right, the shape of the distribution is not.

Although some servers present features that are not captured by the model, in general the model is able to capture the behavior of a wide range of server and network conditions.

5.1 Short measurements

A stronger test of the model is whether, given measurements of short transfers, it can predict the durations of long transfers. To simulate measurements of shorter transfers, we cut off each timing chart after the first 50,000 bytes and predict the duration of 100,000 byte transfers. Again, we divided the datasets in half, using 50 timing charts to build the model and testing it on the other 50.

Table 5.1 shows the results, comparing a goodness-of-fit measure for predictions that use the censored measurements (right column) with the predictions from the previous section (left column). A graphical representation of this data is available from `www.caida.org/performance/tcp/aptimes.50000`.

The distance metric is, roughly speaking, the normalized area between the predicted and actual distributions, computed by summing for each percentile, p, from 1 to 99:

$$\frac{|F_p^{-1}(p) - F_a^{-1}(p)|}{F_a^{-1}(p)}$$

where F_p and F_a are the actual and predicted distibution functions.

For most servers, the predictions based on censored data are reasonably good, either a little worse or, by chance, a little better than the predictions that use all the data. When the predictions fail it is because the truncated datasets don't contain enough information to classify the steady-state behavior of the server. For example, Server 2 is buffer-limited, but in the truncated dataset it never leaves slow start, so the model predicts transfer times that are too short.

Server	Distance 100,000	Distance 50,000
1	0.804	1.044
2	0.771	3.475
3	3.490	8.716
4	0.087	0.303
5	2.639	1.871
6	3.682	6.802
7	0.937	0.627
8	0.265	0.459
9	1.938	2.194
10	1.194	2.023
11	1.237	2.657
12	0.782	0.803
13	0.697	2.254

Table 4: Goodness of fit measurements for predicted transfer times, given measurements truncated after 100,000 or 50,000 bytes.

5.2 Limitations

Application-level measurements are easy to implement, and tools that use them are portable. The price of this convenience is that many of the things we would like to measure, like round trip times and window sizes, are not directly visible to an application. We have shown that it is possible infer this information with acceptable accuracy, but there are two parts of the model that would benefit from more network-level information.

The first is identifying dropped packets. The heuristics our model uses are successful in the sense that they identify characteristics in a timing chart that indicate a dropped packet, but without network-level traces we can't assess their accuracy.

The second limitation is the difficulty of distinguishing server delays from network delays. For the HTTP transfers we looked at, most server delays occur during the request-reply round; after that, the servers kept up with the network. Other kinds of TCP transfers, like ftp, may be similar, but there are other cases where a more detailed model of server performance may be necessary.

Finally, an aspect of TCP performance that we left out of the model is the effect of dropped packets at the end of a transfer. In our datasets, most dropped packets were caught by the Fast Retransmit mechanism, so they tended not to impose long delays, except indirectly by reducing the congestion window. When a packet is dropped at the end of a transfer, there may not be enough ACK packets to trigger Fast Retransmit, and a transfer may suffer a timeout. In our datasets, these events are rare, but their effect is significant.

6 Conclusions

We propose a model of TCP performance that captures the behavior of a diverse set of network paths. The features of this model are:

- It includes a model of slow start and three common steady-state behaviors: congestion avoidance, buffer-limited, and self-clocking. Thus, it is applicable to a wide range of transfer sizes.
- It predicts the entire distribution of transfer times, rather than just the expected value.
- In accordance with our observations, it treats the growth of the congestion window during slow start as a stochastic process.
- Rather than estimate the exogenous drop rate explicitly, the model incorporates both exogenous and endogenous drops in an array of state transition probabilities.
- It involves more parameters than previous models, but we show that these parameters can be estimated with a reasonable number of simple, application-level measurements.
- The model includes a novel and promising way of estimating bottleneck bandwidth based on packet spacing.
- It is applicable to all current and most conceivable implementations of TCP. In some cases, its estimated parameters can be used to identify the implementation of the sender or receiver.

6.1 Future Work

Our experiments show that the parameters of our model are consistent; when we divide a dataset in half at random, we can use the parameters from one half to predict the performance of the other half.

The next step is to make the model predictive; that is, given a set of past measurements, we would like to predict future performance. To do that, we have to address two additional problems. The first is to find a number of measurements, and their timescale, that is sufficient to capture the stochastic properties of a network path. The second is to identify and deal with the nonstationarity induced by path changes and other large-timescale variability. Fortunately, several previous projects have addressed these problems, including the Network Weather Service (NWS) [35]. Our next step is to incorporate our model into the NWS.

In Section 3.2, we propose a way of filtering packet-train measurements to improve the accuracy of bottleneck bandwidth estimation. We have shown that this technique improves the repeatability of estimates, but we don't know about their accuracy. We plan to incorporate it into existing bandwidth estimation tools for a more rigorous evaluation.

References

[1] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. In *SIGCOMM*, pages 263–274, 1999.

