Scriptroute: A Public Internet Measurement Facility*

Neil Spring, David Wetherall and Tom Anderson
{*nspring,djw,tom*}*@cs.washington.edu*
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

Abstract

We present Scriptroute, a system that allows ordinary Internet users to conduct network measurements from remote vantage points. We seek to combine the flexibility found in dedicated measurement testbeds such as NIMI with the general accessibility and popularity of Web-based public traceroute servers. To use Scriptroute, clients use DNS to discover measurement servers and then submit a measurement script for execution in a sandboxed, resource-limited environment. The servers ensure that the script does not expose the network to attack by applying source- and destination-specific filters and security checks, and by rate-limiting traffic.

Scriptroute code is publicly available and has been deployed on the PlanetLab testbed of 42 sites. As proof-of-concept, we have used it both to create RPT, a tool for measuring routing trees toward a destination, and to repeat the experiment used to evaluate GNP, a recently proposed Internet distance estimation technique. We find that our system is flexible enough to implement a variety of measurement tools despite its security restrictions, that access to many remote vantage points makes the system valuable, and that scripting is an apt choice for expressing and combining measurement tasks.

1 Introduction

The ability to measure the Internet is of widespread value for diagnosing connectivity problems and understanding Internet topology [20, 53], routing [35, 54] and performance [3, 51]. This paper considers a simple question: what is the right architecture for a generally available network measurement facility?

Existing systems such as NIMI [45] provide much of the needed functionality, but not all. These research systems provide the advantages of dedicated hardware that can be used for a wide range of network measurements. In return, users must possess credentials or an account, which creates a barrier that limits access to a community of users trusted by the administrator. Thus these systems do not help unaffiliated users like a network operator trying to debug poor network performance.

The popularity of Web-accessible traceroute servers offers a different solution. Several hundred public traceroute servers are available, constituting the largest de facto Internet measurement facility. These servers are typically used to debug two-way connectivity problems, providing indirect benefit to the traceroute server host. They are also easy to secure, because they provide only limited functionality and local administrators retain control to deny access to abusive users. As a result, many network operators now contribute traceroute servers.

However, traceroute servers provide limited functionality – only a hop-by-hop TTL test – and have significant drawbacks when used as a measurement system. They are difficult to coordinate because they were not designed with programmed access in mind. They can be highly inefficient for some applications, such as our RPT tool described in Section 5.1. More importantly, there are many non-intrusive tests of path properties that are not supported by traceroute servers: tests for path MTU [17], available bandwidth [27, 55], capacity [33, 49], queuing and congestion [5], and reordering [4]. In short, it is clear that a much richer diagnostic and measurement capability would be possible with a general-purpose tool.

Our goal is to combine the best of both worlds: the flexibility to run a wide variety of different measurement tools with the general availability of traceroute servers. We begin with the safety properties of traceroute servers: we design the system to prevent misuse, even at the cost of disallowing some kinds of useful measurements. Our thesis is that even within the context of a carefully controlled interface, we can provide more functionality than is currently provided by traceroute servers. We hope to

*Appears in USENIX Symposium on Internet Systems and Technologies (USITS), 2003

succeed to the point where administrators will find it to their advantage to host a Scriptroute server in place of their current traceroute server.

We call our system Scriptroute. We use scripting to facilitate the implementation of measurement tools and the coordination of measurements across servers. For example, traceroute can be expressed in Scriptroute in tens of lines of code (Section 3), instead of hundreds; and tasks can be combined across servers in hundreds of lines (Section 5) instead of the thousands required in a previous project [53]. For security, we use sandboxing and local control over resources to protect the measurement host, and rate-limiting and filters that block known attacks to protect the network. Further, because network measurements often send probe traffic to random Internet hosts and administrators sometimes mistake measurement traffic for an attack, we provide a mechanism for sites to block unwanted measurement traffic.

While none of the pieces of the design are particularly new (e.g., others have sandboxed foreign code [18, 23]), we believe that the result is novel and can substantially improve our ability to make safe, flexible remote measurements. Further, part of our goal is to spark a debate as to how a network measurement facility should be architected. Because we could have made different design choices, we see our system as only one design point in the space of network measurement service architectures. More broadly, given the rising popularity of various forms of widely accessible remote execution facilities, e.g., Akamai, .NET, and seti@home, our work provides an example of how to balance the tradeoff between security and flexibility in this new class of systems.

We have implemented the Scriptroute design and deployed it on servers across 42 PlanetLab sites. The Scriptroute code is publicly available [52] and can be used for local measurement script development or for participation in the global system. To test the system, we have used this initial deployment to run RPT, a tool we created to measure routing trees around a destination, and to repeat the experiment used to evaluate GNP [40], a recently proposed Internet distance estimation technique. We find that our system will be flexible enough to implement a variety of new measurement tools despite its security restrictions, that access to many remote vantage points makes the system valuable, and that scripting is an apt choice for expressing and combining measurement tasks.

The rest of the paper is organized as follows. We describe our goals and approach in the next section and the design of our system in Section 3. We present implementation details such as the default configuration in Section 4. We evaluate our approach using two applications as case studies in Section 5, then conclude.

Measurement Tool	Measures	Support
pathchar [25]	Hop-by-hop b/w	✓[1]
pchar [34]	Hop-by-hop b/w	✓[1]
clink [15]	Hop-by-hop b/w	✓[1]
pathrate [13]	Bottleneck b/w	✓
pathload [27]	Available b/w	✓
sprobe [49]	Bottleneck b/w	✓
nettimer [33]	Bottleneck b/w	✓
bprobe [6]	Bottleneck b/w	✓
cprobe [5]	Congestion	✓
traceroute [26]	Path and RTT	✓
tcptraceroute [56]	Path and RTT	✓
ping	Round trip time	✓
zing [45]	Poisson RTT	✓
ally [53]	Alias resolution	✓[2]
tbit [41]	End-host TCP impl.	✓[2]
king [21]	Estimated RTT	✓[2]
nmap [16]	End-host services	✓[2]
treno [37]	TCP b/w	✗[3]
wping [36]	TCP b/w	✗[3]
iperf [55]	TCP b/w, loss	✗[3]
netperf [29]	TCP b/w	✗[3]
ttcp	TCP b/w	✗[3]
sting [50]	One-way loss	✗[4]
fsd [19]	Router processing	✗[5]

[1] May require an excessively high rate of traffic to execute quickly, which would be limited by policy.
[2] Measures end host properties: supported, so may simplify development, but unnecessary.
[3] Must keep a window of packets in flight, so are not supported by our synchronous interface.
[4] Supported by design, but unimplemented: requires safe raw sockets [47] or kernel firewall support.
[5] Requires address spoofing.

