Sting: a TCP-based Network Measurement Tool

Stefan Savage
Department of Computer Science and Engineering
University of Washington, Seattle
savage@cs.washington.edu

Abstract

Understanding wide-area network characteristics is critical for evaluating the performance of Internet applications. Unfortunately, measuring the end-to-end network behavior between two hosts can be problematic. Traditional ICMP-based tools, such as `ping`, are easy to use and work universally, but produce results that are limited and inaccurate. Measurement infrastructures, such as NIMI, can produce highly detailed and accurate results, but require specialized software to be deployed at both the sender and the receiver. In this paper we explore using the TCP protocol to provide more accurate network measurements than traditional tools, while still preserving their near-universal applicability. Our first prototype, a tool called *sting*, is able to accurately measure the packet loss rate on both the forward and reverse paths between a pair of hosts. We describe the techniques used to accomplish this, how they were validated, and present our preliminary experience measuring the packet loss rates to and from a variety of Web servers.

1 Introduction

Measuring the behavior between Internet hosts is critical for diagnosing current performance problems as well as for designing future distributed services. Unfortunately, the Internet architecture was not designed with performance measurement as a primary goal and therefore has few "built-in" services that support this need [Cla88]. Consequently, measurement tools must either "make do" with the services provided by the Internet, or deploy substantial new infrastructures geared towards measurement.

In this paper, we argue that the behavior of the commonly deployed Transmission Control Protocol (TCP) can be used as an implicit measurement service. We present a new tool, called *sting*, that uses TCP to measure the packet loss rates between a source host and some target host. Unlike traditional loss measurement tools, sting is able to precisely distinguish which losses occur in the forward direction on the path to the target and which occur in the reverse direction from the target back to the source. Moreover, the only requirement of the target host is that it run some TCP-based service, such as a Web server.

The remainder of this paper is organized as follows: In section 2 we review the current state of practice for measuring packet loss. Section 3 contains a description of the basic loss deduction algorithms used by sting, followed by extensions for variable packet size and interarrival times in section 4. We briefly discuss our implementation in section 5 and present some preliminary experiences using the tool in section 6.

2 Measuring packet loss

The rate at which packets are lost can have a dramatic impact on application performance. For example, it has been shown that for moderate loss rates (less than 15 percent) the bandwidth delivered by TCP is proportional to $1/\sqrt{lossrate}$ [MSM97]. Similarly, some streaming media applications only perform adequately under low loss conditions [CB97]. Not surprisingly, there has always been a long-standing operational need to measure packet loss; the popular `ping` tool was developed less than a year after the creation of the Internet. In the remainder of this section we'll discuss two dominant methods for measuring packet loss: tools based on the Internet Control Message Protocol (ICMP) [Pos81] and new measurement infrastructures.

2.1 ICMP-based tools

Common ICMP-based tools, such as `ping` and `traceroute`, send probe packets to a host, and measure loss by observing whether or not response packets arrive within some time period. There are two principle problems with this approach:

> *Loss asymmetry.* The packet loss rate on the forward path to a particular host is frequently quite different from the packet loss rate on the reverse path from that host. Without any additional information from the receiver, it is impossible for an ICMP-based tool to determine if its probe packet was lost or if the response was lost. Consequently, the loss rate reported by such tools is really:
>
> $$1 \quad ((1 \quad loss \quad) \ (1 \quad loss \quad))$$
>
> Where $loss$ is the loss rate the forward direction and $loss$ is the loss rate in the reverse di-

rection. Loss asymmetry is important, because for many protocols the relative importance of packets flowing in each direction is different. In TCP, for example, losses of acknowledgment packets are tolerated far better than losses of data packets. Similarly, for many streaming media protocols, packet losses in the opposite direction from the data stream have little or no impact on overall performance. The ability to measure loss asymmetry allows a network engineering to more precisely locate important network bottlenecks.

