# Measuring Bandwidth

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

*Abstract*— **Accurate network bandwidth measurement is important to a variety of network applications. Unfortunately, accurate bandwidth measurement is difficult. We describe some current bandwidth measurement techniques: using throughput, `pathchar` [8], and Packet Pair [2]. We explain some of the problems with these techniques, including poor accuracy, poor scalability, lack of statistical robustness, poor agility in adapting to bandwidth changes, lack of flexibility in deployment, and inaccuracy when used on a variety of traffic types. Our solutions to these problems include using a packet window to adapt quickly to bandwidth changes, Receiver Only Packet Pair to combine accuracy and ease of deployment, and Potential Bandwidth Filtering to increase accuracy. Our techniques are are at least as accurate as previously used filtering algorithms, and in some situations, our techniques are more than 37% more accurate.**

## I. Introduction

A common complaint about the Internet is that it is slow. Some of this slowness is due to properties of the end points, like slow servers, but some is due to properties of the network, like propagation delay and limited bandwidth. Propagation delay can be measured using widely deployed and well understood algorithms implemented in tools like `ping` and `traceroute`. Unfortunately, tools to measure bandwidth are neither widely deployed nor well understood. This work attempts to develop further understanding of how to measure bandwidth.

Current bandwidth measurement techniques have many problems: poor accuracy, poor scalability, lack of statistical robustness, poor agility in adapting to bandwidth changes, lack of flexibility in deployment, and inaccuracy when used on a variety of traffic types. We propose solutions to these problems and demonstrate their effectiveness:

**Packet Window** We use a packet window (not the TCP window) to adapt quickly to bandwidth changes. In the presence of link failure, a small window is 144% more accurate than an infinite window.

**Receiver Only Packet Pair** We use Receiver Only Packet Pair to allow the deployment of special software at only one host while achieving accuracy within 1% of Receiver Based Packet Pair [12].

**Potential Bandwidth Filtering** We use Potential Bandwidth Filtering to measure bandwidth accurately in the presence of a variety of packet sizes. In such an environment, it is at least 37% more accurate than previously used filtering algorithms [5] [12].

This research is supported by a gift from NTT Mobile Communications Network, Inc. (NTT DoCoMo), a graduate fellowship from the USENIX Association, and a Sloan Foundation Faculty Fellowship.

Our overall goal is to make Packet Pair algorithms practical and robust enough to be widely and frequently used. Our approach has been to derive simple algorithms from statistically valid network models and avoid heuristics. Heuristics, especially in combination, tend to be difficult to debug and explain and lack the robustness to apply to diverse network environments.

The rest of the paper is organized as follows: In Section II, we present motivation for examining bandwidth measurement techniques. In Section III, we propose ways to make Packet Pair algorithms robust and practical. In Section IV, we describe bottleneck bandwidth algorithms. In Section V, we describe our new Packet Pair filtering algorithm, Potential Bandwidth Filtering. In Section VI, we describe how we simulated different bottleneck bandwidth algorithms on a variety of networks. In Section VII, we present the results of our simulations. In Section VIII, we describe our plans for further exploration of this area. Finally, we conclude in Section IX with our overall observations about the algorithms.

## II. Motivation

In this section, we describe the motivation for examining bandwidth measurement techniques.

### A. Applications

Several applications could benefit from knowing the bottleneck bandwidth of a route. Developers of network protocols and applications need to know the bottleneck bandwidth to judge the efficiency of their protocols and applications. For example, if an HTTP server is delivering data at close to the bottleneck bandwidth, then increasing the bandwidth of that link may increase application performance. However, if the bottleneck link already has plenty of bandwidth to spare, increasing its bandwidth will probably not improve application performance.

Network clients could dynamically choose the "best" server for an operation based on the highest bottleneck bandwidth. This has been suggested as a way to choose a web server or proxy [4] [15].

In addition, accurate and timely bandwidth measurement is useful for mobile computing. Mobile computers frequently have more than one network interface, often with very different bandwidths (e.g. 10Mb/s Ethernet and 28 Kb/s wireless). Knowing the bandwidth would allow the mobile host to pick the highest bandwidth interface as the default in-

terface and to degrade service gracefully when it detects that it is operating on a low bandwidth link.

Another application is congestion control. TCP already implicitly measures the bandwidth of the network so that it will not send packets faster than the network can handle, but this has certain disadvantages described in the next section.

Finally, we could use bandwidth information to build multicast routing trees more efficiently and dynamically. Ideally, multicast routing trees would be built so that packets travel along a tree that minimizes duplicate packets and latency while maximizing bandwidth. Currently, multicast routing trees are built either without bandwidth information or with only static information.

### *B. Metrics*

We distinguish between the *bottleneck* bandwidth and the *available* bandwidth of a route. The bottleneck bandwidth of a route is the ideal bandwidth of the lowest bandwidth link (the *bottleneck link*) on that route between two hosts. In most networks, as long as the route between the two hosts remains the same, the bottleneck bandwidth remains the same. The bottleneck bandwidth is not affected by other traffic. In contrast, the available bandwidth of a route is the maximum bandwidth at which a host can transmit at a given point in time along that route. Available bandwidth is limited by other traffic along that route.

The question of which is the better metric can only be answered by the application. Some applications want to know which route will give them the minimum delay or want to use an estimate taken longer than a few seconds ago. For these applications bottleneck bandwidth is probably the best metric. Some applications are only interested in the best average throughput. For these applications, available bandwidth is probably the best metric.

We are interested in both metrics, but have chosen to investigate bottleneck bandwidth first because it is a more stable metric and is therefore useful over a longer period of time, and because it bounds the available bandwidth and can therefore be used later to more accurately compute available bandwidth (see section VIII).

### *C. Current Techniques*

Given the importance of measuring bandwidth, it is not surprising that there are currently several techniques for doing so. However, all have drawbacks for at least some of the applications described above.

The most popular technique is to use throughput as an approximation of bandwidth. Throughput is the amount of data a transport protocol like TCP can transfer per unit of time. One problem with throughput is that other metrics (e.g. packet drop rate) may have a significant effect on TCP throughput, while not affecting bandwidth.

Another problem with measuring throughput is that an application's throughput to a host implies nothing about other transfers, even from the same application to the same host. For example, a web browser sending a request to a web server may experience low throughput because that request involved running a slow CGI script. The same browser sending a different request could experience high throughput because the latter request did not involve running a CGI script. Correlating the throughput of different applications (like telnet and http) is even more inaccurate.