Table 1: Some active measurement tools supported by the design.

2 Goals and Approach

In this section, we describe our design philosophy and the approach that follows from it.

2.1 Philosophy

Our high-level goal is to foster the deployment of a community platform for distributed Internet measurement.

To be provided by the community, this platform must allow different organizations to manage their own portions of the infrastructure. To be of broad use, this platform must see widespread deployment to provide many measurement vantage points. To be of lasting value, this platform must be capable of hosting new measurement techniques.

While these goals are straightforward, achieving them is not: many promising systems fail to achieve widespread adoption. We observe two salient characteristics in successful collaborative systems, such as Gnutella and the Web. First, they are open: all users may contribute and participate. Second, they are valuable to the participants: there is benefit to both service users and providers.

Our philosophy is derived from interpreting these qualities in our domain of network measurement. To be open, we take the position that all users must be able to obtain useful levels of service by default and with negligible prior investments. If all users are authorized to obtain the same service then, just as a public Web server, there is no need to authenticate users further than their IP address. To provide value, we observe that the most compelling use of measurement staples such as traceroute and ping is not for network research, but for operational purposes. Indeed, the array of public traceroute servers is heavily populated by ISPs providing vantage points from which they may check routing and connectivity. We thus seek to seed our system with operationally useful measurement tools.

This philosophy leads to the essential conflict in the design of our system: flexibility versus security. Flexibility is required if we are to support unforeseen measurement tools. At the same time, supporting unauthenticated users poses serious security concerns. To be deployed, Scriptroute cannot serve as a vehicle that facilitates denial-of-service attacks on third parties, nor can it expose its host to attack. We next describe our approach to flexibility, then security.

2.2 Flexible Measurement Tools

Our goal is to provide Scriptroute servers with sufficient extensibility mechanisms that they can implement unanticipated measurement tools. While we cannot prove we can handle all possible new tools, we can design a system that supports their likely space. To define the space, we first considered existing active measurement tools, a sample of which (most from [11]) is shown in Table 1.

We observe that existing tools send a wide variety of types and sequences of packets, with different timing patterns, and using different methods of data analysis. Most of the tools, including some that measure bandwidth, require only a modest level of bandwidth and processing to be useful, and they do not impose tight timing coupling between the reception of one packet and the transmission of the next. The variability in functional details and modest resource requirements of these tools lead us to an architecture where measurements are supported by shipping measurement code to Scriptroute servers. This code is then interpreted in a resource-limited sandbox that includes an API for sending and receiving measurement packets and for reporting results back to the client.

We can also observe from Table 1 that there is a class of tools that need not be supported from distributed vantage points. Tools such as tbit and nmap, for example, probe properties of the endpoint being measured. They can readily be run from any vantage point to obtain the desired measurement. Similarly, tools such as King [21] work by finding unwitting proxy nodes as vantage points. These kind of tools are not targeted as part of the design of Scriptroute; we focus on tools that measure the properties of network paths that can only be observed by using Scriptroute servers themselves as vantage points.

2.3 Protecting Scriptroute Servers

We require that Scriptroute servers not expose their host to unwanted attack, despite an architecture where measurements are scripted and servers execute them on behalf of unauthenticated (and hence untrusted) clients. There are two aspects to protecting servers: restricting access and controlling resource consumption.

To isolate measurements from the host system, servers execute measurement scripts in the strongest sandbox we can construct that provides only a very narrow interface for sending and receiving packets and communicating results to the client. The design of this resource-limited sandbox is described in Section 3.

To ensure that measurement scripts do not consume enough resources to cause denial-of-service to the host, the Scriptroute server limits all aspects of measurement execution. Servers limit the duration, traffic rate, memory footprint, processor time, and number of concurrent measurements, reclaiming their resources as scripts terminate. Limits on the duration of measurements ensure that resources are replenished for subsequent measurements. Such limits prohibit long-lived experiments, but do away with allocation and reservation machinery. Similarly, measurements are not allowed access to local

Prevention Mechanism	Attack Classes Prevented
Verify packet is well-formed	Ping of Death [31]
Verify source address	UDP packet storm using echo/chargen [8]
Deny fragments	Overlapping IP fragments with conflicting data[9]
Deny ICMP error messages	Spurious host unreachable [12]
Deny broadcast	Smurfing [10]
SYNs rate-limited	SYN flooding [7]
Rate-limit traffic	Packet flooding (e.g. flood ping)

Table 2: Attacks prevented by Scriptroute policy. The top half consists of well-known "magic" packet attacks that are prevented with filters. Flooding attacks are prevented by rate-limiting.

storage, which simplifies the system but requires that the client store all intermediate state. Taken together, these limits embody a "best-effort" service model, where the Scriptroute server executes measurements only when resources are available.

2.4 Preventing Network Attacks

We require that Scriptroute servers not facilitate denial-of-service attacks on other parties, either by acting individually or as a whole. Unfortunately, this is a tall order: most Internet hosts can be unwitting participants in a denial-of-service attack, and just one unexpected packet can be interpreted as an attack by an intrusion detection system or watchful administrator. Since new attacks are discovered in existing protocol implementations with disappointing regularity, we also cannot reliably filter out attack packets at servers (e.g., by using an IDS setup) without engaging in an arms race. Instead, we set a lower bar for Scriptroute, which is that it not increase the danger to third parties, either by amplifying or laundering attacks. Attack traffic is amplified when attackers can cause many packets (or much work) to reach the target by sending few packets (or doing little work) themselves, e.g., smurfing [10]. Attack traffic is laundered when attackers cause a third party to send a packet that is not traceable to the true attacker [44].

To understand how to prevent attacks, we first considered the different kinds of attack traffic. A sample of known network attacks is listed in Table 2. We observe that these attacks fall into two classes: those that require only a few "magic" packets, and those that overwhelm targets with a flood of traffic or otherwise tie up system resources. We tackle each class differently. We also streamline the process by which recipients of unwanted measurement traffic can have it blocked.

To mitigate the first class of attacks, we block packets frequently used for attacks and infrequently needed for measurements, e.g., IP packets with broadcast destination addresses. The complete list is given in Section 4.2. We also provide accountability by ensuring that the source address of measurement traffic is that of the server and by logging client activity. The latter is possible because the TCP connection between client and server ensures that the client IP address is genuine. Together, these measures provide an identity chain that allows measurement traffic to be traced to its origin (at least, as far as Scriptroute is concerned) for more subtle attack packets that are not blocked. We note that "magic" packet attacks could be launched from anywhere in the network, probably with less effort and the same effect as via Scriptroute. That is, a Scriptroute server does not contribute to the vulnerability of the network.

