Nettimer: A Tool for Measuring Bottleneck Link Bandwidth

Kevin Lai Mary Baker
{laik, mgbaker}@cs.stanford.edu
Department of Computer Science, Stanford University

January 29, 2001

Abstract

Measuring the bottleneck link bandwidth along a path is important for understanding the performance of many Internet applications. Existing tools to measure bottleneck bandwidth are relatively slow, can only measure bandwidth in one direction, and/or actively send probe packets. We present the `nettimer` bottleneck link bandwidth measurement tool, the `libdpcap` distributed packet capture library, and experiments quantifying their utility. We test `nettimer` across a variety of bottleneck network technologies ranging from 19.2Kb/s to 100Mb/s, wired and wireless, symmetric and asymmetric bandwidth, across local area and cross-country paths, while using both one and two packet capture hosts. In most cases, `nettimer` has an error of less than 10%, but at worst has an error of 40%, even on cross-country paths of 17 or more hops. It converges within 10KB of the first large packet arrival while consuming less than 7% of the network traffic being measured.

1 Introduction

Network bandwidth continues to be a critical resource in the Internet because of the heterogeneous bandwidths of access technologies and file sizes. This can cause an unaware application to stream a 5GB video file over a 19.2Kb/s cellular data link or send a text-only version of a web site over a 100Mb/s link. Knowledge of the bandwidth along a path allows an application to avoid such mistakes by adapting the size and quality of its content [FGBA96] or by choosing a web server or proxy with higher bandwidth than its replicas [Ste99].

Existing solutions to this problem have examined HTTP throughput [Ste99], TCP throughput [MM96], available bandwidth [CC96a], or bottleneck link bandwidth. Although HTTP and TCP are the current dominant application and transport protocols in the Internet, other applications and transport protocols (e.g. for video and audio streaming) have different performance characteristics. Consequently, their performance cannot be predicted by HTTP and TCP throughput. Available bandwidth (when combined with latency, loss rates, and other metrics) can predict the performance of a wide variety of applications and transport protocols. However, available bandwidth depends on both bottleneck link bandwidth and cross traffic. Cross traffic is highly variable in different places in the Internet and even highly variable in the same place. Developing and verifying the validity of an available bandwidth algorithm that deals with that variability is difficult.

In contrast, bottleneck link bandwidth is well understood in theory [Kes91] [Bol93] [Pax97] [LB00], and techniques to measure it are straightforward to validate in practice (see Section 4). Moreover, bottleneck link bandwidth measurement techniques have been shown to be accurate and fast in simulation [LB99]. Furthermore, in some parts of the Internet, available bandwidth is frequently equal to bottleneck link bandwidth because either bottleneck link bandwidth is small (e.g. wireless, modem, or DSL) or cross traffic is low (e.g. LAN). In addition to bottleneck link bandwidth's current utility, it can help the development of accurate and validated available bandwidth measurement techniques because of available bandwidth's dependence on bottleneck link bandwidth.

However, current tools to measure link bandwidth 1) measure all link bandwidths instead of just the bottleneck, 2) only measure the bandwidth in one direction, and/or 3) actively send probe packets. The tools `pathchar` [Jac97], `clink` [Dow99], `pchar` [Mah00], and `tailgater` [LB00] measure all of the link bandwidths along a path, which can be time-consuming and unnecessary for applications that only want to know the bottleneck bandwidth. Furthermore, these tools and `bprobe` [CC96b] can only measure bandwidth in one direction. These tools, `tcpanaly` [Pax97], and `pathrate`

[DRM01] actively send their own probe traffic, which can be more accurate than passively measuring existing traffic, but also results in higher overhead [LB00]. The `nettimer-sim` [LB99] tool only works in simulation.

Our contributions are the `nettimer` bottleneck link bandwidth measurement tool, the `libdpcap` distributed packet capture library, and experiments quantifying their utility. Unlike current tools, `nettimer` can passively measure the bottleneck link bandwidth along a path in real time. `Nettimer` can measure bandwidth in one direction with one packet capture host and in both directions with two packet capture hosts. In addition, the `libdpcap` distributed packet capture library allows measurement programs like `nettimer` to efficiently capture packets at remote hosts while doing expensive measurement calculations locally. Our experiments indicate that in most cases `nettimer` has less than 10% error whether the bottleneck link technology is 100Mb/s Ethernet, 10Mb/s Ethernet, 11Mb/s WaveLAN, 2Mb/s WaveLAN, ADSL, V.34 modem, or CDMA cellular data. `Nettimer` converges within 10308 bytes of the first large packet arrival. Even when measuring a 100Mb/s bottleneck, `nettimer` only consumes 6.34% of the network traffic being measured, and 4.52% of the cycles on the 366MHz remote packet capture server and 57.6% of the cycles on the 266MHz bandwidth computation machine.

The rest of the paper is organized as follows. In Section 2 we describe the packet pair property of FIFO-queueing networks and show how it can be used to measure bottleneck link bandwidth. In Section 3 we describe how we implement the packet pair techniques described in Section 2, including our distributed packet capture architecture and API. In Section 4, we present preliminary results quantifying the accuracy, robustness, agility, and efficiency of the tool. In Section 6, we conclude.

2 Packet Pair Technique

In this section we describe the packet pair property of FIFO-queueing networks and show how it can be used to measure bottleneck link bandwidth.

2.1 Packet Pair Property of FIFO-Queueing networks

The packet pair property of FIFO-queueing networks predicts the difference in arrival times of two packets of the same size traveling from the same source to the same destination:

$$t_n^1 - t_n^0 = \max\left(\frac{s_1}{b_l}, t_0^1 - t_0^0\right) \quad (1)$$

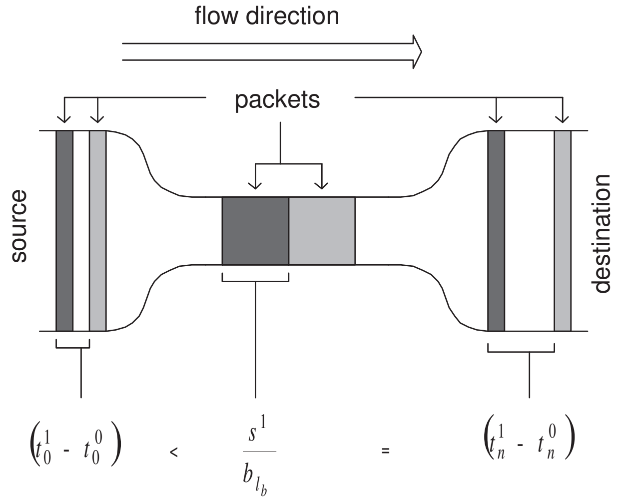

Figure 1: This figure shows two packets of the same size traveling from the source to the destination. The wide part of the pipe represents a high bandwidth link while the narrow part represents a low bandwidth link. The spacing between the packets caused by queueing at the bottleneck link remains constant downstream because there is no additional downstream queueing.

where t_n^0 and t_n^1 are the arrival times of the first and second packets respectively at the destination, t_0^0 and t_0^1 are the transmission times of the first and second packets respectively, s_1 is the size of the second packet, and b_l is the bandwidth of the bottleneck link.