TCP uses another technique to estimate bandwidth. It sends more and more packets until one is dropped. It estimates the bandwidth to be somewhere between the sending rate when the packet was dropped and half that rate. This has several problems: 1) TCP is measuring the bottleneck router's buffer size in addition to the bottleneck bandwidth, 2) TCP wastes network resources by forcing a dropped packet and filling the router's buffers, and 3) TCP has to increase its sending rate slowly, or else it will overshoot the real bandwidth and cause massive packet loss. The last problem is particularly acute on high bandwidth, high latency links, such as satellite connections, because TCP needs $O(log(bandwidth) \cdot latency)$ time to reach the maximum transmission rate.

Another solution is to use `pathchar` [8]. As far as we know, `pathchar` is unique in its ability to measure the bandwidth of every link on a path accurately while requiring special software on only one host. This means it could easily be widely deployed. Although excellent as a testing tool, the problem with `pathchar` is that it is slow and can consume significant amounts of network bandwidth. In particular, `pathchar` runs in time proportional to the round trip time of the network and sends a relatively fixed amount of data, regardless of the actual bandwidth of the network (see Section IV-A). If more hosts were to run `pathchar`, its packets would become a significant burden on the network [15]. Even for isolated hosts with low bandwidth connections, `pathchar` could consume too much bandwidth to be usable regularly.

## III. Making Packet Pair Robust and Practical

The bandwidth measurement technique we have chosen to investigate is called Packet Pair [2] (described in more detail in Section IV-B). The advantages it has over the techniques mentioned above are that 1) it measures the true bandwidth of the network (instead of throughput), 2) it does not cause packet loss (unlike TCP), 3) it does not require many packet round trips to work, and 4) it does not send massive amounts of data (unlike `pathchar`). On the other hand, each of those techniques currently has a robust and practical implementation and is in common use in the Internet, while Packet Pair does not have such an implementation and is rarely used at all (although several tools implement a version of the Packet Pair algorithm, including `bprobe` [5] and `tcpanaly` [11]).

Here are some of the key problems with current Packet

Pair algorithms and how we propose to solve them:

**Not statistically robust** Previous work on filtering Packet Pair samples has used techniques such as adding error bars to values and intersecting them [5], or clustering values that are close together [12]. Such heuristics do not have well understood statistical properties and their effectiveness may depend on a particular data set.
Instead, we use a *kernel density estimator* to filter data, which has well understood statistical properties [14]. In particular, it makes no assumptions about the data, and is therefore statistically robust (see Section IV-C).
**Not scalable** *Active* Packet Pair implementations [5] generate their own traffic and therefore have the same scalability problems as `pathchar` (see Section IV-A).
Packet Pair algorithms do not need to do this [12]. Our *passive* Packet Pair implementation uses existing network traffic to measure bandwidth.
**Slow** On the other hand, previous *passive* implementations are designed to analyze a TCP connection after its completion [12], instead of while it is occurring. Given the long duration of some TCP connections, this could be too late for some of the applications mentioned above.
Our *gradual* Packet Pair implementation forms a bandwidth estimate for every packet that arrives. It initially gives inaccurate answers and then gradually converges to an accurate answer. In this way, applications obtain an estimate as soon as it is available (see Section IV-E). Our results show that Packet Pair can give the correct estimate within three packets of the start of the connection.
**Not robust on all traffic** Most passive implementations are designed to use traffic composed of mostly large packets. However, Internet traffic is a mix of many packet sizes and any one flow between two hosts may contain a wide variety of packet sizes. Existing passive implementations do not account for this and thus give inaccurate results on diverse traffic such as a mix of predominantly small packets and a few large packets (see Section V). This is because the network model used in those implementations does not account for *potential* bandwidth.
We designed a new Packet Pair filtering algorithm, called Potential Bandwidth Filtering (PBF), which can deliver an accurate answer despite variation in packet size and transmission rate. On a mix of packet sizes, PBF is at least 37% more accurate than the standard filtering algorithm.
**Not flexible to bandwidth changes** Some prior implementations detect only one bandwidth over time [5]. Other implementations can detect multiple bandwidths, but only those which differ by a large amount [12]. Although not as frequent as congestion changes, bottleneck bandwidth changes do happen because of routing changes [12] or because of mobility [1]. Some of the applications described above would like to know as soon as possible that the bandwidth has changed and what the new bandwidth is, regardless of magnitude.
To accomplish the above, we propose the use of a limited *window* of past packets to calculate bandwidth. This increases the speed at which the algorithm can adapt to a new bandwidth (the *agility*), but it leaves the results more vulnerable to noise. We believe that an increase in agility fundamentally requires becoming more vulnerable to noise (see Section IV-E). We show that a small window is 144% more accurate than an infinite window in the presence of link failure, but 10% less accurate in the presense of congestion.
**Difficult to deploy** A current highly accurate Packet Pair algorithm, Receiver Based Packet Pair (RBPP) [12], requires that packet timings be taken at both the sender and receiver of those packets. This means that special software must be deployed at both the sender and receiver, which may not be possible. Another algorithm, Sender Based Packet Pair (SBPP) [12] requires timings (and therefore special software) only on the sender. Unfortunately, SBPP is far less accurate than RBPP.
We describe a variation of RBPP, called Receiver Only Packet Pair (ROPP), which is more accurate than SBPP (but less than RBPP), while only requiring timings at the receiver (see Section IV-D). This allows applications to trade some accuracy for ease of deployment. Our results show that ROPP is accurate within 1% of RBPP.
**Not studied under controlled conditions** There have been several studies of Packet Pair algorithms using data from the Internet. This has the advantage of using real TCP/IP code, routers, and network traffic. However, we would like: 1) verifiable and reproducible results and 2) testing under a variety of controlled conditions. Testing under controlled conditions in the Internet at large would be difficult, if not impossible.
To overcome these limitations, we use a network simulator (fully described in Section VI) to compare the effectiveness of the algorithms and modifications described above to previous Packet Pair implementations.

## IV. Bandwidth Measurement Algorithms

In this section, we describe the models and assumptions of the algorithms for measuring bottleneck bandwidth. We consider their accuracy, timeliness, and agility as well as whether they are active or passive and whether they require measurements from multiple network hosts.

