An Integrated Active Measurement Programming Environment

Matthew Luckie[1], Shivani Hariprasad[1], Raffaele Sommese[2], Brendon Jones[1], Ken Keys[1], Ricky Mok[1], and K Claffy[1]

CAIDA, UC San Diego, USA
{mjl,shari,brendonj,kkeys,cskpmok,kc}@caida.org
University of Twente, Netherlands
r.sommese@utwente.nl

Abstract. Active Internet measurement is not a zero-risk activity, and access to Internet measurement vantage points typically requires navigating trust relationships among actors involved in deploying, operating, and using the infrastructure. Operators of vantage points (VPs) must balance VP capability against who gets access: the more capable a vantage point, the riskier it is to allow access. We propose an integrated active measurement programming environment that: (1) allows a platform operator to specify the measurements that a user can run, which allows the platform operator to communicate to the VP's host what their vantage point will do, and (2) provides users with reference implementations of measurement functions that act as building blocks to more complex measurements. We first review active measurement infrastructures and how technical and usability goals have evolved over the years. We prototype and deploy an integrated active measurement programming environment on an existing measurement infrastructure, and illustrate its potential with several case studies.

1 Introduction

Network operators and researchers often require the ability to conduct active measurements of networks from a specific location in order to understand some property of the network. However, access to vantage points requires navigating trust relationships among three types of actors involved in deploying, operating, and using the infrastructure: (1) the hosting site that deploys a vantage point (VP) in their network, (2) the platform operator that maintains the VPs in the field, and (3) the researchers that use the platform. The hosting site incurs risk in hosting a VP, and has to trust that the platform operator will use the VP in ways that do not harm the hosting network. The platform operator incurs significant risk when they allow researcher access to the platform. These trust requirements inhibit deployment of active measurement infrastructure, impeding progress in the field of Internet measurement.

Figure 1 illustrates a spectrum of access models for active measurement infrastructure, ordered from least to most restrictive. The least restrictive solutions

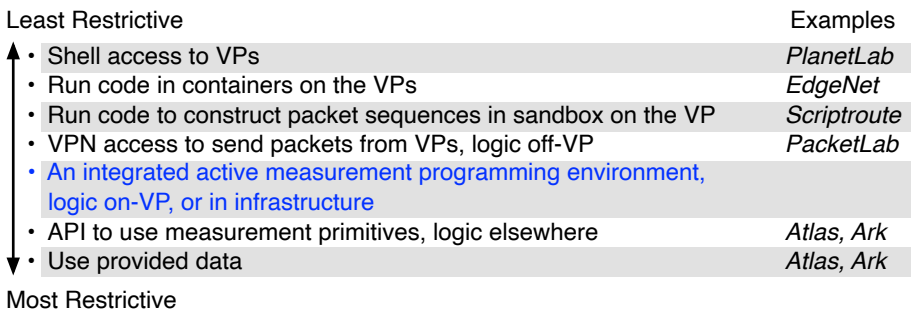

Fig. 1. Spectrum of active measurement infrastructures.

grant researcher access directly to the VP, either bare-metal, or within a container. The platform operator can restrict access with process and capability limits, but has little other control over what the researcher does, and thus assumes significant risk. A step removed from this is VPN-like access: the VP acts as a simple packet forwarder, allowing a researcher to use the node without providing shell access. These solutions allow researchers to craft specific packet sequences that allow for inference based on how the receiver reacts.

More restrictive solutions do not allow access to the VPs, or do not allow researchers to construct their own packet sequences. The most restrictive solutions provide raw data, which relies on the platform operator knowing the needs of the measurement community *a priori*, or provide access to a restricted set of tests via an API. The utility of the platform hinges on the usefulness of the data, the provided tests, and responsiveness of the API.

This paper proposes, implements, and deploys a solution that lies in the middle of the spectrum. Our contribution is to provide a Python-based integrated active measurement programming environment that exposes both a set of distributed VPs, and a set of useful measurement primitives from which to build sophisticated measurement tools. The key benefits to *researchers* are that (1) the environment can provide reference implementations of measurement primitives that are difficult to implement correctly, making the environment useful especially for novice programmers, (2) the environment allows researchers to focus on the logic that ties a series of measurements together in an experiment, and (3) the logic is close to the VP, reducing experiment latency.