The second class of attacks requires a sustained flood of traffic to arrive at the target. Our approach here is straightforward: we rate-limit measurement traffic to an acceptable, background level. This approach works well for the majority of measurement tools, many of which send small volumes of data at low rates to avoid altering the properties that they seek to measure. However, some measurement tools, primarily bandwidth estimators such as treno and pathchar, do send a large volume of high-rate traffic. We cannot safely support them in their current form and instead are hopeful that recent work on bandwidth estimation such as pathload, nettimer, and sprobe [27, 33, 49] will lead to lower rate, less intrusive tools.

We considered and discarded other approaches, such as "packet conservation," where high rates can be used, provided that send and receive traffic is roughly balanced. Unfortunately, while unbalanced traffic indicates a problem (such as high loss or deliberate discard), balanced traffic only indicates the absence of severe network congestion. Because of the best-effort nature of Internet services, request flooding (e.g., TCP connections, DNS requests) may consume nearly all available resources. Further, determining when traffic is too far out of balance is a task that depends on protocol semantics, such as delayed acknowledgements, and so it cannot be applied in a general way.

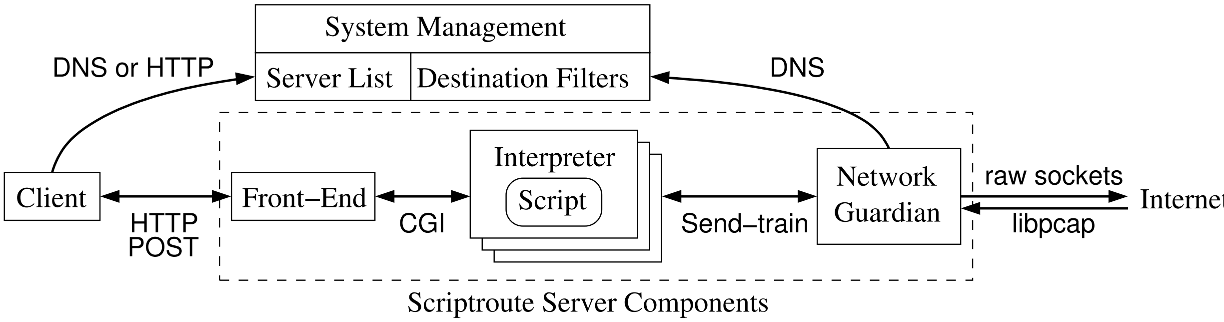

Figure 1: Scriptroute components. Clients discover servers through DNS or HTTP. Clients then submit measurements scripts to a server front-end using HTTP, which executes a new interpreter as a CGI program. Scripts running within the Interpreter use the Send-train API to send probes and receive responses. Only the network guardian can access the network, after first checking that a destination filter does not block requested probes.

Rate limits prevent a single measurement from overwhelming a destination, but we must also prevent the collection of Scriptroute servers being used for distributed denial-of-service (DDOS). Again, our approach in the short term is to rely on a sufficiently low rate limit on individual measurement that does not provide clients with leverage in terms of attack bandwidth. That is, if Scriptroute servers do not significantly amplify attack traffic levels then they do not make DDOS attacks any easier to launch.

Again, we considered more sophisticated centralized or epidemic controls that would detect groups of servers sending large volumes of traffic to the same target, e.g., by requiring that permission tokens be obtained from a pre-determined controller before starting a bandwidth-intense measurement. However, we realized that, even if the complexity issues associated with these controls can be managed, protection by destination IP address (or destination IP prefix) is not sufficient. This is because hosts other than the apparent destination can be saturated by attackers with a modest understanding of current network routes. That is, the target is not always apparent from the measurement traffic, and without a sophisticated understanding of network topology and routing, no centralized controller is in a position to prevent attackers from concentrating traffic. We expect this to be an area of further research as we gain experience.

3 System Design

In this section, we describe the components of the Scriptroute system, how these components communicate, and how a user submits a script for execution.

The set of cooperating components is shown in Figure 1. We separate these components for robustness and security: each performs a simple task, and compromising one does not help compromise others. The task of the front end Web server is to pass measurement scripts uploaded from clients to the interpreter for execution. The interpreter runs in a restricted environment and may fail by exceeding resource limits or by measurement script error. The interpreter's use of the Scriptroute API is carried by a local TCP socket to the network guardian. By separating the interpreter into its own process, it can fail without affecting the network guardian or front end. The network guardian is the only component that needs to be run with special permissions to read and write raw packets.

Each component also has a role in providing security, summarized in Figure 2. The front-end verifies that scripts are submitted from unforged IP addresses (via TCP handshaking) and prevents scripts from running too long or sending too much output. The interpreter provides flexibility in choosing what sort of probe packets to send and when, but restricts execution to a resource-limited sandbox. The practice of combining a sandbox based on a safe language with a narrow interface is well established [2, 18, 23]. Finally, the network guardian enforces rate limits and packet filtering policy, and only permits responses to probes to be returned to the measurement script. The local administrator controls the resource limits and filtering policy.

We now describe the design of each of these components in the order visited by an executing measurement.

3.1 System Management

Scriptroute servers publish their existence in a dynamically updated DNS database. This allows clients to find Scriptroute servers using descriptive host names,

Component:		Front-End	Interpreter	Network Guardian
Program:		`thttpd` [48]	`srrubycgi`	`scriptrouted`
Service:		Remote access via HTTP	Flexible scripted execution	Raw network access
Main security role:		Sanitize script input	Protect host from scripts	Protect network from traffic
Protection Features – Host	Integrity	Provides empty chroot	Interpreter safe mode [b] Runs as user "nobody" Limit: Processes, Files [c]	
Protection Features – Host	Resources	Limit script: [a] Length, Runtime, Output	Limit: Memory, CPU time	Rate limit overall traffic Limit running scripts
Protection Features – Network	Integrity	TCP handshake verifies client IP address		Log all packets by client Filter dangerous packets
Protection Features – Network	Resources			Rate limit SYN packets Rate limit by destination

[a] `thttpd` provides these features by default, otherwise these limits would have to be enforced by the interpreter.
[b] Sandboxing scripts is not necessary when running an interpreter as a local user.
[c] Provides redundant protection to reinforce the safe mode against fork() and open().