[2] Eitan Altman, Konstantin Avrachenkov, Chadi Barakat, and Rudesindo Nu nez Queija. TCP modeling in the presence of nonlinear window growth. Technical Report RR-4312, INRIA, Novemeber 2001.

[3] M. Aron and P. Druschel. TCP: Improving startup dynamics by adaptive timers and congestion control. Technical report, Dept. of Computer Science, Rice University, 1998.

[4] Chadi Barakat and Eitan Altman. Performance of short TCP transfers. In *NETWORKING*, volume 1815 of *Lecture Notes in Computer Science*, pages 567–579. Springer, 2000.

[5] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in web client access patterns: Characteristics and caching implications. *World Wide Web, Special Issue on Characterization and Performance Evaluation*, 2:15–28, 1999.

[6] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. In *SIGCOMM*, pages 127–138, 2000.

[7] Jean-Chrysostome Bolot. Characterizing end-to-end packet delay and loss in the internet. *Journal of High Speed Networks*, 2(3):289–298, September 1993.

[8] Lawrence S. Brakmo and Larry L. Peterson. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[9] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling TCP latency. In *INFOCOM (3)*, pages 1742–1751, 2000.

[10] Robert Carter and Mark Crovella. Measuring bottleneck link speed in packet-switched networks. Technical Report TR-1996-006, Computer Science Dept., Boston University, 1996.

[11] Claudio Casetti and Michela Meo. A new approach to model the stationary behavior of TCP connections. In *INFOCOM (1)*, pages 367–375, 2000.

[12] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. Packet dispersion techniques and capacity estimation. Submitted to submitted to IEEE/ACM Transactions in Networking.

[13] A. Fekete and G. Vattay. On buffer limited congestion window dynamics and packet loss. In *IEEE Internet Performance Symposium*, November 2002.

[14] Sally Floyd. Connections with multiple congested gateways in packet-switched networks part 1: One-way traffic. *ACM Computer Communication Review*, 21(5):30–47, October 1991.

[15] Sally Floyd and Van Jacobson. Traffic phase effects in packet-switched gateways. *Journal of Internetworking: Practice and Experience*, 3(3):115–156, September 1992.

[16] Janey C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *ACM SIGCOMM*, pages 270–280, 1996.

[17] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In *SIGCOMM*, pages 314–329, 1988.

[18] S. Keshav. A control-theoretic approach to flow control. In *ACM SIGCOMM*, pages 3–15, 1991.

[19] Kevin Lai and Mary Baker. Measuring bandwidth. In *IEEE INFOCOM*, pages 235–245, 1999.

[20] 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*, 5(3):336–350, July 1997.

[21] T. V. Lakshman, Upamanyu Madhow, and Bernhard Suter. Window-based error recovery and flow control with a slow acknowledgement channel: A study of TCP/IP performance. In *INFOCOM (3)*, pages 1199–1209, 1997.

[22] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communications Review*, 27(3), 1997.

[23] Marco Mellia, Ion Stoica, and Hui Zhang. Tcp model for short lived flows. *IEEE Communications Letters*, 6(2):85–88, February 2002.

[24] Archan Misra and Teunis J. Ott. The window distribution of idealized TCP congestion avoidance with variable packet loss. In *INFOCOM (3)*, pages 1564–1572, 1999.

[25] Vishal Misra, W. Gong, and Don Towsley. Stochastic differential equation modeling and analysis of tcp windowsize behavior. Technical Report ECE-TR-CCS-99-10-01, University of Massachusetts, 1999.

[26] S. B. Moon, J. Kurose, P. Skelly, and D. Towsley. Correlation of packet delay and loss in the Internet. Technical Report UM-CS-1998-011, University of Massachusetts, March 1998.

[27] Wael Noureddine and Fouad Tobagi. The transmission control protocol. Technical report, Stanford University, July 2002.

[28] T. Ott, J. Kemperman, and M. Mathis. Window size behavior in tcp/ip with constant loss probability. In *IEEE HPCS*, June 1997.

[29] Jitedra Padhye, Victor Firoiu, and Don Towsley. A stochastic model of TCP reno congestion avoidance and control. Technical Report CMP-SCI 99-02, University of Massachusetts, 1999.

[30] Jitedra Padhye, Victor Firoiu, Don Towsley, and Jim Krusoe. Modeling TCP throughput: A simple model and its empirical validation. In *ACM SIGCOMM*, pages 303–314, 1998.

[31] Craig Partridge, Dennis Rockwell, Mark Allman, Rajesh Krishnan, and James Sterbenz. A swifter start for tcp. Technical Report 8339, BBN, March 2002.

[32] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[33] B. Sikdar, S. Kalyanaraman, and K. Vastola. Analytic models for the latency and steady-state throughput of TCP tahoe, reno and SACK. In *IEEE GLOBECOM*, pages 1781–1787, November 2001.

[34] B. Sikdar, S. Kalyanaraman, and K. Vastola. TCP Reno with random losses: Latency, throughput and sensitivity analysis. In *IEEE IPCCC*, pages 188–195, April 2001.

[35] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.