We know of two families of bottleneck bandwidth algorithms. The first family of algorithms, which we call the Pathchar Algorithms, is used in the tools `pathchar` [8] and `utimer` [6]. The second family of algorithms is based on the Packet Pair algorithm and is used in the tools `bprobe` [5], `cprobe` [5], and `tcpanaly` [11]. Variants of the Packet Pair algorithm are Sender Based Packet Pair (SBPP), Receiver Based Packet Pair (RBPP), Packet Bunch Mode (PBM) [12], and our own Receiver Only Packet Pair (ROPP).

An orthogonal issue for Packet Pair algorithms is how they filter bandwidth samples. We call the standard algorithms Measured Bandwidth Filtering (MBF) and propose our own

Potential Bandwidth Filtering (PBF). In addition, we describe our refinements of the Packet Pair algorithms and their filtering methods: the use of a kernel density function to increase statistical robustness, the use of gradual bandwidth calculation to increase timeliness, and the use of a packet window to increase agility.

### A. Pathchar Algorithm

In this section, we analyze the time taken and bandwidth consumed by the Pathchar algorithm. The program works by sending packets of varying sizes and measuring their round trip time. It correlates the round trip times with the packet sizes to calculate bandwidth. It uses the results from earlier hops for calculations on father hops. For a more thorough description of how and why `pathchar` works, see [8]. For our purposes, all we need to know is that the `pathchar` program uses an active algorithm that sends packets varying in size from 64 bytes to the path MTU with a stride of 32 bytes. Therefore, the number of different packet sizes `pathchar` sends is

$$s = \left\lfloor \frac{MTU}{32} \right\rfloor - 1 \tag{1}$$

For Ethernet, the MTU is 1500 bytes, so $s$ is 45. In addition, it sends $p$ packets per size for every hop. In the default configuration, $p = 32$. It must wait for each packet it sends to be acknowledged before sending the next packet. Thus, the total time for `pathchar` to run is

$$\sum_{i=1}^{h} p \cdot s \cdot l_i \tag{2}$$

where $h$ is the number of hops and $l_i$ is the round trip latency from the sender to hop $i$. We assume that the receiver immediately sends an ack in response to a packet and that the sender immediately sends out the next packet when an ack arrives. For a 10-hop Ethernet network with an average round trip latency of 10ms, `pathchar` would run in 144 seconds. This is too slow for a host to run it for every TCP connection, or even every 10 minutes. It can be configured to send fewer packets of each size, but at the cost of accuracy.

More importantly, `pathchar` consumes considerable amounts of network bandwidth. The average bandwidth used for probing a particular hop is

$$\frac{average\ packet\ size}{round\ trip\ latency} = \frac{\frac{32 \cdot s}{2} + 32}{l_i} \tag{3}$$

in bytes/s, where $l_i$ is the round trip latency (in seconds) across that hop. For a 1-hop Ethernet network with a latency of 1ms, the average bandwidth consumed is 6.02Mb/s. This would be a considerable imposition on a 10Mb/s Ethernet. Farther hops would consume less bandwidth, but `pathchar` always has to probe closer hops before farther hops. Furthermore, the total data transferred is

$$(p)(h)\left(\sum_{i=2}^{s} 32i\right) \tag{4}$$

where $h$ is the number of hops. For the 10-hop Ethernet network mentioned before, `pathchar` sends 10 MB of data. In fact, `pathchar` will send 10 MB of data on a 10-hop network regardless of the bandwidth of the network, since it only depends on the number of hops, the path MTU, and $p$. If the path MTU is high and one of the early hops is a low bandwidth network link, such as a 56K modem, then `pathchar` can consume most of the bandwidth of that link for an extended amount of time. This means that we would have problems scaling `pathchar` usage up to a large number of hosts.

### B. Packet Pair

The basic Packet Pair algorithm [9] relies on the fact that if two packets are queued next to each other at the bottleneck link, then they will exit the link $t$ seconds apart:

$$t = \frac{s_2}{b_{bnl}} \tag{5}$$

where $s_2$ is the size of the second packet and $b_{bnl}$ is the bottleneck bandwidth. This is the "bottleneck separation" (see Figure 1). Since there are no links with lower bandwidth than the bottleneck link downstream of that link, and assuming the packets are the same size, the second packet will never catch up to the first packet.

The two packets have to be the same size because different size packets have different "velocities". If the second packet were smaller than the first, then its transmission delay would always be less than the first packet's. Consequently, it would pass through links faster than the first packet and quickly eliminate the bottleneck separation. Similarly, if the the first packet is smaller, then it will be faster than the second packet and continuously grow the bottleneck separation.

Assuming the bottleneck separation is constant, the two packets will arrive at the receiver spaced $t$ seconds apart. Since we know $s_2$, we can then calculate the bottleneck bandwidth:

$$b_{bnl} = \frac{s_2}{t} \tag{6}$$

This algorithm makes several assumptions that may not hold in practice. For instance, it is impossible to guarantee that two packets will queue next to each other at the bottleneck link. If other packets queue in between the two measurement packets, then (6) becomes

$$b_{bnl} = \frac{s_2 + s_o}{t} \tag{7}$$

where $s_o$ is the total size of the other packets. In addition, if other packets queue ahead of the first packet when it is downstream of the bottleneck link, those extra packets will delay

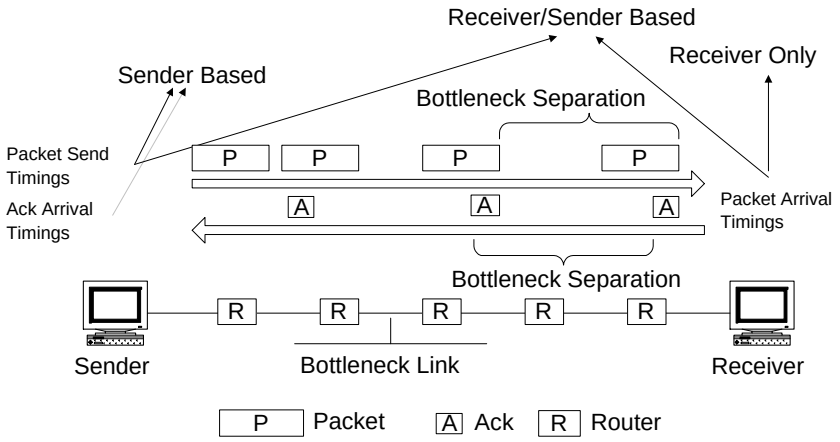


Fig. 1. This figure shows how the Packet Pair algorithm works. Note how the data packets have a greater separation after the bottleneck link and how this separation is maintained by the acks. The arrows pointing to SBPP, RBPP, and ROPP indicate what timing information must be sent from the sender and receiver for each of the algorithms.

the first packet, causing *time compression* of the two packets. Similarly, other packets could only delay the second packet, causing the packets to be *time extended*. Time compression can cause a high estimate of the bottleneck bandwidth, while time extension can cause a low estimate.