Figure 2: Scriptroute server components, annotated with their security roles and features that provide host and network security.

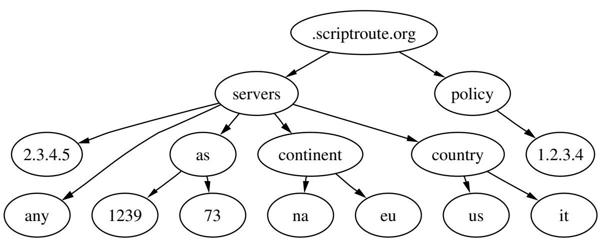

Figure 3: Scriptroute DNS name space.

and servers to publish their feature set (e.g., software version.)[1] Different Scriptroute servers may belong to different groups and use different DNS servers; ours is rooted at `scriptroute.org`. As shown in Figure 3, the name-space is separated into two subtrees: `policy` and `servers`.

The servers subtree returns pseudo-random lists of Scriptroute servers, optionally chosen by AS, country, or continent. This breakdown was chosen for convenience, but the complete database can be accessed from a dynamically generated Web page.

The policy subtree includes entries for measurement targets that wish to block unwanted measurement traffic. The goal of this repository is to restrict traffic from compliant Scriptroute servers in a single step. There are two ways to update this database. Individual targets can connect to a Web server and block measurement traffic back to their own IP address. Alternately email from a domain administrator is used for blocking traffic to entire IP prefixes. The Web interface provides a timely update when it is clear, by the TCP handshake, that a user of the target machine has requested a filter; changes are immediately propagated into the DNS policy subtree. The email-based interface deals with many hosts in the same administrative domain, but requires human verification before coarser filters are installed or removed.

[1] In contrast, traceroute servers are found using directories maintained by hand [32] and research testbeds have a static host list.

3.2 Server Front-End

Each Scriptroute server runs an ordinary Web server on port 3355, which provides a gateway for script submission and administrative tasks. There are three main "pages" on the server: job submission, traceback, and informational.

The job submission page provides an HTTP POST interface for measurement script submission, then replies with the output of the measurement. Again, the TCP handshake demonstrates that the source IP address is valid to provide a measure of accountability. A convenient feature of thttpd [48] is that it limits the execution time, size, and output of the script. We also limit the number of concurrent requests per client (1) and the number of concurrent requests overall (10). If the interpreter fails due to resource limits, the connection is closed signaling an error to the client. Unhandled exceptions in the measurement script itself are handled by the interpreter and returned to the client as text.

The traceback page provides limited access to the logs to reduce anonymity and prevent Scriptroute from "laun-

```
#! /usr/local/bin/srinterpreter

probe = Scriptroute::Udp.new(12)
probe.ip_dst = ARGV[0]
unreach = false
puts "Traceroute to #{ARGV[0]} (#{probe.ip_dst})"

catch (:unreachable) do
  ( 1..64 ).each { |ttl|
    ( 1..3 ).each { |rep|
      probe.ip_ttl = ttl
      packets = Scriptroute::send_train([ Struct::DelayedPacket.new(0,probe) ])
      response = (packets[0].response) ?  packets[0].response.packet : nil
      if(response) then
        puts '%d %s %5.3f ms' % [ ttl, response.ip_src, (packets[0].rtt * 1000.0) ]
        if(response.is_a?(Scriptroute::Icmp)) then
          unreach = true if(response.icmp_type == Scriptroute::ICMP_UNREACH)
        end
      else
        puts ttl.to_s + ' *'
      end
      $stdout.flush
    }
    throw :unreachable if(unreach)
  }
end
```

Figure 4: Traceroute, as implemented in Ruby for Scriptroute. For comparison, a stripped down version of traceroute [14] is implemented in 200 lines of C.

dering" traffic. Specifically, it provides the tcpdump-formatted packets sent to particular IP addresses along with the address of the corresponding client.

Finally, the informational page provides information about the measurement traffic supported, how to contact the administrator of the server, how to learn more about Scriptroute, and how to add destination filters to block unwanted measurement traffic. So that administrators know where to look to when their systems receive unexpected measurement traffic, we encourage Scriptroute servers that also have a port 80 Web server to link this page, to direct concerns to the central management site.

3.3 Script Interpreter

The front end pipes submitted jobs to a scripting language interpreter in a new process. In our implementation, we chose Ruby, but any language that supports a strong sandbox can be used. The interpreter runs as a separate process so that it can fail independently: aggressive kernel resource limits are used to prevent significant resource consumption; when exceeded, the process terminates abruptly.

The interpreter provides access to the Scriptroute API and a simplified interface to packet contents, taking care of such details as network byte ordering. The measurement script can instantiate new packets, fill them in, then send them via the Send-train API call, which the interpreter translates into a socket connection to the network guardian. An example script implementing traceroute is shown in Figure 4.

The interpreter communicates to the network guardian using only the Send-train API. Send-train supports most network measurements by sending a train of probe packets and collecting their responses. The Send-train operation takes an array of (delay, probe packet) pairs as an argument, then returns an array of (time-stamp, probe packet, time-stamp, response packet) tuples. The observation is that most measurements send a train of probes (possibly just one) then wait for the responses and repeat.

3.4 Network Guardian

The network guardian is responsible for limiting the rate of measurement traffic and regulating the type of packets sent. It combines destination-specific filters to block traffic as stored in DNS with the rate limits and additional filters configured by the local administrator.

To support the Send-train API, the network guardian is responsible for matching probes with their responses, which protects the host from measurement tools that might otherwise see unrelated traffic. Matching responses to probes is simple in the case of traceroute-like

UDP probes and ICMP error responses (which match the encapsulated header), ICMP echo request/response (which match the sequence number), and unsolicited TCP probes with TCP RST responses (which match the address, port, and sequence number). It is more complex for TCP connections, where we match responses to the earliest plausible probe.[2]

The network guardian mediates access to the raw sockets and packet capture facilities of the kernel, so must be run "as root" or with special configuration. Finally, the network guardian logs sent and received packets with the client that requested the corresponding measurement. These logs can be used after the fact to infer what sort of traffic might have offended a remote site. We describe the policies enforced by the network guardian in detail in the next section.

4 Implementation

In this section, we describe the implementation of the interpreter and network guardian. We describe the default policy configuration that protects the network and destination hosts. The network guardian consists of 3,000 lines of C, and the interpreter adds another 600, calling on Ruby and tcpdump as libraries.