ICMP filtering. ICMP-based tools rely on the near-universal deployment of the *ICMP Echo* or *ICMP Time Exceeded* services to coerce response packets from a host [Bra89]. Unfortunately, malicious use of ICMP services has led to mechanisms that restrict the efficacy of these tools. Several host operating systems (e.g. Solaris) now limit the rate of ICMP responses, thereby artificially inflating the packet loss rate reported by `ping`. For the same reasons many networks (e.g. microsoft.com) filter ICMP packets altogether. Some firewalls and load balancers respond to ICMP requests on behalf of the hosts they represent, a practice we call *ICMP spoofing*, thereby precluding real end-to-end measurements. Finally, at least one network has started to rate limit all ICMP traffic traversing it. It is increasingly clear that ICMP's future usefulness as a measurement protocol will be reduced [Rap98].

2.2 Measurement infrastructures

In contrast, wide-area measurement infrastructures, such as NIMI and Surveyor, deploy measurement software at both the sender and the receiver to correctly measure one-way network characteristics [Pax96, PMAM98, Alm97]. Such approaches are technically ideal for measuring packet loss because they can precisely observe the arrival and departure of packets in both directions. The obvious drawback is that the measurement software is not widely deployed and therefore measurements can only be taken between a restricted set of hosts. Our work does not eliminate the need for such infrastructures, but allows us to extend their measurements to include parts of the Internet that are not directly participating. For example, access links to Web servers can be highly congested, but they are not visible to current measurement infrastructures.

Finally, there is some promising work that attempts to derive per-link packet loss rates by correlating measurements of multicast traffic among many different hosts [CDH 99]. The principle benefit of this approach is that it allows the measurement of N paths with $O(N)$ messages. The slow deployment of wide-area multicast routing currently limits the scope of this technique, but this situation may change in the future. However, even with universal multicast routing, multicast tools require software to be deployed at many different hosts, so, like other measurement infrastructures, there will likely still be significant portions of the commercial Internet that can not be measured with them.

Our approach is similar to ICMP-based tools in that it only requires participation from the sender. However, unlike these tools, we exploit features of the TCP protocol to deduce the direction in which a packet was lost. In the next section we describe the algorithms used to accomplish this.

3 Loss deduction algorithm

To measure the packet loss rate along a particular path, it is necessary to know how many packets were sent from the source and how many were received at the destination. From these values the one-way loss rate can be derived as:

$$1 \quad (packetsreceived/packetssent)$$

Unfortunately, from the standpoint of a single endpoint, we cannot observe both of these variables directly. The source host can measure how many packets it has sent to the target host, but it cannot know how many of those packets are successfully received. Similarly, the source host can observe the number of packets it has received from the target, but it cannot know how many more packets were originally sent. In the remainder of this section we will explain how TCP's error control mechanisms can be used to derive the unknown variable, and hence the loss rate, in each direction.

3.1 TCP basics

Every TCP packet contains a 32 bit sequence number and a 32 bit acknowledgment number. The sequence number identifies the bytes in each packet so they may be ordered into a reliable data stream. The acknowledgment number is used by the receiving host to indicate which bytes it has received, and indirectly, which it has not. When in-sequence data is received, the receiver sends an acknowledgment specifying the next sequence number that it expects and implicitly acknowledging all sequence numbers preceding it. Since packets may be lost, or reordered in flight, the acknowledgment number is only incremented in response to the arrival of an in-sequence packet. Consequently, out-of-order or lost packets will cause a receiver to issue duplicate acknowledgments for the packet it was expecting.

```
for i := 1 to n                          for each ack received
  send packet w/seq# i                     ackReceived++
  dataSent++
  wait for long time
```

Figure 1: Data seeding phase of basic loss deduction algorithm.

```
lastAck := 0                             for each ack received w/seq# j
while lastAck = 0                          lastAck = MAX(lastAck, j)
  send packet w/seq# n+1

while lastAck < n + 1
  dataLost++
  retransPkt := lastAck
  while lastAck = retransPkt
     send packet w/seq# retransPkt

dataReceived := dataSent - dataLost
ackSent := dataReceived
```

Figure 2: Hole filling phase of basic loss deduction algorithm.