### C. Packet Pair Filtering

The main problem with the basic Packet Pair algorithm is how to filter out the noise caused by time compressed and extended packets. One solution would be to take the mean or median of all the bandwidth samples. Unfortunately, the noise has little correlation to the true bandwidth, so this gives wildly varying estimates.

Previous Packet Pair research has proposed finding the point of greatest density in the distribution of bandwidth estimates. The idea is that valid samples should be closely clustered around the correct value, while incorrect samples should not be clustered around any one value.

A well known method for doing this is to use a histogram. Unfortunately, there are several problems with histograms. One problem is that bin widths are fixed, and it is difficult to choose an appropriate bin width without previously knowing something about the distribution. Another problem is that bin boundaries do not respect the distribution. Two points could lie very close to each other on either side of a bin boundary and the bin boundary ignores that relationship. Finally, a histogram will report the same density for points with the same value as points which are in the same bin, but at opposite ends of the bin.

Previous Packet Pair filtering algorithms [5] [11] have overcome some of these problems, but not all of them. We use the kernel density estimator algorithm, which overcomes all of these problems. This algorithm is well known to statisticians [14] [16]. To use it, we first define a kernel function $K(t)$ with the property

$$\int_{-\infty}^{+\infty} K(t)dt = 1 \tag{8}$$

Then the density at any point $x$ is

$$\frac{1}{n}\sum_{i=1}^{n}\frac{1}{h}K\left(\frac{x-x_i}{h}\right) \tag{9}$$

where $h$ is the kernel width, $n$ is the number of points within $h$ of $x$, and $x_i$ is the $i$th such point. The kernel function we use is

$$y = \left\{ \begin{array}{ll} 1+x & x \leq 0 \\ 1-x & x > 0 \end{array} \right\} \tag{10}$$

This function has the desirable properties that it gives greater weight to samples closer to the point at which we want to estimate density, and it is simple and fast to compute. The kernel density estimator algorithm is known to be statistically valid and we show in section VII that it gives accurate results. Most importantly, it makes no assumptions about the distribution it operates on and therefore should be just as accurate on other data sets.

### D. Receiver and Sender Based Packet Pair

Receiver Based Packet Pair (RBPP) and Sender Based Packet Pair (SBPP) (both [12]) are types of Packet Pair algorithms. They differ in how the $t$ from (6) is measured. Figure 1 shows the difference in where timing measurements must be taken. In Receiver Based Packet Pair, $t$ is measured at the receiver, so (6) becomes

$$b_{bnl} = \frac{s_2}{a_2 - a_1} \tag{11}$$

where $a_1$ and $a_2$ are the arrival times of the first and second packets, respectively.

If we cannot measure the arrival times at the receiver, we have to use the round trip time, which is measured at the sender (SBPP). Equation (6) becomes

$$b_{bnl} = \frac{s_2}{r_2 - r_1} \tag{12}$$

where $r_1$ and $r_2$ are the arrival times of the acks to the first and second packets, respectively. This assumes that the receiver promptly sends back an acknowledgement for both of the packets. With SBPP, packets from other hosts could interfere with the acks as well as the original packets.

In both the receiver and sender based algorithms, we can apply additional filtering techniques to reject incorrect estimates. We can detect time compression or reordering when two packets have a difference between their transmission times greater than the difference between their arrival times (for RBPP) or their round trip times (for SBPP).

RBPP and SBPP are useful in different circumstances. RBPP is more accurate, but it can be harder to deploy since it requires measurement collection at both endpoints. SBPP is easy to deploy, but its results can be highly inaccurate during congestion (see Section VII).

Another difference is that SBPP requires that packets be acknowledged (as in TCP) and that the acks be constant size and relatively small. The acks must be constant size because variation in ack size causes variation in total round trip time, which would causes noise in the bandwidth samples. The acks must be small because as they become larger, the bandwidth of the path back to the sender would start to become the bottleneck. If the bottleneck bandwidth of the path back to the sender is much less than that from the sender (as in an asymmetric network) then ack size becomes that much more important (we see this effect in Section VII).

Finally, the algorithms differ in the kind of traffic they can use and the paths they can measure. SBPP relies on data packets flowing away from the measurement host and can only measure the bandwidth of the path from the sender to the receiver. RBPP can use whatever traffic is available. In the usual situation of data packets flowing in one direction and acks flowing in the other, RBPP can determine the bandwidth in both directions. However, the usually small size of the acks will limit the bandwidth that can be measured (see Section V).

Some applications may need the high accuracy of RBPP and the ease of deployment of SBPP. For those applications, we propose Receiver Only Packet Pair (ROPP). As shown in Figure 1, ROPP only takes timing measurements from the receiver and is therefore easier to deploy than RBPP. However, without timing information from the sender, ROPP cannot filter out time compressed packets or reordered packets, as SBPP and RBPP can. On the other hand, it is much less likely than SBPP to have such samples (like RBPP) because it is not relying on round trip latency. Another limitation is that it cannot use the new Potential Bandwidth Filtering algorithm described in Section V. Finallym, it has the limitation that it needs packets (although these can be acks) flowing on paths towards the measurement host and can only determine the bandwidth of such paths.

Despite these limitations, our results show that ROPP achieves accuracy within 1% of RBPP (see VII). We conclude that ROPP achieves the ease of deployment of SBPP, while sacrificing little accuracy. It is an excellent choice for applications needing to know the bandwidth of paths towards a host.

### *E. Timeliness versus Accuracy*

In this section we describe the tradeoff of accuracy versus timeliness in Packet Pair algorithms and how we implemented our algorithms to take advantage of these tradeoffs.

The Packet Pair algorithms described in the previous sections are usually implemented as running over a fixed number of packets or over an entire connection before providing an estimate. This translates into a long delay before providing an estimate. One problem is that some applications would prefer to have a ballpark answer sooner in addition to an accurate answer later.

Our solution is to calculate bandwidth *gradually*. Instead of calculating a single bandwidth, we calculate a new estimate with every packet arrival. In Section VII-B, we show that a gradual algorithm can converge to the correct bandwidth within three packets, instead of having to wait the entire life of the connection.