The system management interface is a combination of a Web server (thttpd), a DNS server (tinydns), and a small daemon that updates the zone file based on registration messages sent by servers and destination filters submitted by Web and email. Implementation details of this component are straightforward and not described further.

4.1 Script Interpreter

The interpreter provides an environment to support measurement scripts and hand packet trains to the network guardian. It creates a sandbox with a name space that includes the Scriptroute API and class definitions for standard packet types.

The class-based packet interface simplifies development by attending to details such as network byte ordering and host name lookup. The packet class's `to_s` (to string) method uses code from tcpdump to present a familiar representation of the packet for debugging.

[2]To establish a TCP connection in a measurement requires that the host kernel not see the SYN/ACK and respond immediately with a RST. This can be prevented using safe raw sockets [47] or the kernel's firewall [50].

The interpreter uses the kernel to limit the script's resource consumption in processor time (4 second default) and memory footprint (50K stack, 50K data, 8MB address space, though these limits depend on the operating system). Each of these limits is configurable by the local administrator. Additional resource limits on concurrently opened file handles (7) and processes (1) are used to reinforce the interpreter's safe mode against inadvertent calls to `open()` or `fork()`. Scripts that exceed these limits are abruptly terminated, which is why each script executes in its own interpreter process.

Resource limits on individual processes must be combined with a limit on the number of concurrent measurement scripts. A new interpreter requests permission to execute from the network guardian, and may be told to try again later if there are too many scripts in the system or too many scripts being executed on behalf of the same user (the default limits are one per user to a maximum of ten per system). A user is defined by the client IP address if accessed through the front end, or by the user name of the process if executed locally.

The *chroot* environment created by the front-end is inherited by the interpreter. A chroot-ed process executes with all file accesses confined to sub-tree of the file system. While not designed for sandboxing processes, it can be used to isolate processes from from the rest of the machine, in this case preventing the interpreter from accessing any files in the system. We make the chroot robust to common attacks by both running the interpreter as "nobody," which lacks permission to write the filesystem, and keeping the chroot empty; it contains only the statically-linked interpreter and the sent packet logfile.

We chose Ruby because it is a lightweight, type-safe, general-purpose interpreted language with a safe mode that guards access to system calls. While most of these features are just convenient, a flexible safe mode is essential. For example, Ruby's safe mode prevents files and sockets from being opened, but permits the script to write its results back to the client over an already existing socket. We believed that a scripting language would make development simple, which was an important consideration given that many existing tools would need to be ported. We believed that choosing a general-purpose language was important for encouraging adoption: those who already know Ruby should find it trivial to write measurement scripts, and those who are new to the language can apply their new experience to ordinary tasks. Finally, we found that the Ruby interpreter integrated well with C, which was important because the isolation enforced by the safe mode prevents the script from accessing the network guardian directly.

Local Socket (Send-train API)

⇓ ⇑

Well-formedness Filter	
Source Policy Filter	
Destination Policy Filter	
Sanitizer	Sent & Recv'd Packet Log
Rate Limiter	Expectations
Raw socket	libpcap

⇓ ⇑

Operating system network stack

Figure 5: Architecture of the network guardian. At left is the downward path probes take out to the network; right is the upward path packets take back. On Planetlab, the raw socket and libpcap interfaces are replaced by safe raw sockets [47].

4.2 Network Guardian

The network guardian is responsible for protecting the network and destination hosts by applying policy checks before traffic is sent. It is the only component that requires special privilege to read and write a raw socket. It provides this packet generation service using the Send-train API to interpreters (or any other process) on the local machine. The architecture of the network guardian is shown in Figure 5. We describe the components in the order they are visited by a measurement.

The network guardian accepts TCP socket connections on the localhost address from the interpreter. Listening only to localhost allows the network guardian to operate on behalf of local processes without providing remote service, adding a small measure of security. The interface across this socket is text-based for extensibility and ease of debugging. However, binary packets must be encoded to be transferred across a text-based interface; we chose base64 encoding, a method commonly used for encoding MIME attachments.

Packets face a series of verification steps. First, they are checked for integrity and that the reflector can recognize likely responses to the probe. For example, this verifies the packet has sufficient length for its headers and is of a known protocol.

Second, the source's filter is applied. The administrator of the Scriptroute server has discretion over what traffic should be generated, and can decide what packets can be sent. The default source filters remove broadcast and multicast packets, IP fragments, ICMP error messages, TCP resets, UDP and TCP traffic to "priviledged" ports (those below 1024) other than 80 (HTTP) and 53 (DNS), and traffic to the local host and subnet.

Bucket	Recharge rate	Burst size
Measurement SYN	1 packet/s	4 packets
Destination I	1 Kbytes/s	8 Kbytes
Destination II	3 packet/s	10 packets
Source	3 Kbytes/s	100 Kbytes

Table 3: Default rate limit parameters: each can be adjusted by the local administrator. The source and destination rate limits are shared by all measurements, while the SYN limit applies to each measurement.

Third, the destination policy is applied. The network guardian executes a lookup on the destination address in the policy subtree of the DNS described in Section 3.1. A filter may be stored (as a TXT record and in BPF [38] format) under the destination's address. If no entry exists for that destination, no additional filters are applied. The filter is cached for five minutes, but if the DNS server is unreachable, the previously cached entry is used.

Fourth, packets are "sanitized" by setting the source address to that of the local machine and setting the source port, if UDP or TCP, to one owned by the network guardian. This prevents harmful interactions with other traffic on the same machine and provides accountability by avoiding source spoofing. The packet is then checksummed.

As a final step, the probes are scheduled to be sent by passing them through a series of rate-limiting token buckets. The default burst size (bucket depth) and recharge rate parameters of these buckets are shown in Table 3. If the packet is a TCP SYN, it is passed through a per-experiment rate-limiter that is intended to prevent SYN flooding attacks. Next, the packet passes through per-destination limiting to prevent flooding attacks. The first per-destination limit is on the rate of traffic in bytes to prevent bandwidth-consuming flooding attacks. The second limit is on the rate of packets sent, because a packet represents some overhead at the destination, possibly involving application-layer processing. The final rate limiter prevents excessive bandwidth consumption at the source.

When probes are sent, "expectation" state is created, representing the set of possible responses to associate with the probe. For example, sending an ICMP echo request creates the expectation of either an ICMP echo response or an ICMP error message. These expectations filter the packets read from libpcap – preventing unrelated traffic from escaping to measurement scripts – and match responses with probes, simplifying tool development.

Matched probes and responses with their timestamps, or sent probes that received no response after a timeout period, constitute the response to the Send-train API. The reflector logs each probe/response pair before returning it to the interpreter, ending with the status message "done."