The key benefit to a *site host* is that the environment makes it difficult for a researcher to cause harm, intentionally or not, as researchers are restricted to the available measurements. The environment allows the *platform operator* to describe to the hosting site how researchers can use their VPs. However, researchers rely on the environment maintainers and platform operators to expose useful measurement primitives and to keep the environment current with modern systems and evolving Internet protocols.

After a review of related measurement platforms, we articulate our design goals, describe our architecture and implementation, and demonstrate its potential using several case studies.

2 Researcher-oriented client-side measurement platforms

Several early (now defunct) active measurement platforms such as Skitter [12], Surveyor [14], AMP [22], NIMI [25, 26], and DIMES [31] provided (primarily simple traceroute topology) data for use by the research community. We do not discuss M-Lab because it provides server-side facilities for client-server active measurements [10], or NLNOG RING because the infrastructure requires a user be an operator at an AS with a participating VP [24].

PlanetLab: In 2002, Peterson *et al.* began deploying PlanetLab, a platform for deploying and managing distributed network services [27]. PlanetLab operators distributed *customized* Linux-based hardware systems to research and education organizations. The customizations included (1) virtual *slices* isolated from other slices running on the same system, (2) the ability to use socket APIs that typically required root privileges, and (3) management software. The measurement community made extensive use of PlanetLab. At its peak, PlanetLab had systems in ≈700 organizations. PlanetLab shut down in 2020.

Scriptroute: Released in 2003, Scriptroute [32] provided (1) a set of distributed VPs, and (2) a sandboxed scripting environment so that unvetted users could use them. An application programmer wrote Ruby scripts that embedded logic for sending packets and processing received packets. Users found VPs with DNS queries, and uploaded scripts to VPs of interest via an HTTP API, requiring that each VP have a publicly reachable IP address. Each VP's Scriptroute instance protected the hosting site from accidental or malicious transgressions by running user scripts in distinct sandboxes that limited the resources and system capabilities available to each script, enforced policy around the types and frequency of packets that each script could send, and matched probes with responses so that each script could only observe responses to packets it sent.

Ark: In 2007, CAIDA began operating the Ark infrastructure to perform comprehensive global topology mapping as well as support third-party experiments on the platform. As of October 2024, the infrastructure consists of ≈170 VPs distributed in 57 countries across 133 ASes. The infrastructure consists of x86 rack-mount systems, Raspberry Pis (versions 2-4), as well as VMs and containers. To coordinate measurements between VPs, CAIDA implemented (in Ruby) a distributed tuple-space named Marinda. CAIDA made extensive use of Marinda for its own global measurements and research, but no external researcher published a paper where they had used Marinda to coordinate measurements. Researchers could deploy vetted measurement software on the nodes, but deployment was cumbersome because Ark had a mix of operating systems, both vendors and vintages, and a mix of CPU architectures.

RIPE Atlas: operated by RIPE NCC since 2010, Atlas is currently the largest deployed operational active measurement infrastructure, with 12,111 VPs in 3,649 IPv4 (1,844 IPv6) ASes as of October 2024, representing 4-5% of routed ASes [30]. Atlas consists of different types of VPs. The majority (7,697) are small single-board computers with limited CPU, storage, and memory. Atlas also consists of more-powerful *anchors* (794), as well as software VPs (3,620) using the same software as deployed on the single-board computers. Factors in

Atlas' success include (1) the VPs were cheap to produce, (2) RIPE restricts the types of measurements conducted on the VPs to mitigate risk to site hosts, (3) these primitives provide useful building blocks, (4) RIPE incentivizes VP deployment by providing credits to site hosts that enables site hosts to conduct measurements from other Atlas VPs, and (5) RIPE subsidizes Atlas through RIR fees. Atlas exposes simple measurement primitives through their web-based API that allows users to conduct ping, traceroute, and selected DNS and NTP queries. Users schedule measurements through the API, and then fetch the results when they become available. To accomplish a complex measurement, the user must parse the raw data, and then issue new requests through the API. It is challenging to deploy reactive measurements, as it "generally takes a few minutes to get the result of a measurement" [8] and most VPs send 4–12 packets per second [8].