The intuitive rationale for this equation (a full proof is given in [LB00]) is that if two packets are sent close enough together in time to cause the packets to queue together at the bottleneck link ($\frac{s_1}{b_l} > t_0^1 - t_0^0$), then the packets will arrive at the destination with the same spacing ($t_n^1 - t_n^0$) as when they exited the bottleneck link ($\frac{s_1}{b_l}$). The spacing will remain the same because the packets are the same size and no link downstream of the bottleneck link has a lower bandwidth than the bottleneck link (as shown in Figure 1, which is a variation of a figure from [Jac88]).

This property makes several assumptions that may not hold in practice. First, it assumes that the two packets queue together at the bottleneck link and at no later link. This could by violated by other packets queueing between the two packets at the bottleneck link, or packets queueing in front of the first, the second or both packets downstream of the bottleneck link. If any of these events occur, then Equation 1 does not hold. In Section 2.2, we describe how to mitigate this limitation by filtering out samples that suffer undesirable queueing.

In addition, the packet pair property assumes that the two packets are sent close enough in time that they queue together at the bottleneck link. This is a problem for very high bandwidth bottleneck links and/or for passive measurement. For example, from Equation 1, to cause queueing between two 1500 byte packets at a 1Gb/s bottleneck link, they would have to be transmitted no more than 12 microseconds apart. An active technique is more likely than a passive one to satisfy this assumption because it can control the size and transmission times of its packets. However, in Section 2.2, we describe how passive techniques can detect this problem and sometimes filter out its effect.

Another assumption of the packet pair property is that the bottleneck router uses FIFO-queueing. If the router uses fair queueing, then packet pair measures the available bandwidth of the bottleneck link [Kes91].

Finally, the packet pair property assumes that transmission delay is proportional to packet size and that routers are store-and-forward. The assumption that transmission delay is proportional to packet size may not be true if, for example, a router manages its buffers in such a way that a 128 byte packet is copied more than proportionally faster than a 129 byte packet. However, this effect is usually small enough to be ignored. The assumption that routers are store-and-forward (they receive the last bit of the packet before forwarding the first bit) is almost always true in the Internet.

Using the packet pair property, we can solve Equation 1 for b_l, the bandwidth of the bottleneck link:

$$b_l = \frac{s_1}{t_n^1 - t_n^0} \tag{2}$$

We call this the *received* bandwidth because it is bandwidth measured at the receiver. When filtering in the next section, we will also use the the bandwidth measured at the sender (the *sent* bandwidth):

$$\frac{s_1}{t_0^1 - t_0^0} \tag{3}$$

2.2 Filtering Techniques

In this section, we describe in more detail how the assumptions in Section 2.1 can be violated in practice and how we can filter out this effect. Using measurements of the sizes and transmission and arrival times of several packets and Equation 1, we can get samples of the received bandwidth. The goal of a filtering technique is to determine which of these samples indicate the bottleneck link bandwidth and which do not. Our approach is to develop a filtering function that gives higher priority to the good samples and lower priority to the bad samples.

Before describing our filtering functions, we differentiate between the kinds of samples we want to keep and those we want to filter out. Figure 2 shows one case that satisfies the assumptions of the packet pair property and three cases that do not. There are other possible scenarios but they are combinations of these cases.

Case A shows the ideal packet pair case: the packets are sent sufficiently quickly to queue at the bottleneck link and there is no queueing after the bottleneck link. In this case the bottleneck bandwidth is equal to the received bandwidth and we do not need to do any filtering.

In case B, one or more packets queue between the first and second packets, causing the second packet to fall farther behind than would have been caused by the bottleneck link. In this case, the received bandwidth is less than the bottleneck bandwidth by some unknown amount, so we should filter this sample out.

In case C, one or more packets queue before the first packet after the bottleneck link, causing the second packet to follow the first packet closer than would have been caused by the bottleneck link. In this case, the received bandwidth is greater than the bottleneck bandwidth by some unknown amount, so we should filter this sample out.

In case D, the sender does not send the two packets close enough together, so they do not queue at the bottleneck link. In this case, the received bandwidth is less than the bottleneck bandwidth by some unknown amount, so we should filter this sample out. Active techniques can avoid case D samples by sending large packets with little spacing between them, but passive techniques are susceptible to them. Examples of case D traffic are TCP acknowledgements, voice over IP traffic, remote terminal protocols like telnet and ssh, and instant messaging protocols.

2.2.1 Filtering using Density Estimation

To filter out the effect of case B and C, we use the insight that samples influenced by cross traffic will tend not to correlate with each other while the case A samples will correlate strongly with each other [Pax97] [CC96b]. This is because we assume that cross traffic will have random packet sizes and will arrive randomly at the links along the path. In addition, we use the insight that packets sent with a low bandwidth that arrive with a high bandwidth are definitely from case C and can be filtered out [Pax97]. Figure 3 shows a hypothetical example of how we apply these insights. Using the second insight, we eliminate the case C samples above the received bandwidth = sent bandwidth ($x = y$) line. Of the remaining samples, we calculate their smoothed distribution and pick the point with the highest density as the bandwidth.