5 Evaluation

Applications are the key to evaluating Scriptroute. That is, the most important evaluation questions are: what new measurements does Scriptroute enable, how readily can they be expressed, and how efficiently are they run? To begin to answer these questions, we used Scriptroute for two case studies. First, we use Scriptroute to implement a new debugging tool, "reverse path tree" (RPT), that gathers and summarizes network routes towards a target. Second, we use Scriptroute to gather a dataset suitable for assessing the merits of Global Network Positioning (GNP), a newly proposed Internet distance prediction technique. Both of these case studies were undertaken primarily for the purpose of evaluating the capabilities of Scriptroute. At the same time, both represent real tasks that could not be accomplished without access to many measurement vantage points.

5.1 Reverse Path Tree (RPT)

By "reverse path tree" we mean the tree of routes that are used to reach a specific host from other locations on the Internet, as opposed to paths from that host outwards to other locations that are provided by regular traceroute. The reverse path tree summarizes how a host is reached from the rest of the Internet, and it can only be generated with the help of remote hosts. It generalizes the practice of ISPs manually using a remote traceroute server to check connectivity and routing to themselves.

The Scriptroute-based RPT discovery tool proceeds in two logical steps: tracing the routes from as many servers as possible to the destination; and merging them with IP alias resolution to recognize interface IP addresses that belong to the same router [20, 53]. Scriptroute provides the opportunity to reduce the amount of traffic needed to construct the tree by recognizing segments that have already been traversed on-line. In contrast, assembling a tree from standard traceroutes would probe routers close to the destination repeatedly. We do this by embedding a list of previously observed IP addresses in the measurement script, having the script terminate when it reaches a part of the tree that has already been mapped, and mapping from different servers

Component	Code length
Traceroute (shipped)	33 lines
Alias res. (interpreted)	137 lines
Alias res. (client)	50 lines
Tree analysis (client)	148 lines
Overall	318 lines

Phase	Execution
Traceroute	195 packets / 1min 20sec
Alias resolution	102 packets / 3min 36sec

Table 4: Reverse path tree (RPT) code and runtime statistics. Components are "shipped" to remote Scriptroute servers, "interpreted" by the local Scriptroute server, or executed as part of the "client's" analysis

sequentially. This reduction allows the system to scale without loading the network. We also note that alias resolution is run on the local Scriptroute daemon. It does not need to be distributed because it measures endpoint rather than path properties.

A sample tree mapped by RPT to one of our hosts is shown in Figures 6 and 7. Already we can see that Scriptroute deployment on PlanetLab provides a rich enough set of servers to construct a useful tree. Code size and runtime statistics for the RPT tool are given in Table 4. Code size shows the number of lines of code at the client and shipped to Scriptroute servers. The number of packets includes only measurement traffic, and the execution time for each phase is given. These phases could be overlapped, but performance is already adequate for the task. We can see that both client and server code is small; the choice of a scripting language for constructing tools appears worthwhile. Further, Scriptroute supports a useful measurement task despite the rate limits imposed for security.

We expect RPT to serve as a foundation for future tools to infer the location of performance problems by observing which parts of the tree are shared between Scriptroute servers. For example, Scriptroute could be used to measure loss between each server and a destination, as well as trace the tree. Techniques such as [35, 42] could then be used to pinpoint lossy segments.

5.2 Validating GNP

To demonstrate the value of Scriptroute for network measurement research, we undertook to validate claims for Global Network Positioning (GNP) [40], a recently proposed technique for estimating Internet latency between points. GNP estimates latency using multi-dimensional mappings derived from measurements between each point and special landmarks. The details

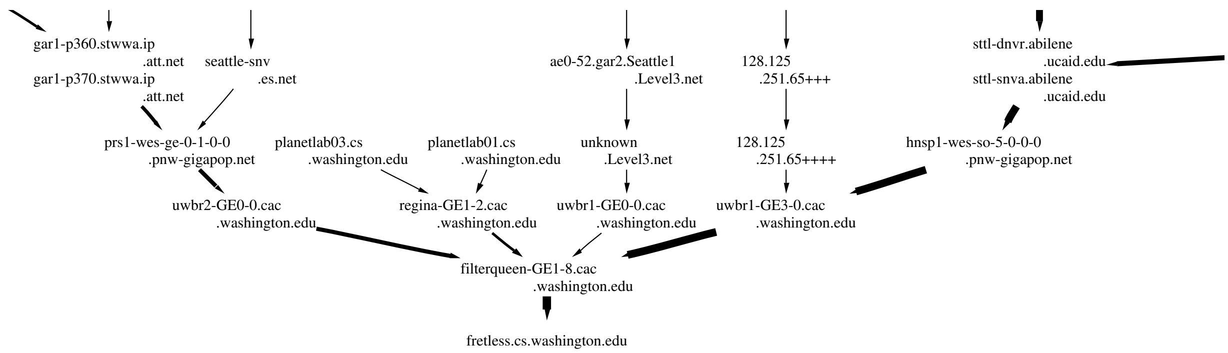

Figure 6: An excerpt from a reverse path tree measured by Scriptroute. Line thickness represents the number of paths that traverse the link. Aliases are listed together. Most of the tree is to the right and above: this is the neighborhood of the root. Those IP addresses listed with '+' are unresponsive successors of an IP address.

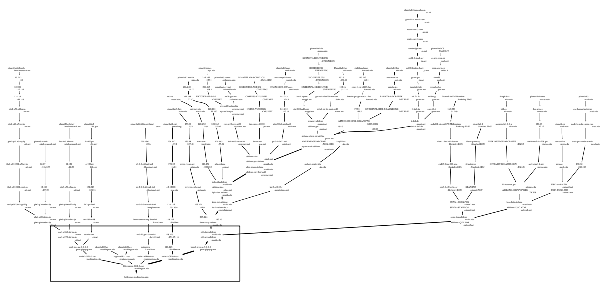

Figure 7: The complete reverse path tree, of which Figure 6 is an excerpt. This overall view shows the detail available by combining many vantage points.