PacketLab: Proposed in 2017, PacketLab [17] provides a packet-oriented interface for sending and receiving packets via a distributed set of VPs, similar in goal to Scriptroute. PacketLab's architecture includes (1) a controller that provides centralized access to a set of VPs, (2) packet-sending policy enforced through BPF filters, and (3) authentication of measurements through cryptographic certificates. In recent years, the PacketLab authors reported prototype deployment on EdgeNet [33, 34] and implementations of ping, traceroute, DNS lookups, and HTTP requests. A PacketLab implementation of a protocol that uses TLS (such as HTTPS) would be complex, requiring the implementer to marshal packets through a TLS library off the VP.

EdgeNet: In 2017, researchers at Sorbonne began building a software-only platform for deploying distributed network services, motivated by the observation that maintaining and debugging hardware required six full-time people at PlanetLab [6]. Site hosts contribute *software* (VM) nodes to EdgeNet. EdgeNet operators seek to manage the nodes with off-the-shelf software, such as Kubernetes, rather than customize the operating system. Researchers use these software nodes by publishing Docker containers that EdgeNet can deploy on the nodes [7]. As of 2024, EdgeNet consists of ≈50 nodes.

FLOTO and PINOT: Both PINOT [4] and FLOTO [15] are recent (2022) infrastructures consisting of densely deployed Raspberry Pi 4 VPs in select locations. As of October 2024, PINOT is mostly deployed in Santa Barbara, while FLOTO is mostly deployed in Illinois. Following EdgeNet's lead, both are managed with existing solutions – FLOTO uses openBalena and Kubernetes, while PINOT uses SaltStack. Both FLOTO and PINOT invite researchers to deploy containers built for the ARM architecture on their nodes.

3 Platform Goals and Design Decisions

Goal: Easy to Use: To make the environment easy to use, we provide Python interfaces to measurement capabilities present on a collection of remote vantage points, and thoroughly document our interfaces [20]. The environment executes the measurements on the VPs, and provides the results as objects. We chose to provide Python interfaces as Python is extensively used in the measurement

community, both in academia and industry, with a large set of modules available for re-use.

Goal: Performant: The delay between measurement and result should be small, so that researchers can build complex reactive measurements. We built our environment with an event-driven API, where results return to the researcher's code as they arrive, with simple method calls. Further, we provide centralized access to the VP controller interface, where code runs as close as possible to the VP controller to further minimize delay.

Goal: Site-host transparent: The environment should allow platform operators to accurately describe the types of measurements the VPs will do. We chose to build our enviroment with measurement primitives, rather than provide a packet sending interface, so that we can precisely describe the type of traffic that the site host should expect to see, and communicate risks around each of the available measurement primitives.

Goal: Interoperable and Extensible: Using off-the-shelf and easily deployable components will maximize avenues of future deployment. We used components available in scamper [18] to provide measurement capabilities on VPs, and to support centrally scheduling and receiving of measurements on VPs (§4). Importantly, scamper is interoperable, as it builds and runs on a diverse set of operating systems and architectures, has few (all optional) external dependencies, can run inside containers, and is available in packaged form. Crucially, scamper is extensible, and provides interfaces to add measurement primitives.

Value of our approach: In 2023, Fiebig described four requirements for producing robust and effective measurement artifacts [11]. Our proposal shares Fiebig's aspirations for a community measurement infrastructure that improves reliability and accessibility of active measurement capability. Compared to prior work (§2), our approach provides programmatic interfaces to coordinate the use of measurement primitives across a distributed collection of VPs. These interfaces allow researchers to focus on collecting and analyzing data, which we believe will increase accessibility of active measurement capability. Researchers do not have to build containers in order to use VPs, or reimplement measurement techniques using packet-sending interfaces. Our approach allows for performant researcher access to VPs through a centralized controller, while being transparent with site-hosts about what types of measurements their VPs will do.