There are many ways to compute the density function of a set of samples [Pax97] [CC96b], including us-

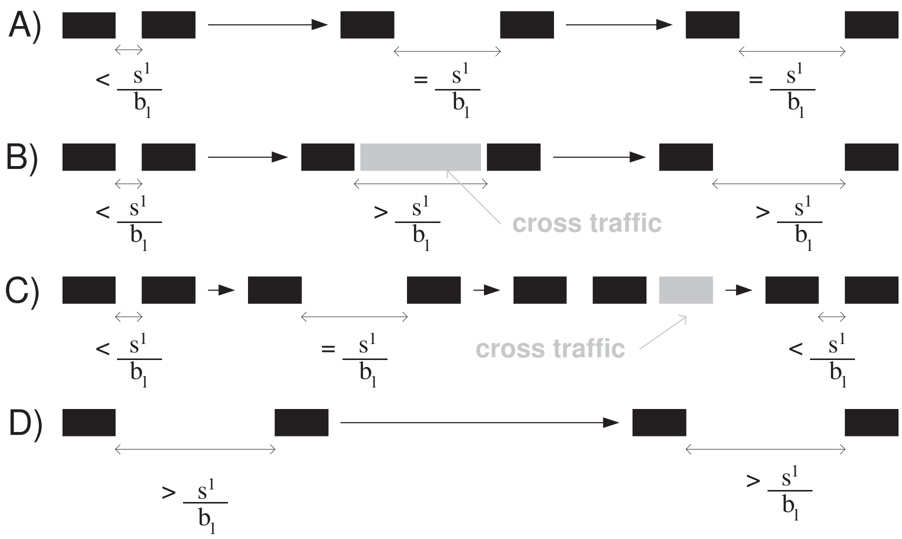

Figure 2: This figure shows four cases of how the spacing between a pair of packets changes as they travel along a path. The black boxes are packets traveling from a source on the left to a destination on the right. Underneath each pair of packets is their spacing relative to the spacing caused by the bottleneck link. They gray boxes indicate cross traffic that causes one or both of the packets to queue.

ing a histogram. However, histograms have the disadvantages of fixed bin widths, fixed bin alignment, and uniform weighting of points within a bin. Fixed bin widths make it difficult to choose an appropriate bin width without previously knowing something about the distribution. For example, if all the samples are around 100,000, it would not be meaningful to choose a bin width of 1,000,000. On the other hand, if all the samples are around 100,000,000, a bin width of 10,000 could flatten an interesting maxima. Another disadvantage is fixed bin alignment. For example, two points could lie very close to each other on either side of a bin boundary and the bin boundary ignores that relationship. Finally, uniform weighting of points within a bin means that points close together will have the same density as points that are at opposite ends of a bin. The advantage of a histogram is its speed in computing results, but we are more interested in accuracy and robustness than in saving CPU cycles.

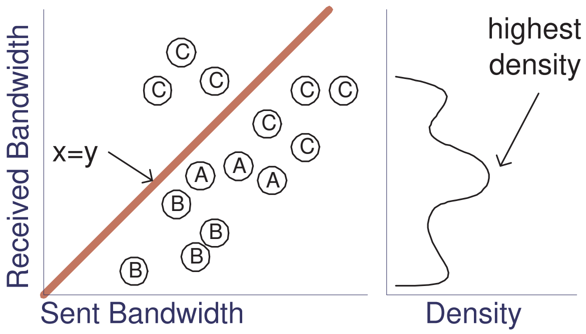

Figure 3: The left graph shows some packet pair samples plotted using their received bandwidth against their sent bandwidth. "A" samples correspond to case A, etc. The right graph shows the distribution of different values of received bandwidth after filtering out the samples above the $x = y$ line. In this example, density estimation indicates the best result.

To avoid these problems, we use *kernel density estimation* [Sco92]. The idea is to define a kernel function $K(t)$ with the property

$$\int_{-\infty}^{+\infty} K(t)dt = 1 \tag{4}$$

Then the density at a received bandwidth sample x is

$$d(x) = \frac{1}{n} \sum_{i=1}^{n} K\left(\frac{x - x_i}{c * x}\right) \tag{5}$$

where c is the kernel width ratio, n is the number of points within $c * x$ of x, and x_i is the ith such point. We use the kernel width ratio to control the smoothness of the density function. Larger values of c give a more accurate result, but are also more computationally expensive. We use a c of 0.10. The kernel function we use is

$$K(t) = \left\{ \begin{array}{ll} 1 + t & t \leq 0 \\ 1 - t & t > 0 \end{array} \right\} \tag{6}$$

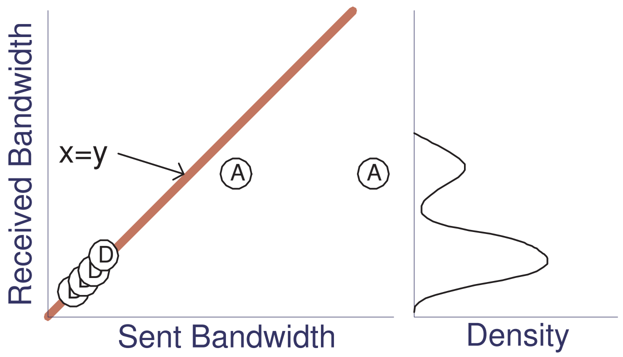

Figure 4: This figure has the same structure as Figure 3. In this example, the ratio of received bandwidth to sent bandwidth is a better indicator than density estimation.

This function gives greater weight to samples close to the point at which we want to estimate density, and it is simple and fast to compute.

2.2.2 Filtering using the Received/Sent Bandwidth Ratio

Although density estimation is the best indicator in many situations, many case D samples can fool density estimation. For example, a host could transfer data in two directions over two different TCP connections to the same correspondent host. If data mainly flows in the forward direction, then the reverse direction would consist of many TCP acknowledgements sent with large spacings and a few data packets sent with small spacings. Figure 4 shows a possible graph of the resulting measurement samples. Density estimation would indicate a bandwidth lower than the correct one because there are so many case D samples resulting from the widely spaced acks.

We can improve the accuracy of our results if we favor samples that show evidence of actually causing queueing at the bottleneck link [LB99]. The case D samples are unlikely to have caused queueing at the bottleneck link because they are so close to the line $x = y$. On the other hand, the case A samples are on average far from the $x = y$ line, meaning that they were sent with a high bandwidth but received with a lower bandwidth. This suggests that they did queue at the bottleneck link.

Using this insight we define the received/sent bandwidth ratio of a received bandwidth sample x to be

$$p(x) = 1 - \frac{ln(x)}{ln(s(x))} \tag{7}$$

where $s(x)$ is the sent bandwidth of x. We take the log of the bandwidths because bandwidths frequently differ by orders of magnitude.