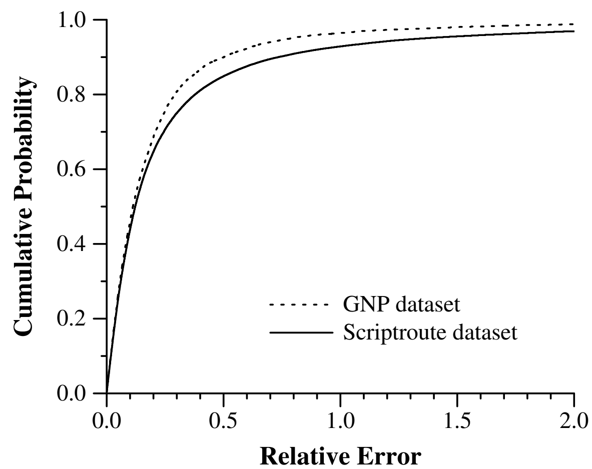

Figure 8: Relative error distribution of GNP.

of GNP itself are unimportant for this paper; our aim is simply to demonstrate the utility and scalability of Scriptroute by repeating a real experiment. For this analysis, we require a dataset consisting of measured latencies between Scriptroute servers and many Internet hosts. These measurements can then be compared against GNP-derived estimates. The GNP study required the authors to obtain accounts on 19 machines distributed around the globe – we would like to make this sort of measurement study nearly trivial.

As input to the GNP analysis tool, we gathered a set of latency measurements from 31 Scriptroute servers as vantage points and roughly 3200 other Internet hosts from a previously selected database. This is actually a considerably larger dataset than that used in [40]. Each Scriptroute script pinged a random selection of ten hosts at a time and returned the minimum round trip latency to the client. Each host was pinged 15 times, rather than 220 as in [40].

We used these latency measurements as a dataset to evaluate the accuracy of GNP estimates. Fifteen Scriptroute servers are designated as landmarks, and we use the GNP analysis tool to compare GNP estimates to measured latencies between the non-landmark Scriptroute servers and the hosts. In Figure 8, we plot both our results for the cumulative distribution of relative error (as defined in [40]) and the results from the data set used in [40]. We find a slightly higher relative error, but on the whole the results are comparable, despite our lack of tuning.

This experiment highlights the capabilities of Scriptroute as a tool for gathering network performance data. The code size for the client and server scripts is given

Component	Code Length
Ping sweep (shipped)	33 lines
Analysis (client)	55 lines
Ping packets	60K per server 1.8M overall
Execution time	2 hours 30 minutes (in parallel)

Table 5: GNP dataset collector statistics.

in Table 5, along with the run time and packet counts as before. We see that the experiment scripts again are very small and run relatively quickly. Similar datasets could be gathered for other experiments, such as checking the latency savings of Detour paths in RON [3, 51].

6 Related Work

We describe existing distributed network measurement and debugging systems classified by whether they support unauthenticated clients, as this is a key feature of our design. We then describe safe local interfaces for network measurement that share attributes of the Scriptroute software architecture.

6.1 Unauthenticated Systems

Unauthenticated systems are often provided to aid in network debugging. Such debugging infrastructure includes public looking-glass servers, which show BGP configuration, and public traceroute servers, which show the path to an arbitrary destination. They are widely deployed: in the Rocketfuel project, traceroute servers represented over 700 vantage points in the network [53]. Such servers are inflexible: only a few measurements are supported, optimizations such as those we used in Section 5.1 are unavailable, and modifications to use less-filtered protocols [39, 56] or different logic are impossible. These servers are often tedious to use cooperatively: they may come and go faster than Web directories can be updated, and often use distinct interfaces. Scriptroute was designed to address these problems while building on the successes of these unauthenticated systems.

6.2 Authenticated Systems

Systems for network research, including Netbed (formerly emulab) [57], NIMI [45], Surveyor [30], IPMA [24], AMP [1], and RON [3]. From our perspective, these systems are similar, so we describe the most established one, NIMI. The National Internet Measurement Infrastructure (NIMI) is a research platform for distributed network measurement. Their design focus was on scalability and security, and a goal of their project was to support standardized network metrics from the IETF's IPPM working group [46].

NIMI, and the Network Probe Daemon upon which it is based, have similar goals as Scriptroute but different approaches. The NIMI approach to security is one of a closed system of trusted users who authenticate themselves, communicate using an encrypted protocol, and run standardized measurement tools. The Scriptroute approach, in contrast, is to permit any user to connect and run arbitrary measurement scripts, so long as the generated traffic conforms to a model of safe traffic.

The most significant advantage that authenticated systems have is that users are assumed to be friendly, which simplifies resource allocation. As an example, storage resources can be allocated to users, allowing measurements to be scheduled and their results stored until the user returns to claim them.

6.3 Extensible Network Measurement

Safe interfaces for network measurement have generated recent interest. The FLAME project provides a system for passive monitoring of network traffic, using a type-safe language (Cyclone [28]) and run-time verification [2]. FLAME provides extensibility to the monitoring facilities offered by routers, installing code into the operating system kernel.

Two projects support active measurements on a single system using a similar API. The PeriScope [22] project provides a kernel API to send groups of ICMP echo requests without returning to user space, which they argue helps accuracy. Pásztor and Veitch [43] also separate measurement logic from sending probe packets in different processes, but they do so for precisely scheduled packet transmission using a real time task in RTLinux. Scriptroute complements these systems by providing a layer between scripts and the kernel that can be extended to support these richer interfaces. Scriptroute currently supports raw sockets with libpcap by default, and Scout's safe raw sockets on Planetlab, allowing measurement scripts to transparently take advantage of new host operating system features.

7 Conclusions and Future Work

We have presented the design and implementation of Scriptroute, a new platform that allows ordinary Internet users to make network measurements from remote vantage points. Scriptroute is motivated by the popularity and utility of public traceroute servers. Clients locate servers using the DNS and ship measurement tasks as scripts. This provides the flexibility to implement a variety of non-intrusive tools for measuring path properties and makes it easy to coordinate measurements across servers. To protect servers from abuse, measurement scripts are executed in a resource-limited sandbox controlled by the local administrator. To prevent the system from being used to launch denial-of-service attacks, measurement traffic is checked, rate-limited, and logged for accountability.

The Scriptroute software is publicly available [52], including clients and sample measurement scripts, as well as the server and interpreter source. We have deployed servers across the PlanetLab testbed of 42 sites. We have used the resulting system to measure routing trees around a destination and to collect a latency dataset suitable for evaluating Internet distance prediction techniques. Our early experience suggests that the system is quite flexible and useful, despite its security restrictions, and that scripting is an apt choice for expressing and combining measurement tasks.

We view Scriptroute as a work in progress. We believe that Scriptroute shows how a public infrastructure can substantially improve our ability to make safe, flexible network measurements. With experience, we hope to improve the system and better assess our design choices. Some interesting features are not yet implemented, including support for measurements using TCP connections and tools that send responses rather than probes. We also expect our security policies to evolve as we uncover patterns of preferred usage and attempted abuse, and as our model of safe network measurement traffic is broadened with the advent of new tools.