3.2 Forward loss

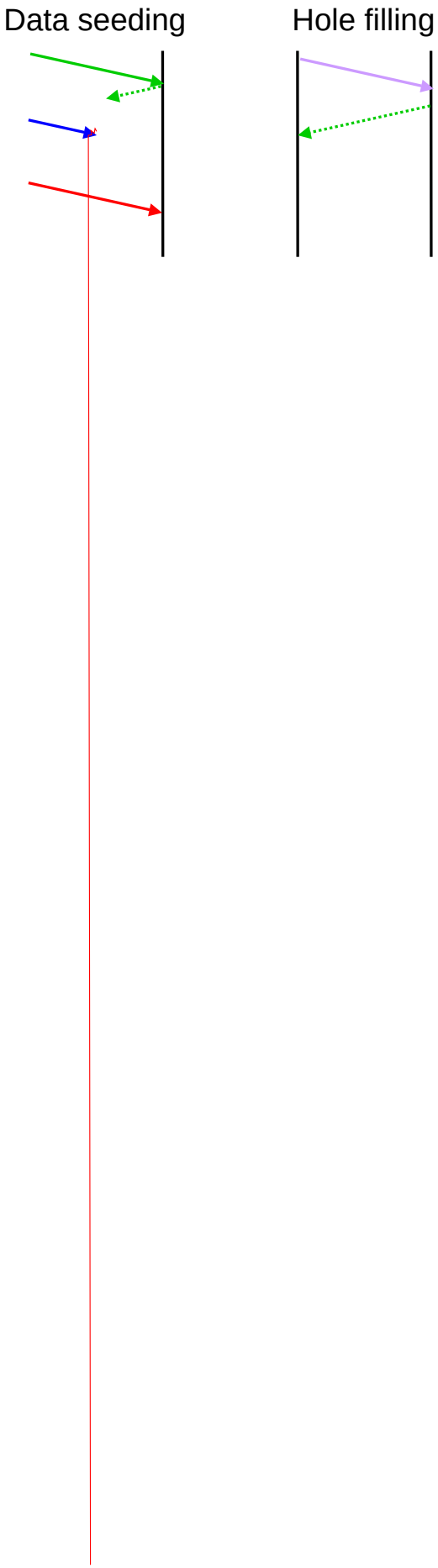
Deriving the loss rate in the forward direction, from source to target, is straightforward. The source host can observe how many data packets it has sent, and then can use TCP's error control mechanisms to query the target host about which packets were received. Accordingly, we divide our algorithm into two phases:

Data-seeding. During this phase, the source host sends a series of in-sequence TCP data packets to the target. Each packet sent represents a binary sample of the loss rate, although the value of each sample is not known at this point. At the end of the data-seeding phase, the measurement period is concluded and any packets lost after this point are not counted in the loss measurement.

Hole-filling. The hole-filling phase is about discovering which of the packets sent in the previous phase have been lost. This phase starts by sending a TCP data packet with a sequence number *one greater* than the last packet sent in the data-seeding phase. If the target responds by acknowledging this packet, then no packets have been lost. However, if any packets have been lost there there will be a "hole" in the sequence space and the target will respond with an acknowledgment indicating exactly where the hole is. For each such acknowledgment, the source host retransmits the corresponding packet, thereby "filling the hole", and records that a packet was lost. We repeat this procedure until the last packet sent in the data-seeding phase has been acknowledged. Unlike data-seeding, hole-filling must be reliable and so the implementation must timeout and retransmit its packets when expected acknowledgments do not arrive.