4 Architecture

Our system consists of three components: (1) reference implementations of active measurement primitives deployed on VPs around the world, (2) a controller that interfaces with VPs, making them available for use from a central location, and (3) an environment for scheduling, interpreting, and storing measurements. Figure 2 illustrates our high-level architecture.

For the first component, we deploy scamper to provide implementations of our measurement primitives. Scamper, written in C, contains implementations of traceroute and ping for simple IP topology and delay measurements, DNS

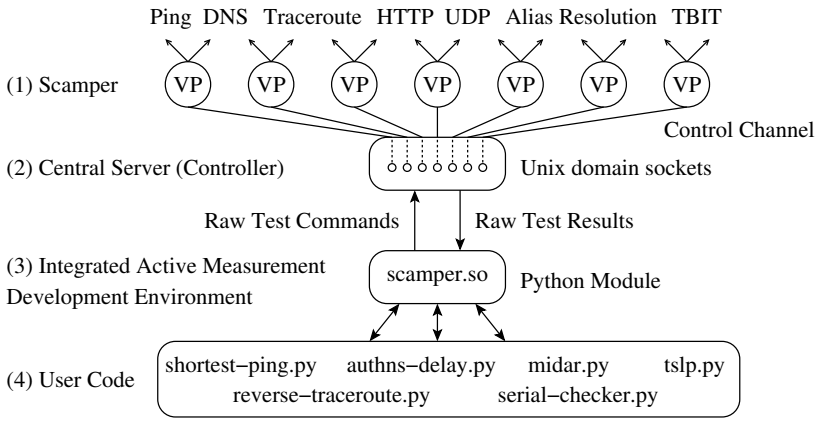

Fig. 2. System Architecture. Scamper processes on VPs connect to a central controller. Scripts access measurement primitives on VPs using an integrated active measurement development environment deployed on, or next to, the controller.

lookups for resolving names, HTTP(S) to interact with webservers, UDP probes to interact with query-response services such as NTP and SNMP, alias resolution methods for identifying which IP addresses belong to the same router, packet capture to selectively record specific packets, and TBIT [23] for inferring properties of a remote TCP stack.

Scamper's remote controller terminates connections from these VPs on the central server. Each VP is represented by a Unix domain socket in the file system. To overcome the complexity of scamper's existing APIs, we built a Python module that abstracts this complexity. We implemented this module using Cython [28], which provides a simple way to write Python bindings for C libraries, allowing us to provide native Pythonic interfaces for scamper's active measurement capabilities. The binding is implemented in ≈11K lines of Cython.

Our module exposes two broad collections of classes. The first collection consists of interfaces for interacting with vantage points, which minimize the complexity of managing tasks distributed across a collection of VPs. The second collection consists of interfaces for interacting with measurement results, which normalize the methods and attributes across different measurement types, as the interfaces presented by scamper's underlying primitives were inconsistently named or presented [13].

Coordination Classes: Figure 3 identifies the core classes and methods for managing and executing measurements across VPs. The most important class is the *ScamperCtrl* class, through which scripts schedule measurements, find VPs, and obtain measurement results. We illustrate these methods with examples in §5. Briefly, the *add* methods allow a script to add VPs to an experiment. Because selecting an initial set of VPs is a common workflow, the *ScamperCtrl* constructor allows specification of VPs when creating a *ScamperCtrl* object.