Unfortunately, given two samples with the same sent bandwidth (7) favors the one with the smaller received bandwith. To counteract this, we define the received bandwidth ratio to be

$$r(x) = \frac{ln(x) - ln(x_{min})}{ln(x_{max}) - ln(x_{min})} \tag{8}$$

2.2.3 Composing Filtering Algorithms

We can compose the filtering algorithms described in the previous sections by normalizing their values and taking their linear combination:

$$f(x) = 0.4 * \frac{d(x)}{d(x)_{max}} + 0.3 * p(x) + 0.3 * r(x) \tag{9}$$

where $d(x)_{max}$ is the maximum kernel density value. By choosing the maximum value of $f(x)$ as the bottleneck link bandwidth, we can take into account both the density and the received/sent bandwidth ratio without favoring smaller values of x. The weighting of each of the components is arbitrary. Although it is unlikely that this is the optimal weighting, the results in Section 4.2 indicate that these weightings work well.

2.3 Sample Window

In addition to using a filtering function, we also only use the last w bandwidth samples. This allows us to quickly detect changes in bottleneck link bandwidth (*agility*) while being resistant to cross traffic (*stability*). A large w is more stable than a small w because it will include periods without cross traffic and different sources of cross traffic that are unlikely to correlate with a particular received bandwidth. However, a large w will be less agile than a small w for essentially the same reason. It is not clear to us how to define a w for all situations, so we currently punt on the problem by making it a user-defined parameter in `nettimer`.

3 Implementation

In this section, we describe how `nettimer` implements the algorithms described in the previous section. The issues we address are how to define flows, where to take measurements, and how to distribute measurements.

3.1 Definition of Flows

In Section 2.1, the packet pair property refers to two packets from the same source to the same destination. For `nettimer`, we interpret this flow to be defined by a `(source IP address, destination IP address)` tuple (network level flow), but we could also have interpreted it to be defined by a `(source IP address,` `source port number, destination IP address, source port number)` tuple (transport level flow). The advantage of using transport level flows is that they can penetrate Network Address Translation (NAT) gateways. The advantage of network level flows is that we can aggregate the traffic of multiple transport level flows (e.g. TCP connections) so that we have more samples to work with. We chose network level flows because when we started implementing `nettimer`, NAT gateways were not widespread while popular WWW browsers would open several short TCP connections with servers. We describe a possible solution to this problem in Section 5.

3.2 Measurement Host

In Section 2.1, we assume that we have the transmission and arrival times of packets. In practice, this requires deploying measurement software at both the sender and the receiver, which may be difficult. In this section, we describe how we mitigate this limitation in `nettimer` and the trade-offs of doing so.

3.2.1 Two Hosts

In the ideal case, we can deploy measurement software at both the sender and the receiver. Using this technique, called Receiver Based Packet Pair (RBPP) [Pax97], `nettimer` can employ all of the filtering algorithms described in Section 2.2 because we have both the transmission times and reception times. However, in addition to deploying measurement software at both the sender and the receiver, `nettimer` also needs an architecture to distribute the measurements to interested hosts (described in Section 3.3). We show in Section 4 that RBPP is the most accurate technique.

3.2.2 One Host

When we can only deploy software at one host, we measure the bandwidth from that host to any other host using Sender Based Packet Pair (SBPP) [Pax97] or from any other host to the measurement host using Receiver Only Packet Pair (ROPP) [LB99].

SBPP works by using the arrival times of transport- or application-level acknowledgements instead of the arrival times of the packets themselves. One application of this technique would be to deploy measurement software at a server and measure the bandwidth from the server to clients where software could not be deployed. The issues with this technique are 1) transport- or application-level information, 2) non-per-packet acknowledgements, and 3) susceptibility to reverse path cross traffic. `nettimer` uses transport- or application-level information to match acknowledgements to packets. Currently it only implements this functionality for TCP. Unfortunately, TCP does not have a strict per-packet acknowledgement policy. It only acks every other packet or packets out of order. Furthermore, it sometimes delays acks. Finally, the acks could be delayed by cross traffic on the reverse path, causing more noise for the filtering algorithm to deal with. We show in Section 4 that the non-per-packet acknowledgements make SBPP much less accurate that the other packet pair techniques. We describe a solution to this problem in Section 5.

ROPP works by using only the arrival times of packets. This prevents us from using some of the filtering algorithms described in Section 2.2 because we can no longer calculate the sent bandwidth. One application of this technique would be to deploy measurement software at a client and measure the bandwidth from servers that cannot be modified to the client. We show in Section 4 that in some cases where there is little cross traffic, ROPP is close in accuracy to RBPP.

3.3 Distributed Packet Capture

In this section, we describe our architecture to do distributed packet capture. The `nettimer` tool uses this architecture to measure both transmission and arrival times of packets in the Internet. We first explain our approach and then describe our implementation.

3.3.1 Approach

Our approach is to distinguish between packet capture servers and packet capture clients. The packet capture servers capture packet headers and then distribute them to the clients. The servers do no calculations. The clients receive the packet headers and perform performance calculations and filtering. This allows flexibility in where the packet capture is done and where the calculation is done.

Another possible approach is to do more calculation at the packet capture hosts [MJ98]. The advantage of approach is that packet capture hosts do not have to consume bandwidth by distributing packet headers.

The advantages of separating the packet capture and performance calculation are 1) reducing the CPU burden of the packet capture hosts, 2) gaining more flexibility in the kinds of performance calculations done, and 3) reducing the amount of code that has to run with root privileges. By doing the performance calculation only at the packet capture clients, the servers only capture packets and distribute them to clients. This is especially important if the packet capture server receives packets at a high rate, the packet capture server is collocated with other servers (e.g. a web server), and/or the performance calculation consumes many CPU cycles (as is the case with the filtering algorithm described in Section 2.2). Another advantage is that clients have the flexibility to change their performance calculation code without modifying the packet capture servers. This also avoids the possible security problems of allowing client code to be run on the server. Finally, some operating systems (e.g. Linux) require that packet capture code run with root privileges. By separating the client and server code, only the server runs with root privilege while the client can run as a normal user.

3.3.2 libdpcap

Our implementation of distributed packet capture is the `libdpcap` library. It is built on top of the `libpcap` library [MJ93]. As a result, the `nettimer` tool can measure live in the Internet or from `tcpdump` traces.

To start a `libdpcap` server, the application specifies the parameters `send_thresh`, `send_interval`, `filter_cmd`, and `cap_len`. `send_thresh` is the number of bytes of packet headers the server will buffer before sending them to the client. This should usually be at least the TCP maximum segment size so fewer less than full size packet report packets will sent. `send_interval` is the amount of time to wait before sending the buffered packet headers. This prevents packet headers from languishing at the server waiting for enough data to exceed `send_thresh`. The server sends the buffer when `send_interval` or `send_thresh` is exceeded. The `filter_cmd` specifies which packets should be captured by this server using the `libpcap` filter language. This can cut down on the amount of unnecessary data sent to the clients. For example, to capture only TCP packets between `cs.stanford.edu` and `eecs.harvard.edu` the `filter_cmd` would be "`host cs.stanford.edu and host harvard.stanford.edu and TCP`". `cap_len` specifies how much of each packet to capture.