A problem with the gradual Packet Pair algorithm is that it is slow to detect a bandwidth change, i.e. it has poor agility. A bandwidth change may be caused by a route change such as a link failing or host mobility. The gradual algorithm described above will initially detect a bandwidth change as noise and stick to its initial estimate.

To compensate for this problem and to be able to detect multiple bandwidth changes, we use a packet window. We use at most $w$ (the window size) packets into the past to calculate the bandwidth at a particular packet. This has the advantage that only the most recent and probably most relevant samples are used to calculate bandwidth.

The disadvantage of using a window is that it may reduce stability. With smaller windows, we are more affected by transient conditions like congestion, which we may detect as a temporary bandwidth change, as shown in Section VII-B. We believe this is a fundamental tradeoff. A Packet Pair algorithm cannot distinguish between true changes in bandwidth and persistent congestion. However, given this fundamental limitation, our addition of windows to the basic Packet Pair algorithm enables it to distinguish bandwidth changes in the presence of light to moderate congestion.

## V. Potential Bandwidth Filtering

In this section we describe a previously unaddressed problem with using the filtering algorithm described above in

Section IV-C. We call that algorithm Measured Bandwidth Filtering (MBF) to distinguish it from our solution to the problem, Potential Bandwidth Filtering (PBF).

### *A. The Potential Bandwidth Problem*

One problem with Packet Pair algorithms is that they cannot measure a higher bandwidth than the bandwidth at which the sender sends. If a sender sends two packets of 1000 bytes each with 1ms separation, then the receiver cannot measure a higher bottleneck bandwidth than 8Mb/s, even if the true bottleneck bandwidth is 100Mb/s. This is a fundamental property of all Packet Pair algorithms regardless of how the filtering is done. We call the bandwidth at which the sender sends two packets the *potential* bandwidth because the measured bandwidth cannot exceed it.

The problem arises when the sender sends small packets, or sends packets slowly, or both. Then the potential bandwidth is likely to be lower than the actual bottleneck bandwidth of the path, and any measured bandwidth will be wrong.

Fortunately, some packets have a large potential bandwidth. Most HTTP and FTP packets are large and rapidly sent, and therefore have a high potential bandwidth. Unfortunately, it may be that not all packets in a flow have a high potential bandwidth, and in fact, it may frequently be the case that the high potential bandwidth packets are not the most common type of packets.

For example, consider someone browsing a site using HTTP/1.1. HTTP/1.1 opens one TCP connection to a site and uses that connection for all communication. The client will receive many large packets filled with HTML pages while sending many acks and a few medium-sized packets filled with HTTP requests. The outbound link will be dominated by many small packets, with a few medium-sized packets. If we used the normal MBF algorithm, we would report the measured bandwidth of the small packets.

We discovered this problem in our simulation of an asymmetric network, where this is even more of a problem. On an asymmetric network with a high bandwidth inbound link and low bandwidth outbound link, an inbound data transfer will fill the outbound link with acks at a packet/second rate that is likely to exceed that of any outbound data packets.

### *B. The Potential Bandwidth Filtering Solution*

The general idea of Potential Bandwidth Filtering is that we should correlate the potential bandwidth and measured bandwidth of a sample in deciding how to filter. Samples with the same potential bandwidth and measured bandwidth are not particularly informative because the actual bandwidth could be much higher. Samples with a high measured bandwidth and low potential bandwidth are time compressed and should be filtered out. Samples with a high potential bandwidth and low measured bandwidth are the most informative because they are likely to indicate the true bandwidth.

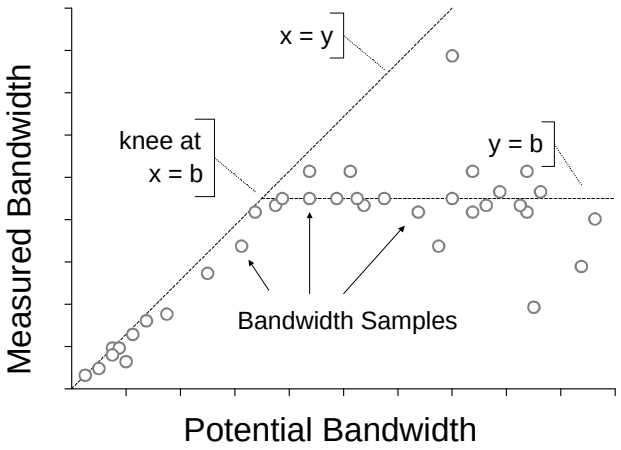


Fig. 2. This graph shows how PBF works. The dots represent bandwidth samples plotted using their potential and measured bandwidth. All samples above the $x = y$ line are filtered out. Notice how there is a knee in the samples.

We implement the algorithm by plotting all the samples on a graph with potential bandwidth on the x-axis and measured bandwidth on the y-axis. An example is shown in Figure 2. We would expect that in the absence of congestion, the samples would fall along the line $x = y$ until some point $x = b$. These samples have potential bandwidth approximately equal to measured bandwidth. The packets that generated these samples did not queue behind each other at the bottleneck link. After $b$ the samples should run along the line $y = b$. These are the samples with a higher potential bandwidth than measured bandwidth. The packets that generated these samples did queue behind each other at the bottleneck link. The value $b$ is the actual bandwidth.

If the samples never divert from the line $x = y$, then we know that our samples had insufficient potential bandwidth. For example, this would be case if we only had the samples to the left of $x = b$ in Figure 2.In this case, we should try an active algorithm.

To compensate for noise, we fit the $x = y$ and $y = b$ lines to the data and compute the relative error for each point as the distance of that point to the nearest line divided by the x-value of that point. This ensures that errors when x is large do not dominate the calculation. We then sum the errors for all the points and attempt to minimize the sum to choose the optimal $b$.

Our results show that PBF is just as accurate as MBF on an Ethernet network and 37% to 435% more accurate than MBF on an asymmetric network (see Section VII-C). We believe

that PBF is essential to the practical use of passive Packet Pair algorithms.

## VI. Simulation Environment

In this section, we discuss why we use a network simulator, how we simulated the network and why we believe the results are valid.

We use a network simulator because 1) we want verifiable and reproducible results, 2) we want to test the algorithms in a variety of conditions, and 3) we believe the limitations of current simulator technology have limited and accountable effects on our experiments. We discuss this final point in Section VI-B.

### *A. Simulator Goals and Setup*

In this section, we describe our goals for the simulator and how we configure it to meet those goals.