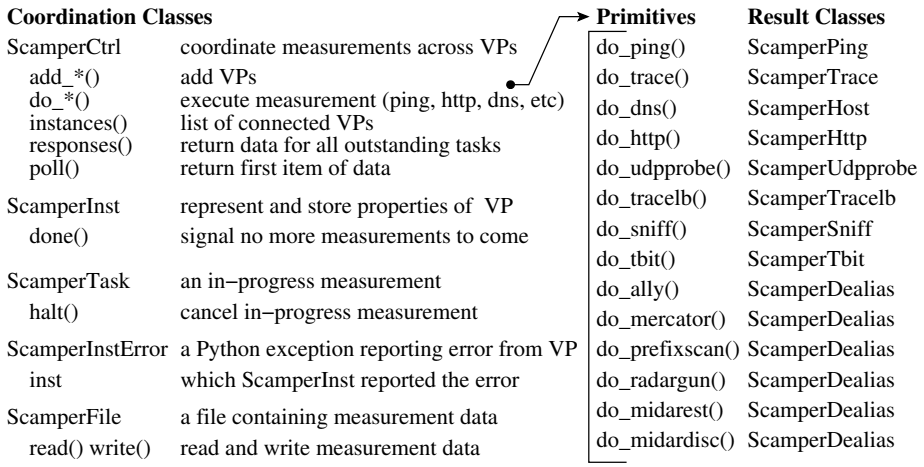

Fig. 3. VP coordination classes (left), primitives (middle), and result classes (right) in our integrated active measurement development environment.

The *do* methods allow a script to instruct a VP to use one of its measurement primitives (middle-column of figure 3). The *do* methods allow measurements to execute synchronously, where the *do* method blocks until it returns the completed measurement, or asynchronously, where the method returns a *ScamperTask* object representing the issued measurement, with control immediately returning to the script. A script would use a synchronous call when a measurement must complete before the script will issue additional measurements, and would use an asynchronous call when it wanted to issue further measurements or do some other computation. These *do* methods present a consistent API over the inconsistent API provided by scamper. The parameters to each method have the same name when they mean the same thing, and the parameters have consistent units. For example, all time-related parameters use Python's *timedelta* or a *float* representing the number of seconds. The *instances* method lists the available VPs, with each VP represented in the environment with a *ScamperInst* object.

Finally, the *responses* method returns measurement results, as they arrive, for all scheduled measurements, while the *poll* method allows a script to obtain just the next available result, waiting for up to the specified length of time. This provides researchers with multiple possible workflows, each of which is applicable in different scenarios. A script can use (1) synchronous measurement via *do* when a measurement must complete before the script will issue additional measurements, (2) blocking asynchronous measurements via *do* and *responses* when the script needs to issue multiple parallel measurements, and then collect all responses before reacting to the results of the measurement, or (3) non-blocking asynchronous measurements via *do* and *poll* when they have a large number of measurements to stream in parallel across one or more VPs. We decided to centralize as much of the coordination on the *ScamperCtrl* object as possible,

including issuing measurements, as otherwise scripts would issue measurements via *ScamperInst* and collect them via *ScamperCtrl*, which we judged to be an unusual, inelegant, workflow.

Scripts can indicate that they have no further work for a given VP by calling the *done* method on a *ScamperInst* object. Similarly, scripts can indicate that they no longer need a given measurement to complete by calling the *halt* method on a *ScamperTask* object. If any VP encounters an error while executing a measurement, the environment will raise a *ScamperInstError* exception, allowing the script to identify the specific instance that raised the exception, and a text string explaining the exception condition. Finally, the *ScamperFile* class allows scripts to work with scamper's native binary output format. To support the common workflow of writing measurement output to a file, a script can provide a *ScamperFile* object to the *ScamperCtrl* constructor, which will automatically record all measurements to that file.

Measurement Result Classes: The middle and right columns of figure 3 list the available measurement primitives and measurement result classes. For each measurement result, we added Pythonic interfaces to the data. We represent time using Python's *datetime* and *timedelta* classes, provide iterators and generators for convenience, normalize the names of fields across result classes, and provide convenient methods to minimize the amount of code and increase clarity. For example, our environment provides a *min_rtt* attribute for *ScamperPing* that internally iterates through the responses the script receives to obtain the minimum RTT observed so that the script's author does not have to embed that code themselves. Similarly, our environment provides an *ans_addrs* method for *ScamperHost* that internally iterates through the DNS resource records in the answer section of the response, collecting only IPv4 and/or IPv6 addresses in the answer. This method saves the script's author from writing code to interpret each resource record's class and type.

5 Illustrative Examples

We detail three examples that illustrate the capabilities described in §4, and demonstrate the elegance and succinctness of code that uses the environment. We then summarize several other experiments the environment has supported.