To start a `libdpcap` client, the application specifies a set of servers to connect to and its own `filter_cmd`. The client sends this `filter_cmd` to the servers with whom it connects. This further restrict the types of packet headers that the client receives.

After a client connects to a server, the server responds with its `cap_len` and its clock resolution. Different machines and operating systems have different clock resolutions for captured packets. For example Linux < 2.2.0 had a resolution of 10ms, while Linux >= 2.2.0 has a resolution < 20 microseconds, almost a thousand times difference. This can make a significant difference in the accuracy of a calculation, so the server reports this clock resolution to the client.

To calculate the bandwidth consumed by the packet reports that the distributed packet capture server sends to its clients, we start with the size of each report: `cap_len` + sizeof(timestamp) (8 bytes) + sizeof(cap_len) (2 bytes) + sizeof(flags) (2 bytes). For TCP traffic, `nettimer` needs at least 40 bytes of packet header. In addition, link level headers consume some variable amount of space. To be safe, we set the capture length to 60 bytes, so each `libdpcap` packet report consumes 72 bytes. 20 of these headers fit in a 1460 byte TCP payload, so the total overhead is approximately 1500 bytes / 20 * 1500 = 5.00%. On a heavily loaded network, this could be a problem. However, if we are only interested in a pre-determined subset of the traffic, we can use the packet filter to reduce the number of packet reports. We experimentally verify this cost in Section 4.2.5 and describe some other ways to reduce it in Section 5.

Table 1: This table shows the different path characteristics used in the experiments. The Short and Long column list the number of hops from host to host for the short and long path respectively. The RTT columns list the round-trip-times of the short and long paths in ms.

Type	Short	RTT	Long	RTT
Ethernet 100 Mb/s	4	1	17	74
Ethernet 10 Mb/s	4	1	17	80
WaveLAN 2 Mb/s	3	4	18	151
WaveLAN 11 Mb/s	3	4	18	151
ADSL	14	19	19	129
V.34 Modem	14	151	18	234
CDMA	14	696	18	727

4 Experiments

In this section we describe the experiments we used to quantify the utility of `nettimer`.

4.1 Methodology

In this section we describe and explain our methodology in running the experiments. Our approach is to take `tcpdump` traces on pairs of machines during a transfer between those machines while varying the bottleneck link bandwidth, path length, and workload. We then run these traces through `nettimer` and analyze the results. Our methodology consists of 1) the network topology, 2) the hardware and software platform, 3) accuracy measurement, 4) the network application workload, and 5) the network environment.

Our network topology consists of a variety of paths (listed in Table 1) where we vary the bottleneck link technology and the length of the path. WaveLAN [wav00] is a wireless local area network technology made by Lucent. ADSL (Asymmetric Digital Subscriber Line) is a high bandwidth technology that uses

Table 2: This table shows the different software versions used in the experiments. The release column gives the RPM package release number.

Name	Version	Release
GNU/Linux Kernel	2.2.16	22
RedHat	7.0	-
tcpdump	3.4	10
tcptrace	5.2.1	1
openssh	2.3.0p1	4
nettimer	2.1.0	1

phone lines to bring connectivity into homes and small businesses. We tested the Pacific Bell/SBC [dsl00] ADSL service. V.34 is an International Telecommunication Union (ITU) [itu00] standard for data communication over analog phone lines. We used the V.34 service of Stanford University. CDMA (Code Division Multiple Access) is a digital cellular technology. We tested CDMA service by Sprint PCS [spr00] with AT&T Global Internet Services as the Internet service provider. These are most of the link technologies that are currently available for users.

In all cases the bottleneck link is the link closest to one of the hosts. This allows us to measure the best and worst cases for `nettimer` as described below. The short paths are representative of local area and metropolitan area networks while the long paths are representative of a cross-country, wide area network. We were not able to get access to an international tracing machine.

All the tracing hosts are Intel Pentiums ranging from 266MHz to 500MHz. The versions of software used are listed in Table 2.

We measure network accuracy by showing a lower bound (TCP throughput on a path with little cross traffic) and an upper bound (the nominal bandwidth specified by the manufacturer). TCP throughput by itself is insufficient because it does not include the bandwidth consumed by link level headers, IP headers, TCP headers and retransmissions. The nominal bandwidth is insufficient because the manufacturer usually measures under conditions that may be difficult to achieve in practice. Another possibility would be for us to measure each of the bottleneck link technologies on an isolated test bed. However, given the number and types of link technologies, this would have been difficult.

The network application workload consists of using `scp` (a secure file transfer program from openssh) to copy a 7476723 byte MP3 file once in each directions along a path. The transfer is terminated after five minutes even if the file has not been fully transferred.

We copy the file in both directions because 1) the ADSL technology is asymmetric and we want to measure both bandwidths and 2) we want to take measurements where the bottleneck link is the first link and the last link. A first link bottleneck link is the worst case for `nettimer` because it provides the most opportunity for cross traffic to interfere with the packet pair property. A last link bottleneck link is the best case for the opposite reason.

We copy a 7476723 byte file as a compromise between having enough samples to work with and not having so many samples that traces are cumbersome to work with. We terminate the tracing after five minutes so that we do not have to wait hours for the file to be transferred across the lower bandwidth links.

The network environment centers around the Stanford University campus but also includes the networks of Pacific Bell, Sprint PCS, Harvard University and the ISPs that connect Stanford and Harvard.

We ran five trials so that we could measure the effect of different levels of cross traffic during different times of day and different days of the week. The traces were started at 18:07 PST 12/01/2000 (Friday), 16:36 PST 12/02/2000 (Saturday), 11:07 PST 12/04/2000 (Monday), 18:39 PST 12/04/2000 (Monday), and 12:00 PST 12/05/2000 (Tuesday). We believe that these traces cover the peak traffic times of the networks that we tested on: commute time (Sprint PCS cellular), weekends and nights (Pacific Bell ADSL, Stanford V.34, Stanford residential network), work hours (Stanford and Harvard Computer Science Department networks).

Within the limits of our resources, we have selected as many different values for our experimental parameters as possible to capture some of the heterogeneity of the Internet.