3.3 Reverse Loss

Deriving the loss rate in the reverse direction, from target to source, is somewhat more problematic. While the source host can count the number of acknowledgments it receives, it is difficult to be certain how many acknowledgments were sent. The ideal condition, which we refer to as *ack parity* is that the target sends a single acknowledgment for every data packet it receives. Unfortunately, most TCP implementations use a *delayed acknowledgment* scheme that does not provide this guarantee. In these implementations, the receiver of a data packet does not respond immediately, but instead waits for an additional packet in the hopes that the cost of sending an acknowledgment can be amortized [Bra89]. If a second packet has not arrived within some small timeout (the standard limits this delay to 500ms, but 100-200ms is a common value) then the receiver will issue an acknowledgment. If a second packet does arrive before the timeout, then the receiver will issue an acknowledgment immediately. Consequently, the source host cannot reliably


Data seeding
Hole filling

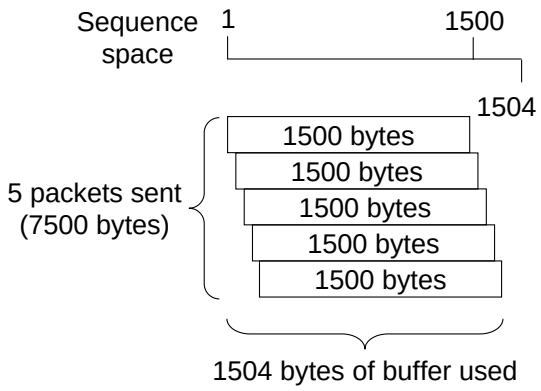


Figure 4: Mapping packets into sequence numbers by overlapping sequence numbers.

1500 byte packets, a number far too small to be statistically significant. While we could simply create a new connection and restart the tool, this limitation prevents us from exploring larger packet bursts.

Luckily, we observe that TCP implementations *trim* packets that overlap the sequence space that has already been received. Consequently, if a packet arrives that overlaps a previously received packet, then the receiver will only buffer the portion that occupies "new" sequence space. By explicitly overlapping the sequence numbers of our probe packets we can map each large packet into a single byte of sequence space, and hence only a single byte of buffer at the receiver.

Figure 4 illustrates this technique. The first 1500 byte packet is sent with sequence number 1500, and when it arrives at the target it occupies 1500 bytes of buffer space. However, the next 1500 byte packet is sent with sequence number 1501. The target will note that the first 1499 bytes of this packet have already be received, and will only use a single byte of buffer space. Using this technique we can map every additional packet into a single sequence number, eliminating much of the buffering limitation. This technique only allows us to send bursts of data in one direction – towards the target host. Coercing the target host to send arbitrarily sized bursts of data back to the source is more problematic since TCP's congestion control mechanisms normally control the rate at which the target may send data. We have investigated techniques to remotely bypass TCP's congestion control [SCWA99] but we believe they represent a security risk and aren't suited for common measurement tasks.

4.3 Delaying connection termination

One final problem is that some TCP servers do not close their connections in a graceful fashion. TCP connections are full-duplex – data flows along a connection in both directions. Under normal conditions, each "half" of the connection may only be closed by the sending side (by sending a FIN packet). Our algorithms implicitly assume this is true, since it is necessary that the target host respond with acknowledgments until the testing period is complete. While most TCP-based servers follow this termination protocol, we've found that some Web servers simply terminate the entire connection by sending a RST packet – sometimes called an *abortive release*. Once the connection has been reset, the sender discards any related state so any further probing is useless and our measurement algorithms will fail.

To ensure that our algorithms have sufficient time to execute, we've developed two ad hoc techniques for delaying premature connection termination. First, we ensure that the data sent during the data seeding phase contains a valid HTTP request. Some Web servers (and even some "smart" firewalls and load balancers) will reset the connection as soon as the HTTP parser fails. Second, we use TCP's *flow control* protocol to prevent the target from actually delivering its HTTP response back to the source. TCP receivers implement flow control by advertising the number of bytes they have available for buffering new data (called the *receiver window*). A TCP sender is forbidden from sending more data than the receiver claims it can buffer. By setting the source's receiver window to zero bytes we can keep the HTTP response "trapped" at the target host until we have completed our measurements. The target will not reset the connection until its response has been sent, so this technique allows us to inter-operate with such "ill-behaved" servers.