Our goal for the simulation is to stress the algorithms in both optimal and pathological conditions. We want to know how the worst possible conditions affect these algorithms. The bottleneck bandwidth algorithms are affected by the following conditions:

*1. Lack of Queueing at Bottleneck Link* This destroys the causality between packet arrival times and the bottleneck bandwidth.

*2. Queueing after Bottleneck Link* This destroys the causality between packet arrival times and the bottleneck bandwidth.

*3. Packet Loss* This causes algorithms to take longer to converge.

*4. Changing Bottleneck Bandwidth* Some algorithms detect this faster than others.

*5. Asymmetric Bandwidth* This could cause algorithms that assume symmetric bandwidth paths to fail. In particular, TCP packets arriving through a high bandwidth downlink will cause many acks to exit the low bandwidth uplink. These low potential bandwidth acks may cause MBF algorithms to fail (see Section V-A).

To model these conditions in a controlled manner, we used the `ns` network simulator [10]. We generated an 87 node network using the `tiers` topology generator [3]. `tiers` generates a network that reflects the semi-hierarchical topology of the Internet. The topology consists of 4 Wide Area Network (WAN) nodes, 16 Metropolitan Area Network (MAN) nodes, and 67 Local Area Network (LAN) nodes and includes redundant links between different MAN nodes and LAN nodes.

The client is usually 9 hops from the server and sometimes as many as 14 hops away, depending on which links have failed.

The traffic measured is one TCP connection from the client to the server beginning at 0.5 seconds into the simulation. The client and server are on different LANs and MANs.

TABLE I
Types of Client Connections.

| Link Type | Bandwidth | Latency |
|---|---|---|
| Cable Modem Uplink | 500Kb/s | 3ms |
| Cable Modem Downlink | 10Mb/s | 3ms |
| Ethernet | 10Mb/s | 3ms |

TABLE II
Traffic Source Parameters: This table lists the traffic source parameters used by `ns` in our simulations. Size is the size of packets. Burst and Idle give the average on and off times for sending packets. Shape is the shape parameter used for the Pareto distribution generator.

| Size | Burst | Idle | Shape |
|---|---|---|---|
| 1500 bytes | 1000ms | 500ms | 1.5 |
| 576 bytes | 500ms | 1000ms | 1.5 |
| 41 bytes | 50ms | 1000ms | 1.5 |

The simulation runs for 30 seconds of simulation time. The different link characteristics are summarized in Table III.

We varied three simulation parameters: client connectivity, congestion, and link failure model. We used the two client connections listed in Table I. Only the client is connected to the network using one of the client connections. All other nodes use links described in Table III.

We created congestion by placing three traffic sources at each LAN node. Each source sends data according to a Pareto distribution [7]. The parameters for these traffic sources are summarized in Table II. We varied congestion by using average data rates of 0Kb/s, 400Kb/s, and 1Mb/s. The variety of levels of congestion allows us to explore situations where the packets to and from the client did not queue together at the bottleneck link and/or did queue after the bottleneck link.

We varied the link failure model by using either no failure or a deterministic failure model where selected links along the path from client to server fail at specific times. The first

TABLE III
Simulator Link Characteristics: Type 1 links are used to carry packets unless they fail, in which case type 2 links are used.

| Type | From | To | Modeling | BW | Latency |
|---|---|---|---|---|---|
| 1 | WAN | WAN | T3 | 44Mb/s | 40ms |
| 1 | WAN | MAN | Ethernet | 10Mb/s | 20ms |
| 1 | MAN | MAN | Ethernet | 10Mb/s | 10ms |
| 1 | MAN | LAN | Ethernet | 10Mb/s | 10ms |
| 1 | LAN | LAN | Ethernet | 10Mb/s | 5ms |
| 2 | WAN | MAN | T1 | 1.5Mb/s | 20ms |
| 2 | MAN | LAN | T1 | 1.5Mb/s | 20ms |

link fails for 5.0 seconds beginning at 10.0 seconds. The second link fails for 6.0 seconds at 20.0 seconds. We chose the following two links for failure: client LAN to client MAN, and WAN to server MAN. Link failures cause packets to be lost, and when combined with redundant links, create the possibility for multiple paths, asymmetric bandwidth, and changing bottleneck bandwidth.

### B. Simulator Validity

We believe that the limitations of current simulator technology have limited effect on our results. Although the Internet exhibits effects that no current simulator can reproduce [13], our results do not depend on having high fidelity. Our goal is not to determine precisely how well these algorithms perform in the Internet on average. Our goal is to compare how well these algorithms perform under certain conditions known to exist in the Internet.

## VII. Simulation Results

In this section we present the simulation results. The following tables show the accuracy of Packet Pair algorithms and how they react to changes in network conditions. We use gradual versions of all the algorithms described in Section IV, so we compute a bandwidth estimate for every packet arrival. We then calculate the difference from the estimate to the real bandwidth at that point in time (the real bandwidth varies in some of the simulations). We express this difference as a ratio of the error to the actual bandwidth. The tables show the mean of these ratios. For example, a 0.10 mean error indicates that the algorithm's estimate deviated by an average of $\pm 10\%$ from the actual bandwidth.

In the tables shown later, the Alg. column describes which algorithm we are using: Sender Based (SB), Receiver Based (RB), and Receiver Only (RO). The Filter column describes the filtering algorithm used. The BW column lists the actual bottleneck bandwidth of the route between sender and receiver. The Fail column lists whether links fail in the simulation. The $w$ column gives the size of the packet window. The Traffic column gives the amount of extra traffic simulation. The Mean, $\sigma$, Med., and Max. columns describe the mean, standard deviation, median, and maximum, respectively, of the ratio of the estimate error to actual bandwidth.

For the graph, we plot the bandwidth measured against elapsed time in the flow. We collect measurements at every packet arrival. Packet arrivals are not evenly distributed in time, so the points are not evenly distributed along the x-axis. In each graph we also plot the actual bandwidth so we can gauge the accuracy of each algorithm. Note that all graphs in this section plot bandwidth on a log scale starting at 10,000 b/s rather than 1 b/s.

### A. Receiver Only Measurements

In this section we compare the accuracy of Sender Based Packet Pair, Receiver Based Packet Pair, and the new Receiver Only Packet Pair. We configure the simulator to use Ethernet as the client technology and use either no congestion or 400Kb/s of congestion.