4.2 Results

In this section, we analyze the results of the experiments.

4.2.1 Varied Bottleneck Link

One goal of this work is to determine whether `nettimer` can measure across a wide variety of network technologies. Dealing with different network technologies is not just a matter of dealing with different bandwidths because different technologies have very different link and physical layer protocols that could affect bandwidth measurement.

Using Table 3, we examine the short path Receiver Based Packet Pair results for the different technologies. This table gives the mean result over all the times and days of the TCP throughput and Receiver Based result reported by `nettimer`.

The Ethernet 100Mb/s case and to a lesser extent the Ethernet 10Mb/s case show that using TCP to measure the bandwidth of a high bandwidth link can be

Table 3: This table summarizes `nettimer` results over all the times and days. "Type" lists the different bottleneck technologies. "D" lists the direction of the transfer. "u" and "d" indicate that data is flowing away from or towards the bottleneck end, respectively. "Path" indicates whether the (l)ong or (s)hort path is used. "N" lists the nominal bandwidth of the technology. "TCP" lists the TCP throughput. "RB" lists the `nettimer` results for Receiver Based packet pair. (σ) lists the standard deviation over the different traces.

High bandwidth technologies (Mb/s):

Type	D	P	N	TCP (σ)	RB (σ)
Ethernet	d	s	100	21.22 (.13)	88.39 (.01)
Ethernet	d	l	100	2.09 (.41)	59.15 (.04)
Ethernet	u	s	100	19.92 (.05)	90.16 (.06)
Ethernet	u	l	100	1.51 (.58)	92.03 (.02)
Ethernet	d	s	10.0	6.56 (.06)	9.65 (.00)
Ethernet	d	l	10.0	1.85 (.14)	9.62 (.00)
Ethernet	u	s	10.0	7.80 (.03)	9.46 (.00)
Ethernet	u	l	10.0	1.66 (.21)	9.30 (.02)
WaveLAN	d	s	11.0	4.33 (.16)	6.52 (.20)
WaveLAN	d	l	11.0	1.63 (.13)	7.25 (.22)
WaveLAN	u	s	11.0	4.64 (.17)	5.30 (.12)
WaveLAN	u	l	11.0	1.51 (.32)	5.07 (.14)
WaveLAN	d	s	2.0	1.38 (.01)	1.48 (.02)
WaveLAN	d	l	2.0	1.05 (.09)	1.47 (.02)
WaveLAN	u	s	2.0	1.07 (.05)	1.21 (.01)
WaveLAN	u	l	2.0	0.87 (.26)	1.17 (.00)
ADSL	d	s	1.5	1.21 (.01)	1.24 (.00)
ADSL	d	l	1.5	1.16 (.01)	1.23 (.00)

Low bandwidth technologies (Kb/s):

Type	D	P	N	TCP (σ)	RB (σ)
ADSL	u	s	128	96.87 (.19)	109.28 (.00)
ADSL	u	l	128	107.0 (.01)	109.51 (.00)
V.34	d	s	33.6	26.43 (.04)	27.04 (.03)
V.34	d	l	33.6	26.77 (.04)	27.52 (.04)
V.34	u	s	33.6	27.98 (.01)	28.62 (.01)
V.34	u	l	33.6	28.05 (.00)	28.82 (.00)
CDMA	d	s	19.2	5.30 (.05)	10.88 (.05)
CDMA	d	l	19.2	5.15 (.09)	10.83 (.09)
CDMA	u	s	19.2	6.76 (.24)	18.48 (.05)
CDMA	u	l	19.2	6.50 (.53)	17.21 (.11)

inaccurate and/or expensive. For both Ethernets, the TCP throughput is significantly less than the nominal bandwidth. This could be caused by cross traffic, not being able to open the TCP window enough, bottlenecks in the disk, inefficiencies in the operating system, and/or the encryption used by the `scp` application. In general, using TCP to measure bandwidth requires actually filling that bandwidth. This may be expensive in resources and/or inaccurate. We have no explanation for the RBPP result of 59Mb/s for the down long path.

In the WaveLAN cases, both the `nettimer` estimate and the TCP throughput estimate deviate significantly from the nominal. However, another study [BPSK96] reports a peak TCP throughput over WaveLAN 2Mb/s of 1.39Mb/s. We took the traces with a distance of less than 3m between the wireless node and the base station and there were no other obvious sources of electromagnetic radiation nearby. We speculate that the 2Mb/s and 11Mb/s nominal rates were achieved in an optimal environment shielded from external radio interference and conclude that the `nettimer` reported rate is close to the actual rate achievable in practice.

Another anomaly is that the `nettimer` measured WaveLAN bandwidths are consistently higher in the down direction than in the up direction. This is unlikely to be `nettimer` calculation error because the TCP throughputs are similarly asymmetric. Since the hardware in the PCMCIA NICs used in the host and the base station are identical, this is most likely due to an asymmetry in the MAC-layer protocol.

The `nettimer` measured ADSL bandwidth consistently deviates from the nominal by 15%-17%. Since the TCP throughput is very close to the `nettimer` measured bandwidth, this deviation is most likely due to the overhead from PPP headers and byte-stuffing (Pacific Bell/SBC ADSL uses PPP over Ethernet) and the overhead of encapsulating PPP packets in ATM (Pacific Bell/SBC ADSL modems use ATM to communicate with their switch). Link layer overhead is also the likely cause of the deviation in V.34 results.

The CDMA results exhibit an asymmetry similar to the WaveLAN results. However, we are fairly certain that the base station hardware is different from our client transceiver and this may explain the difference. However, this may also be due to an interference source close to the client and hidden from the base station. In addition, since the TCP throughputs are far from both the nominal and the `nettimer` measured bandwidth, the deviation may be due to `nettimer` measurement error.

We conclude that `nettimer` was able to measure the bottleneck link bandwidth of the different link technologies with a maximum error of 41%, but in most cases with an error less than 10%.

4.2.2 Resistance to Cross Traffic

We would expect that the long paths would have more cross traffic than the short paths and therefore interfere with `nettimer`. In addition, we would expect that bandwidth in the up direction would be more difficult to measure than bandwidth in the down direction because packets have to travel the entire path before their arrival time can be measured.

However, Table 3 shows that the RBPP technique and nettimer's filtering algorithm are able to filter out the effect of cross traffic such that nettimer is accurate for long paths even in the up direction.

In contrast, ROPP is much less accurate on the up paths than on the down paths (Section 4.2.3).