5 Implementation

In principle, it should be straightforward to implement the loss deduction algorithms we have described. How-

5.1 Building a user-level TCP

Most operating systems provide two mechanisms for low-level network access: *raw sockets* and *packet filters*. A raw socket allows an application to directly format and send packets with few modifications by the underlying system. Using raw sockets it is possible to create our own TCP segments and send them into the network. Packet filters allow an application to acquire *copies* of raw network packets as they arrive in the system. This mechanism can be used to receive acknowledgments and other control messages from the network. Unfortunately, another copy of each packet is also relayed to the TCP stack of the host operating system; this can cause some difficulties. For example, if sting sends a TCP SYN request to the target, the target responds with a SYN of its own. When the host operating system receives this SYN it will respond with a RST because it is unaware that a TCP connection is in progress.

An alternative implementation would be to use a secondary IP address for the sting application, and implement a user-level proxy ARP service. This would be simple and straightforward, but has the disadvantage that users of sting would need to request a second IP address from their network administrator. For this reason, we have resisted this approach.

Finally, many operating systems are starting to provide proprietary firewall interfaces (e.g. Linux, FreeBSD) that allow the user to filter outgoing or incoming packets. The former ability could be used to intercept packets arriving from the target host, while the later ability could be used to suppress the responses of the host operating system. We are investigating this approach for a future version.

5.2 The Sting prototype

Our current implementation is based on raw sockets and packet filters running on FreeBSD 3.x and Digital Unix 3.2. As a work-around to the SYN/RST problem mentioned previously, we use the standard Unix `connect()` service to create the connection, and then hijack the session in progress using the packet filter and raw socket mechanisms. Unfortunately, this solution is not always sufficient as the host system can also become confused by acknowledgments for packets it has never sent. In our current implementation we have been forced to change one line in the kernel to control such unwanted interactions.[1] We are currently unsure if a completely portable user-level implementation is possible on today's Unix systems.

Figure 5 shows the output presented by sting. From the command line the user can select the inter-arrival distribution between probe packets (periodic, uniform, or exponential), the distribution mean, the number of total packets sent, as well as the target host and port. Our implementation verifies that the wire time distribution conforms to the expected distribution according to the tests provided in [PAMM98].

```
# sting www.audiofind.com


Source = 128.95.2.93
Target = 207.138.37.3:80
dataSent = 100
dataReceived = 98
acksSent = 98
acksReceived = 97
Forward drop rate = 0.020000
Reverse drop rate = 0.010204
```

Figure 5: Sample output from the sting tool. By default, sting sends 100 probe packets according to a uniform inter-arrival distribution with a mean of 100ms.

We have tested our implementation in several ways. First, we have synthetically dropped packets in the tool and using an emulated network [Riz97] and verified that sting reports the correct loss rate. Second, we have compared the results of sting to results obtained from `ping`. Using the derivation for `ping`'s loss rate presented in section 2 we have verified that the the results returned by each tool are compatible. Finally, we have tested sting with a large number of different host operating systems, including Windows 95, Windows NT, Solaris, Linux, FreeBSD, NetBSD, AIX, IRIX, Digital Unix, and MacOS. While we occasionally encounter problems with very poor TCP implementations (e.g. laser printers) and Network Address Translation boxes, the tool is generally quite stable.

6 Experiences

Anecdotally, our experience in using sting has been very positive. We've had considerable luck using it to debug network performance problems on asymmetric access technologies (e.g. cable modems). We've also used it as a day-to-day diagnostic tool to understand the source of Web latency. In the remainder of this section we present some preliminary results from a broad experiment to quantify the character of the loss seen from our site to the rest of the Internet.

For a twenty four hour period, we used sting to record loss rates from the University of Washington to a collection of 50 remote web servers. Choosing a reasonably-sized, yet representative, set of server sites is a difficult task due to the diversity of connectivity and load expe-

[1] We modify the ACK processing in tcp_input.c so the response to an acknowledgment entirely above snd_max is to drop the packet instead of acknowledging it.