Acknowledgements

We wish to thank Intel Research for providing access to their PlanetLab resources. Mike Wawrzoniak and Andy Bavier made it possible for Scriptroute to run using PlanetLab's safe raw sockets. Vern Paxson provided helpful comments. We also thank David Richardson, Brent Chun, and Ratul Mahajan.

This work was supported by DARPA under grant no. F30602-00-2-0565.

References

[1] Active Measurement Project. `http://amp.nlanr.net/`.

[2] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, and M. Greenwald. Open packet monitoring on FLAME: Safety, performance and applications. In *IFIP Int'l Working Conference on Active Networks (IWAN)*, 2002.

[3] D. Anderson, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, 2002.

[4] J. Bellardo and S. Savage. Measuring packet reordering. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.

[5] R. L. Carter and M. E. Crovella. Measuring bottleneck link speed in packet-switched networks. Tech. Rep. TR-96-006, Boston University CS Dept., 1996.

[6] R. L. Carter and M. E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. In *IEEE INFOCOM*, 1997.

[7] CERT. TCP SYN flooding and IP spoofing attacks. `http://www.cert.org/advisories/CA-1996-21.html`, 1996.

[8] CERT. UDP port denial of service attack. `http://www.cert.org/advisories/CA-1996-01.html`, 1996.

[9] CERT. IP denial of service attacks. `http://www.cert.org/advisories/CA-1997-28.html`, 1997.

[10] CERT. Smurf IP denial of service attacks. `http://www.cert.org/advisories/CA-1998-01.html`, 1998.

[11] Cooperative Association for Internet Data Analysis (CAIDA). Internet tools taxonomy. `http://www.caida.org/tools/taxonomy/`, 2002.

[12] Cowzilla and P. Dreamer. Puke. `http://www.cotse.com/sw/dos/icmp/puke.c`, 1996.

[13] C. Dovrolis, P. Ramanathan, and D. Moore. What do packet dispersion techniques measure? In *IEEE INFOCOM*, 2001.

[14] A. B. Downey. trout. `http://rocky.wellesley.edu/downey/trout/`, 1999.

[15] A. B. Downey. Using pathchar to estimate Internet link characteristics. In *ACM SIGCOMM*, 1999.

[16] Fyodor. NMAP: The network mapper. `http://www.insecure.org/nmap/`.

[17] E. Gavron. NANOG traceroute. `ftp://ftp.login.com/pub/software/traceroute/beta/`.

[18] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *USENIX Security Symposium*, 1996.

[19] R. Govindan and V. Paxson. Estimating router ICMP generation delays. In *Passive & Active Measurement (PAM)*, 2002.

[20] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In *IEEE INFOCOM*, 2000.

[21] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.

[22] K. Harfoush, A. Bestavros, and J. Byers. PeriScope: An active measurement API. In *Passive & Active Measurement (PAM)*, 2002.

[23] C. Hawblitzel, *et al.* Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, 1998.

[24] Internet Performance Measurement and Analysis (IPMA) project. `http://www.merit.edu/ipma/`, 2002.

[25] V. Jacobson. Pathchar. `ftp://ftp.ee.lbl.gov/pathchar`.
[26] V. Jacobson. Traceroute. `ftp://ftp.ee.lbl.gov/traceroute.tar.Z`.
[27] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. In *ACM SIGCOMM*, 2002.
[28] T. Jim, *et al.* Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.
[29] R. Jones. Netperf. `http://www.netperf.org/`.
[30] S. Kalidindi and M. J. Zekauskas. Surveyor: An infrastructure for Internet performance measurements. In *INET'99*, 1999.
[31] M. Kenney, ed. Ping o' death. `http://www.insecure.org/sploits/ping-o-death.html`, 1996.
[32] T. Kernen. traceroute.org. `http://www.traceroute.org`.
[33] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *USITS*, 2001.
[34] B. Mah. Estimating bandwidth and other network properties. In *Internet Statistics and Metrics Analysis Workshop*, 2000.
[35] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link weights using end-to-end measurements. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.
[36] M. Mathis. Windowed ping: an IP layer performance diagnostic. In *INET'94/JENC5*, 1994.
[37] M. Mathis. Diagnosing Internet congestion with a transport layer performance tool. In *INET'96*, 1996.
[38] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter Technical Conference*, 1993.
[39] N. McCarthy. fft. `http://www.mainnerve.com/fft/`.
[40] T. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *IEEE INFOCOM*, 2002.
[41] J. Padhye and S. Floyd. Identifying the TCP behavior of Web servers. In *ACM SIGCOMM*, 2001.
[42] V. N. Padmanabhan, L. Qiu, and H. J. Wang. Passive network tomography using bayesian inference. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.
[43] A. Pásztor and D. Veitch. A precision infrastructure for active probing. In *Passive & Active Measurement (PAM)*, 2001.
[44] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *ACM Computer Communication Review*, 2001.
[45] V. Paxson, A. Adams, and M. Mathis. Experiences with NIMI. In *Passive & Active Measurement (PAM)*, 2000.
[46] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. RFC 2330, 1998.
[47] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *HotNets-I*, 2002.
[48] J. Poskanzer. thttpd. `http://www.acme.com/software/thttpd/`.
[49] S. Saroiu, P. K. Gummadi, and S. D. Gribble. Sprobe: A fast technique for measuring bottleneck bandwidth in uncooperative environments. In *Submitted for publication*, 2002. `http://sprobe.cs.washington.edu/sprobe.ps`.
[50] S. Savage. Sting: a TCP-based network measurement tool. In *USITS*, 1999.
[51] S. Savage, *et al.* The end-to-end effects of Internet path selection. In *ACM SIGCOMM*, 1999.
[52] Scriptroute. `http://www.scriptroute.org/`.
[53] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *ACM SIGCOMM*, 2002.
[54] H. Tangmunarunkit, R. Govindan, D. Estrin, and S. Shenker. The impact of routing policy on Internet paths. In *IEEE INFOCOM*, 2001.
[55] A. Tirumala, F. Qin, J. Dugan, and J. Ferguson. Iperf. `http://dast.nlanr.net/Projects/Iperf/`, 2002.
[56] M. C. Toren. tcptraceroute. `http://michael.toren.net/code/tcptraceroute/`.
[57] B. White, *et al.* An integrated experimental environment for distributed systems and network. In *OSDI*, 2002.