It was pointed out by an anonymous reviewer that there may be environments (e.g. a busy web server) where packet sizes and arrival times are highly correlated, which would violate some of the assumptions described in Section 2.2.1. There are definitely parts of the Internet containing technologies and/or traffic patterns so different from those described here that they cause nettimer's filtering algorithm to fail. One example is multi-channel ISDN, which is no longer in common use in the United States. We simply claim that nettimer is accurate in a variety of common cases which justifies further investigation into its effectiveness in other cases.

4.2.3 Different Packet Pair Techniques

In this section, we examine the relative accuracy of the different packet pair techniques. Table 4 shows the Receiver-Only and Sender-Based results of one day's traces.

Sender Based Packet Pair is not particularly accurate, reporting 20%-50% of the estimated bandwidth, even on the short paths. As mentioned before, this is most likely the result of passively using TCP's non-per-packet acknowledgements and delayed acknowledgements. We discuss possible solutions to this in Section 5.

In the down direction for both long and short paths, Receiver Only Packet Pair is almost as accurate as RBPP. In contrast, Receiver Only Packet Pair is amazingly inaccurate in the up direction. For ROPP to make an accurate measurement, packets have to preserve their spacing resulting from the first link during their journey along all of the later links. ROPP cannot filter using the sent bandwidth (Section 2.2.2) because it does not have the cooperation of the sending host. Consequently, ROPP has poor accuracy compared to RBPP.

Table 4: This table shows 11:07 PST 12/04/2000 nettimer results."Type" lists the different bottleneck technologies. "D" lists the direction of the transfer. "u" and "d" indicate that data is flowing away from or towards the bottleneck end, respectively. "P" indicates whether the (l)ong or (s)hort path is used. "Nom" lists the nominal bandwidth of the technology. "RO" and "SB" list the Receiver Only or Sender Based packet pair bandwidths respectively. (σ) lists the standard deviation over the duration of the connection.

High bandwidth technologies (Mb/s):

Type	D	P	Nom	RO (σ)	SB (σ)
Ethernet	d	s	100.0	87.69 (.12)	29.22 (.46)
Ethernet	d	l	100.0	63.65 (.27)	22.56 (1.8)
Ethernet	u	s	100.0	697.39 (.12)	52.28 (.22)
Ethernet	u	l	100.0	706.34 (.04)	13.96 (1.1)
Ethernet	d	s	10.0	9.65 (.03)	92.80 (.49)
Ethernet	d	l	10.0	9.65 (.04)	12.44 (2.5)
Ethernet	u	s	10.0	84.03 (.47)	4.63 (.04)
Ethernet	u	l	10.0	97.85 (.04)	6.42 (2.6)
WaveLAN	d	s	11.0	8.00 (.22)	3.40 (3.1)
WaveLAN	d	l	11.0	8.36 (.25)	2.11 (.33)
WaveLAN	u	s	11.0	11.30 (.03)	2.43 (.29)
WaveLAN	u	l	11.0	11.56 (.03)	1.77 (.31)
WaveLAN	d	s	2.0	1.46 (.03)	0.76 (.03)
WaveLAN	d	l	2.0	1.46 (.04)	0.74 (.05)
WaveLAN	u	s	2.0	1.20 (.03)	0.60 (.00)
WaveLAN	u	l	2.0	1.20 (.03)	0.59 (.06)
ADSL	d	s	1.5	1.24 (.03)	0.59 (.04)
ADSL	d	l	1.5	1.24 (.04)	0.59 (.05)

Low bandwidth technologies (Kb/s):

Type	D	P	Nom	RO (σ)	SB (σ)
ADSL	u	s	128.0	465.34 (.04)	54.53 (.01)
ADSL	u	l	128.0	390.58 (.07)	53.89 (.04)
V.34	d	s	33.6	26.43 (.04)	6.94 (.95)
V.34	d	l	33.6	28.54 (.07)	5.35 (1.3)
V.34	u	s	33.6	831.67 (3.6)	14.45 (.05)
V.34	u	l	33.6	674.15 (2.7)	14.50 (.03)
CDMA	d	s	19.2	11.40 (.17)	9.85 (.36)
CDMA	d	l	19.2	12.07 (.09)	12.45 (.36)
CDMA	u	s	19.2	508.12 (1.5)	11.08 (.26)
CDMA	u	l	19.2	484.07 (1.2)	7.56 (2.0)

4.2.4 Agility

In this section, we examine how quickly nettimer calculates bandwidth when a connection starts. Figure 5 shows the bandwidth that nettimer using RBPP reports at the beginning of a connection. The connection begins 1.88 seconds before the first point on the graph. nettimer initially reports a low bandwidth, then a (correct) high bandwidth, then a low bandwidth, then converges at the high bandwidth. The total time from the beginning of the connection to convergence is 3.72 seconds. It takes this long because scp requires several round trips to authenticate and negotiate the encryption.

If we measure from when the data packets begin to flow, nettimer converges when the 8th data packet arrives, 8.4 ms after the first data packet arrives, 10308 bytes into the connection. TCP would have reported the throughput at this point as 22.2Kb/s. Converging within 10308 bytes means that an adaptive web server could measure bandwidth using just the text portion of most web pages and then adapt its images based on that measurement.

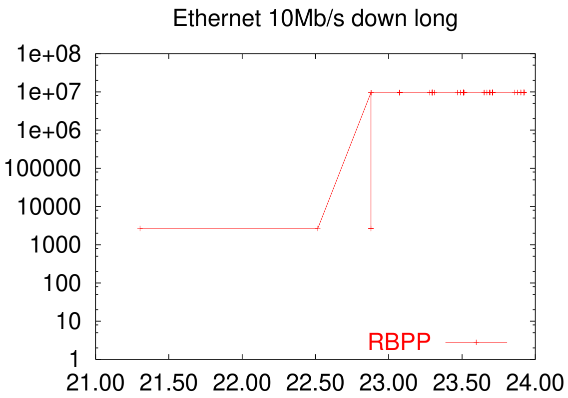

Figure 5: This graph shows the bandwidth reported by `nettimer` using RBPP at a particular time for Ethernet 10Mb/s in the down direction along the long path. The Y-axis shows the bandwidth in b/s on a log scale. The X-axis shows the number of seconds since tracing began.

Table 5: This table shows the CPU overhead consumed by nettimer and the application it is measuring. "User" lists the user-level CPU seconds consumed. "System" lists the system CPU seconds consumed. "Elapsed" lists the elapsed time that the program was running. "% CPU" lists (User + System) / `scp` Elapsed time.