TABLE IV
THIS TABLE COMPARES THE ACCURACY OF VARIOUS PACKET PAIR ALGORITHMS DEPENDING ON WHETHER THEY ARE SENDER BASED, RECEIVER BASED, OR RECEIVER ONLY.

| Alg. | $w$ | Traffic | Mean | $\sigma$ | Med. | Max. |
|---|---|---|---|---|---|---|
| SB | $\infty$ | 0Kb/s | 0.001 | 0.026 | 0.000 | 0.998 |
| RB | $\infty$ | 0Kb/s | 0.001 | 0.028 | 0.000 | 0.998 |
| RO | $\infty$ | 0Kb/s | 0.001 | 0.035 | 0.000 | 0.998 |
| SB | $\infty$ | 400Kb/s | 12.204 | 12.264 | 1.000 | 25.667 |
| RO | $\infty$ | 400Kb/s | 0.009 | 0.051 | 0.006 | 0.998 |
| RB | $\infty$ | 400Kb/s | 0.008 | 0.034 | 0.005 | 0.998 |

TABLE V
THIS TABLE COMPARES THE ACCURACY OF VARIOUS PACKET PAIR ALGORITHMS WITH VARYING WINDOW SIZE AND VARYING CONGESTION.

| Alg. | Fail | $w$ | Traffic | Mean | $\sigma$ | Med. | Max. |
|---|---|---|---|---|---|---|---|
| RB | N | $\infty$ | 400Kb/s | 0.009 | 0.051 | 0.006 | 0.998 |
| RB | N | 128 | 400Kb/s | 0.005 | 0.037 | 0.000 | 0.998 |
| RB | N | 32 | 400Kb/s | 0.110 | 0.252 | 0.000 | 0.998 |
| RB | Y | $\infty$ | 0Kb/s | 1.705 | 2.598 | 0.000 | 5.667 |
| RB | Y | 128 | 0Kb/s | 0.602 | 1.463 | 0.000 | 5.667 |
| RB | Y | 32 | 0Kb/s | 0.263 | 0.794 | 0.000 | 5.667 |

The results are summarized in Table IV. With no congestion, all of the Packet Pair algorithms have less than 1% error. With 400Kb/s of congestion, the 1200% error of SBPP is probably too much for most applications, while the error of RBPP and ROPP are still less than 1%.

These results confirm our assertion in Section IV-D that ROPP can achieve an accuracy close to that of RBPP, while maintaining the ease of deployment of SBPP.

### B. Congestion Tolerance and Detecting Bandwidth Change

In this section, we explore how well RBPP tolerates congestion and detects bandwidth changes. We use RBPP because it is more accurate than SBPP and ROPP and we wanted to isolate the effects of our new algorithms. We enable RBPP to detect bandwidth changes by setting a packet window size ($w$) of less than $\infty$. The question is whether our assertions in Section IV-E are accurate that a larger window size will be more resistant to congestion and a smaller window size will adapt more quickly to bandwidth changes. The results indicate that this assertion is correct.

Table V summarizes the statistics. The first three lines show the accuracy of three different window sizes when experiencing moderate congestion. As expected, smaller window values give less accurate results. However, even the 11% average error of the $w = 32$ estimate is probably tolerable for

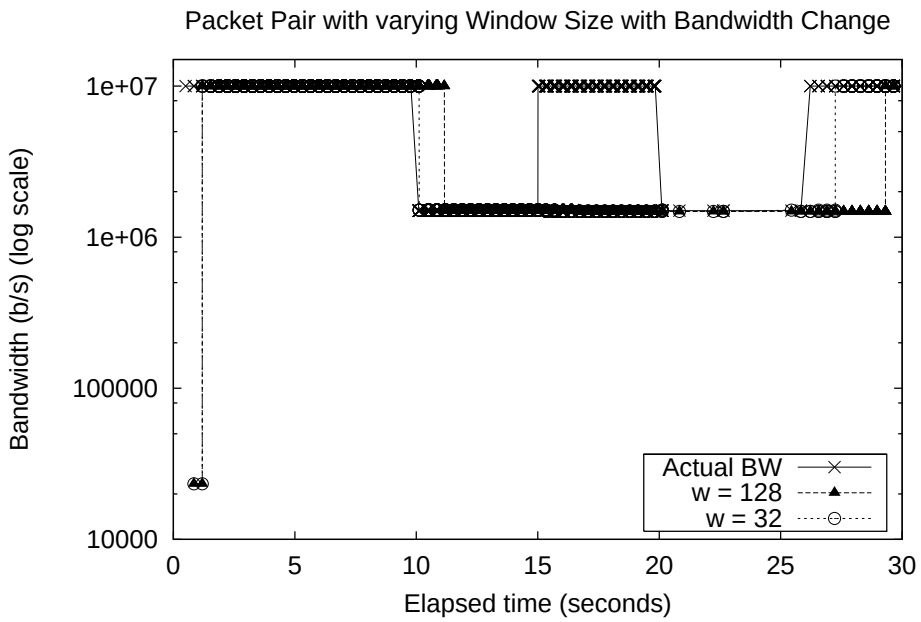


Fig. 3. This graph shows the effect of varying window size (w) in an Ethernet client simulation with varying bandwidth and no congestion. Notice the change in actual bandwidth at 10, 15, 20 and 26 seconds.

many applications.

The next three lines show how different window sizes affect agility. To test the agility of smaller window sizes in adapting to changing bandwidth, we configure the simulator to shut down the primary links periodically and route traffic through lower bandwidth secondary links (see Section VI). When we use all the packets from the beginning of the connection (i.e. $w = \infty$), RBPP has a significant error. As we would expect, the error decreases as we decrease the window size. The estimate with $w = 32$ is 144% more accurate than the $w = \infty$ estimate.

To visualize the effect of the different window sizes, we plotted the estimated and actual bandwidth in Figure 3. Notice the changes in actual bandwidth (the thin solid line) at 10, 15, 20 and 26 seconds. The actual bandwidth begins at 10Mb/s (the bottleneck bandwidth of the primary route), dips to 1.5Mb/s (the bottleneck bandwidth of the secondary route) at 10 seconds, rises again to 10Mb/s at 15 seconds and switches again between these values at 20 and 26 seconds. We removed the $w = \infty$ plot from this graph because it always remains at 10Mb/s and obscures the other plots. Notice that all the estimates jump to the correct estimate within three packets of the start of the connection. The plot with $w = 32$ adapts to the change at $t = 10$ almost instantly, while the $w = 128$ plot is slower to adapt. At the $t = 26$ change, the $w = 32$ plot is again more agile than the $w = 128$ plot.