Asynchronous Case, Shortest Ping: Figure 4(a) contains a complete script that finds the VP with the shortest delay to a given IP address, rendering that VP a proxy for the approximate geolocation of that IP address. First, the script instantiates a *ScamperCtrl* object with a directory of Unix domain sockets, each of which represents a remote VP. Then, the script issues a ping measurement using each instance held in the *ScamperCtrl* object. These measurements run on the VPs asynchronously in parallel. The script collects measurement replies using the *responses* method on *ScamperCtrl*, finally exiting when there are no measurements outstanding. The script tracks the minimum RTT observed across all VPs, printing the name of the VP that provided the minimum RTT. Figure 4(a) is a complete script in 19 lines of code, with 4 blank for readability,

```
import sys
from scamper import ScamperCtrl

if len(sys.argv) != 3:
  print("usage: shortest-ping.py $dir $ip")
  sys.exit(-1)

ctrl = ScamperCtrl(remote_dir=sys.argv[1])
for inst in ctrl.instances():
  ctrl.do_ping(sys.argv[2], inst=inst)

min_rtt = min_vp = None
for res in ctrl.responses():
  if res.min_rtt is None: continue
  if min_rtt is None or min_rtt > res.min_rtt:
    min_rtt, min_vp = res.min_rtt, res.inst

print("%s %.1f ms" % (min_vp.name,
      min_rtt.total_seconds() * 1000))
```

(a) Shortest Ping

(b) Authoritative Nameserver Delay ⟶

```
import sys
from scamper import ScamperCtrl
if len(sys.argv) != 3:
  print("usage: authns-delay.py $inst $zone")
  sys.exit(-1)
inst, zone = sys.argv[1:]
ctrl = ScamperCtrl(remote=inst)
res = ctrl.do_dns(zone, qtype="NS", sync=True)
for ns in res.ans_nses():
  ctrl.do_dns(ns, qtype="A")
  ctrl.do_dns(ns, qtype="AAAA")
addrs = {}
for res in ctrl.responses():
  for addr in res.ans_addrs():
    addrs[addr] = res.qname
for addr in addrs.keys():
  ctrl.do_ping(addr)

for res in ctrl.responses():
  if res.min_rtt is not None:
    print(f"{res.dst} {addrs[res.dst]}",
          f"{res.min_rtt.total_seconds() * 1000}")
  else:
    print(f"{res.dst} {addrs[res.dst]} none")
```

Fig. 4. Illustrative examples: finding the VP closest to an IP address (left) and measuring the delay from a VP to a zone's authoritative nameservers (right).

and 3 reporting the correct usage of the script. The script completes in ≈5 seconds on Ark (§2): 4 seconds for 4 echo requests from each VP, plus time sending measurement commands and receiving responses from VPs.

Synchronous Case, Nameserver Delay: Figure 4(b) illustrates the synchronous approach with a script that determines the RTT to nameservers authoritative for a given zone. The first measurement – to determine the names of those nameservers – must complete before subsequent measurements can proceed. Therefore, the first DNS query in figure 4(b) uses *sync=True* so that the measurement completes synchronously. Note, the script could have issued the measurement asynchronously and collected the result using *responses* or *poll* (§4), but that would have been inelegant, and would have required writing more code. The remaining steps in figure 4(b), which determine the IP addresses of those nameservers, and then obtain the RTTs to them, complete asynchronously. This is a complete 27-line script, with 4 blank lines, lines, 3 usage lines, and one wrapped line to fit within a single column. The script completes in ≈7 seconds – 4 seconds for 4 echo requests, 1-2 seconds for the DNS queries, plus time sending measurement commands and receiving responses.