Name	User	System	Elapsed	% CPU
server	.31	.43	32.47	4.52%
client	9.28	.15	26.00	57.6%
`scp`	.050	.21	16.37	1.59%

4.2.5 Resources Consumed

In this section, we quantify the resources consumed by `nettimer`. In contrast to the other experiments where we took traces and then used `nettimer` to process the traces, in this experiment, `nettimer` captured its own packets and calculated the bandwidth as the connection was in progress. We measure the Ethernet 100Mb/s short up path because this requires the most efficient processing. We use `scp` and copy the same file as before. The distributed packet capture server ran on an otherwise unloaded 366MHz Pentium II while the packet capture client and `nettimer` processing ran on an otherwise unloaded 266MHz Pentium II.

Table 5 lists the CPU resources consumed by each of the components. The CPU cycles consumed by the distributed packet capture server are negligible, even for a 366MHz processor on a 100Mb/s link. `Nettimer` itself does consume a substantial number of CPU seconds to classify packets into flows and run the filtering algorithm. However, this was on a relatively old 266MHz machine and this functionality does not need to be collocated with the machine providing the actual service being measured (in this case the `scp` program).

Transferring the packet headers from the `libdpcap` server to the client consumed 473926 bytes. Given that the file transferred is 7476723 bytes, the overhead is 6.34%. This is higher than the 5.00% predicted in Section 3.3.2 because 1) `scp` transfers some extra data for connection setup, 2) some data packets are retransmitted, and most significantly, 3) the `libdpcap` server captures its own traffic. The server captures its own traffic because it does not distinguish between the `scp` data packets and its own packet header traffic, so it captures the headers of packets containing the headers of packets containing headers and so on. Fortunately, there is a limit to the recursion so the net overhead is close to the predicted overhead.

5 Future Work

In this section, we describe some possible future improvements to the `nettimer` implementation. One improvement would be to determine what the optimal weighting of components in the filtering algorithm (Section 2.2.3) is. Another improvement would be to allow runtime choice of flow definition (Section 3.1), so that we could measure bandwidth behind NAT gateways. Another improvement would be to allow distributed packet capture servers to randomly sample traffic to reduce the amount of bandwidth that packet reports consume. Finally, we could add an active probing component like [Sav99] which can cause large packets to flow in both directions from hosts without special measurement software and can cause prompt acknowledgements to flow back to the sender. The `pathrate` [DRM01] tool shows that active packet pair can be very accurate.

6 Conclusion

In this paper, we describe the trade-offs involved in implementing `nettimer`, a Packet Pair-based tool for passively measuring bottleneck link bandwidths in real time in the Internet. We show its utility across a wide variety of bottleneck link technologies ranging from 19.2Kb/s to 100Mb/s, wired and wireless, symmetric and asymmetric bandwidth, across local area and cross country paths, while using both one and two packet capture hosts.

In the future, we hope that `nettimer` will ease the creation of adaptive applications, provide more insight for network performance analysis, and lead to the development of more precise performance measurement algorithms.

7 Acknowledgments

We would like to thank several people without whom this research would have been extremely difficult. The anonymous USITS reviewers provided many valuable comments. Margo Seltzer and Dave Sullivan at Harvard University graciously allowed us to do the critical cross-country tracing at one of their machines. Brian Roberts gave us the last available 100Mb/s Ethernet switch port on our floor. Ed Swierk hosted one of our laptops in his campus apartment. T.J. Giuli lent us his laptop for several weeks.

This work was supported by a gift from NTT Mobile Communications Network, Inc. (NTT DoCoMo). In addition, Kevin Lai was supported in part by a USENIX Scholar Fellowship.

References

[Bol93] Jean-Chrysostome Bolot. End-to-End Packet Delay and Loss Behavior in the Internet. In *Proceedings of ACM SIGCOMM*, 1993.

[BPSK96] Hari Balakrishnan, Venkata Padmanabhan, Srinivasan Seshan, and Randy Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *Proceedings of ACM SIGCOMM*, 1996.

[CC96a] Robert L. Carter and Mark E. Crovella. Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks. Technical Report BU-CS-96-007, Boston University, 1996.

[CC96b] Robert L. Carter and Mark E. Crovella. Measuring Bottleneck Link Speed in Packet-Switched Networks. Technical Report BU-CS-96-006, Boston University, 1996.

[Dow99] Allen B. Downey. Using pathchar to Estimate Internet Link Characteristics. In *Proceedings of ACM SIGCOMM*, 1999.

[DRM01] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. What do packet dispersion techniques measure? In *Proceedings of IEEE INFOCOM*, April 2001.

[dsl00] DSL. http://www.pacbell.com/DSL, 2000.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[itu00] ITU. http://www.itu.int, 2000.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM*, 1988.

[Jac97] Van Jacobson. pathchar. ftp://ftp.ee.lbl.gov/pathchar/, 1997.

[Kes91] Srinivasan Keshav. A Control-Theoretic Approach to Flow Control. In *Proceedings of ACM SIGCOMM*, 1991.

[LB99] Kevin Lai and Mary Baker. Measuring Bandwidth. In *Proceedings of IEEE INFOCOM*, March 1999.

[LB00] Kevin Lai and Mary Baker. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *Proceedings of ACM SIGCOMM*, August 2000.

[Mah00] Bruce A. Mah. pchar. http://www.ca.sandia.gov/~bmah/Software/pchar/, 2000.

[MJ93] Steve McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, 1993.

[MJ98] G. Robert Malan and Farnam Jahanian. An Extensible Probe Architecture for Network Protocol Performance Measurement. In *Proceedings of ACM SIGCOMM*, 1998.

[MM96] M. Mathis and J. Mahdavi. Diagnosing Internet Congestion with a Transport Layer Performance Tool. In *Proceedings of INET*, 1996.

[Pax97] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamics.* PhD thesis, University of California, Berkeley, April 1997.

[Sav99] Stefan Savage. Sting: a TCP-based Network Measurement Tool. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1999.

[Sco92] Dave Scott. *Multivariate Density Estimation: Theory, Practice and Visualization.* Addison Wesley, 1992.

[spr00] Sprint PCS. http://www.sprintpcs.com/, 2000.

[Ste99] Mark R. Stemm. *An Network Measurement Architecture for Adaptive Applications.* PhD thesis, University of California, Berkeley, 1999.

[wav00] WaveLAN. http://www.wavelan.com/, 2000.