Strangely, neither plot adapts to the $t = 15$ change. Examination of the trace revealed that the TCP code in ns was not increasing its window as it should have. Therefore, it was sending packets with a potential bandwidth of only 1.5Mb/s.

TABLE VI
THIS TABLE COMPARES THE ACCURACY OF DIFFERENT FILTERING ALGORITHMS.

| Alg. | Filter | BW | Mean | $\sigma$ | Med. | Max. |
|---|---|---|---|---|---|---|
| SB | MBF | 10Mb/s | 0.001 | 0.000 | 0.020 | 0.998 |
| SB | PBF | 10Mb/s | 0.001 | 0.000 | 0.020 | 0.998 |
| RB | MBF | 10Mb/s | 0.001 | 0.000 | 0.020 | 0.998 |
| RB | PBF | 10Mb/s | 0.001 | 0.000 | 0.020 | 0.998 |
| SB | MBF | 500Kb/s | 0.442 | 2.368 | 0.250 | 25.667 |
| SB | PBF | 500Kb/s | 0.078 | 0.268 | 0.000 | 0.998 |
| RB | MBF | 500Kb/s | 4.355 | 3.394 | 7.000 | 7.000 |
| RB | PBF | 500Kb/s | 0.000 | 0.021 | 0.000 | 0.998 |

As we discussed in Section V-A, Packet Pair algorithms cannot report a measured bandwidth higher than the potential bandwidth.

We conclude from these results that we must decrease the window size to detect changes in bandwidth quickly. This supports the conclusion in Section IV-E that we must make a tradeoff between timeliness and accuracy in choosing a window size.

### C. Potential Bandwidth Filtering

In this section, we investigate the effectiveness of our new filtering algorithm, PBF. As discussed in Section V-B, PBF is designed to overcome the problems that the standard Measured Bandwidth Filtering (MBF) algorithm has on traffic with mostly low potential bandwidth packets. It would also be desirable if PBF performed no worse than MBF on traffic where most of the packets have high potential bandwidth. In

order to test PBF, we used our simulated Ethernet network for the mostly high potential bandwidth traffic and the uplink of a simulated asymmetric cable modem network for the mostly low potential bandwidth traffic. In addition to a varying amount of overall congestion, we also set up a second TCP connection between the same two hosts as the first connection, but in the reverse direction. This ensures that there are at least a few high potential bandwidth packets in the outbound direction.

The results are summarized in Table VI. The first four lines show that PBF is equivalent to MBF on the Ethernet network. The next four lines show that PBF is anywhere from 37% to 435% more accurate on average than MBF on the cable modem network. MBF also has a significantly lower median than PBF, so MBF's poor average accuracy cannot be blamed on a few outliers. Examination of the trace verifies the analysis of Section V that the outbound link is filled with acks which overwhelm the few data packets. This causes MBF to incorrectly report the bandwidth of the acks as the true bandwidth. PBF is able to filter out those samples and discern the true bandwidth.

## VIII. Future Work

In the future, we are interested in simulating more networks and algorithms, calculating different metrics, and testing our ideas in the Internet. One type of network we did not simulate is a wireless network. `ns` has support for wireless networks, but this was not fully functional at the time we did our experiments. Wireless networks are interesting to examine because they tend to have high loss rates and high variance in latency, both situations that would challenge Packet Pair algorithms. In addition, we would like to simulate the Pathchar algorithm so that we can compare its accuracy to the passive techniques.

We would like to apply the methods described here to calculate available bandwidth. As mentioned in section II-B, some applications would find that a more useful metric. We believe that the methods described here would apply with some minor modifications.

We are currently using these bottleneck bandwidth measurement algorithms to implement `nettimer`, which can take live measurements from the Internet.

## IX. Conclusion

We examined the characteristics of current bandwidth measurement techniques and found several problems. We propose statistically robust algorithms which overcome these problems by giving timely estimates, being agile in the face of bandwidth changes, giving more flexibility in deployment, and working with a variety of different traffic types. Our simulation results show that our implementation is more than 37% more accurate than previous techniques.

We conclude that accurate, flexible and scalable bandwidth measurement is not only possible, but desirable in order to maintain the growth and reliability of many Internet applications.

## X. Acknowledgments

We would like to acknowledge the help of several people. Stuart Cheshire provided the code for `utimer`, which was the inspiration for `nettimer`. Guido Appenzeller suggested investigating robust methods of calculating density. Marcos de Alba pointed out an error in the `pathchar` analysis. Vern Paxson gave us an early copy of `tcpanaly` and provided valuable feedback on Potential Bandwidth Filtering. Craig Partridge provided advice on the motivation. Finally, we would like to thank the many anonymous reviewers for their feedback.

## References

[1] Mary G. Baker, Xinhua Zhao, Stuart Cheshire, and Jonathan Stone. Supporting mobility in mosquitonet. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[2] Jean-Chrysostome Bolot. End-to-end packet delay and loss behavior in the internet. In *Proceedings of SIGCOMM*, 1993.

[3] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 1997.

[4] 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.

[5] Robert L. Carter and Mark E. Crovella. Measuring bottleneck link speed in packet-switched networks. Technical Report BU-CS-96-006, Boston University, 1996.

[6] Stuart Cheshire and Mary Baker. Experiences with a wireless network in mosquitonet. In *Proceedings of the IEEE Hot Interconnects Symposium*, 1995.

[7] William Feller. *An Introduction to Probability Theory and its Applications, volume II*. Wiley Eastern Limited, 1988.

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

[9] Srinivasan Keshav. A control-theoretic approach to flow control. In *Proceedings of SIGCOMM*, 1991.

[10] Steven McCanne, Sally Floyd, Kevin Fall, and Kannan Varadhan et al. ns. http://www-mash.cs.berkeley.edu/ns/, 1997.

[11] Vern Paxson. End-to-end internet packet dynamics. In *Proceedings of SIGCOMM*, 1997.

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

[13] Vern Paxson and Sally Floyd. Why we don't know how to simulate the internet. In *Proceedings of the 1997 Winter Simulation Conference*, 1997.

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

[15] Srinivasan Seshan, Mark Stemm, and Randy Katz. Spand: Shared passive network performance discovery. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[16] Ronald A. Thisted. *Elements of Statistical Computing: Numerical Computation*. Chapman and Hall, 1988.