Characterizing Netflix CDN Infrastructure: Characterizing CDNs requires geographically distributed VPs to discover and probe cache servers from different locations [1, 9]. Our example collects the IP topology towards Netflix's CDN infrastructure of Open Connect Appliances (OCAs) [5], through Netflix's fast.com speed test service. Netflix directs clients (usually a web browser) to a nearby OCA, from which the client would then transfer large objects in order

```
if len(sys.argv) != 2:
  print("usage: fast.py $dir")
  sys.exit(-1)

url = "https://api.fast.com/netflix/speedtest/v2?"
date = datetime.now().strftime(
        "%Y-%m-%d_%H:%M:%S")
filename = f"fast.{date}.warts.gz"
ctrl = ScamperCtrl(remote_dir=sys.argv[1],
        outfile=ScamperFile(filename, "w"))

# query for api.fast.com IP address from each VP
http_addrs = {}
for inst in ctrl.instances():
  ctrl.do_dns(urlparse(url).hostname, inst=inst)
for obj in ctrl.responses(timeout=5):
  addrs = obj.ans_addrs()
  if len(addrs) > 0:
    http_addrs[obj.inst] = addrs[0]

# HTTP query for server names from each VP
dns_hosts = {}
for inst, ip in http_addrs.items():
    ctrl.do_http(ip, url, inst=inst)
for obj in ctrl.responses(timeout=25):
  if not isinstance(obj, ScamperHttp): continue
  json_data = json.loads(obj.response.decode())
  tgts = json_data.get("targets", [])
  tgturls = [tgt["url"] for tgt in tgts]
  dns_hosts[obj.inst] = [urlparse(tgt).hostname
                         for tgt in tgturls]

# query for speedtest server IPs from each VP
server_ips = {}
for inst, hosts in dns_hosts.items():
  for host in hosts:
    ctrl.do_dns(host, inst=inst)
for obj in ctrl.responses(timeout=5):
  if isinstance(obj, ScamperHost):
    server_ips[obj.inst] = obj.ans_addrs()

# collect topology to each IP with TCP traceroute
for inst, ips in server_ips.items():
  for ip in ips:
    ctrl.do_trace(ip, wait_timeout=1, dport=443,
                  method="tcp", inst=inst)
for obj in ctrl.responses(timeout=25): pass
```

Fig. 5. Collecting data on topological deployment of fast.com speed test servers with VP-specific DNS lookups, HTTP queries, and TCP traceroutes.

to measure speed. The speedtest servers returned by Netflix depend on the VP. From each VP, (1) we need DNS lookups to know the IP address of the fast.com web-based API that returns speedtest servers for each VP, in case the address returned depends on the resolver used by the VP, (2) we need HTTP capability to fetch JSON from the fast.com RESTful API via HTTP that contains URLs, one for each speedtest server recommended by Netflix for the VP to test against, (3) we need DNS lookups to know the IP addresses of these servers, and (4) we need traceroute and ping to determine basic topological and performance properties. Our environment provides all of these primitives, and our example (figure 5) closely follows these steps, collecting measurements from all available VPs in parallel. The script runs in ≈60 seconds on Ark – up to 25 seconds each for the traceroutes and HTTP queries, plus two rounds of DNS queries. Appendix A describes this example in further detail.

Complex Measurements: We reproduced a portion of Trufflehunter [29], which infers the popularity of rare domains through queries to large public recursive resolvers operated by Google, OpenDNS, Quad9, and Cloudflare (see [21]). We have also reproduced two macroscopic studies of the Internet's router infrastructure – the first used SNMPv3 queries to identify router vendors and aliases [3], and the second used those vendors to train a fingerprint classifier to infer vendors for other routers that did not return an SNMPv3 response [2]. Finally, we added primitives to support MIDAR [16], which uses a set of VPs to probe router interfaces with the goal of finding which interface IP addresses belong to the same router (are aliases) – those where response IPID values appear to be derived from a central counter. Intuitively, MIDAR solves this problem at

Internet scale by providing a distributed set of VPs with a set of probe definitions (ICMP, UDP, TCP, IP addresses) and a sliding-window schedule that specifies when these probes should enter the network, so that two candidate aliases have a high chance of receiving probes that allow this single central counter property to be observed. We replaced 2554 lines of opaque Ruby code with a 902 line Python script that clearly conveys the organizational requirement involved in the measurements. Appendix B describes this example in further detail.

These experiments are difficult to support on existing measurement platforms (§2), as they require distributed, coordinated probing facilities that allow for fine-grained control of measurements. We found these examples straightforward to implement. We did not have to write code to execute directly on VPs, copy results off the VPs, or coordinate VPs, and the remainder of the scripts used environment features that made our measurement intentions clear.

6 Discussion and Future Work

We have designed and built a programming environment to accelerate innovation in scientific Internet measurement. Our priorities were to lower the threshold for implementing complex measurement experiments, in a performant environment, while also allowing platform operators to accurately describe the types of measurements the VPs will do to site hosts.

One gratifying outcome of our Python-based platform architecture is its use in a Python-based Jupyter Notebook environment. Rather than edit scripts in a text editor, one student developed solutions in a Jupyter Notebook environment, with which they were familiar. They reported that this approach significantly lowered the learning curve required to conduct their active measurements.

We designed our environment to operate on, or adjacent to, a central controller that interfaces with VPs distributed around the world. The logic for measurement primitives (e.g., traceroute, HTTP, etc) executes on the VPs, while the logic that uses the results executes on or adjacent to the central controller. For experiments where delay between the controller and VP is problematic, we are currently exploring deployment architectures, such as sandboxed containers, that would enable researchers to safely deploy scripts onto the VPs, without requiring a platform operator to manage shell accounts for platform users.

We have publicly released our implementation [19] and documentation [20], so that the Internet measurement and operations communities can extend it, and benefit from our work. We plan to support its use on CAIDA's Archipelego (Ark) infrastructure, and seek to spur discussion with other active measurement infrastructures as to how they can safely modernize their capabilities. We believe that thinking about distributed measurement through the lens of required measurement primitives, rather than ad-hoc collections of software to collect measurement data, is a useful exercise, as implementing primitives lowers the barrier to other researchers continuing the work and increases incentives to repeat measurements. Our ultimate vision for this work is a world where researchers can ask, and answer, grand questions about the global Internet in near-real-time.

A Measuring CDN Catchment and Routing

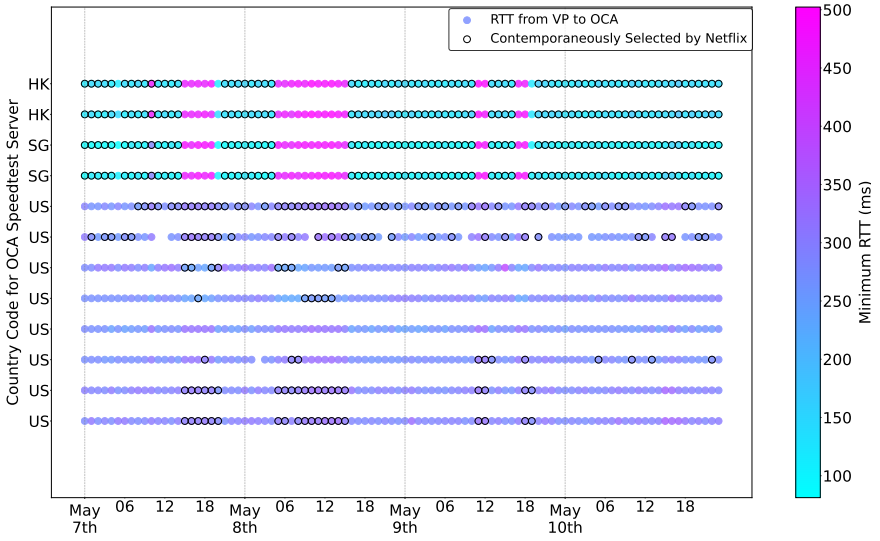

Fig. 6. Effect of latency variation on server selection strategy during May 2024, for a VP located in Thimphu, Bhutan. The X-axis shows the VP's local time. Each row shows a unique /24, annotated with the country using Netflix-assigned hostnames.

This measurement consists of two scripts. The first collects details of the fast.com servers returned to each VP, and is shown in figure 5. This script runs hourly out of cron, and stores the results in scamper's archival format for subsequent analysis. The second script, which we elide, processes the archived data returned from the first script to extract the speedtest servers that Netflix had returned over time for each VP, building history of possible speedtest servers for each VP. The script measures latency, with ping, between the VPs and the set of OCAs (one randomly selected per /24) for each VP, to characterize the condition of the path between the VPs and the proximate OCAs.

With the data that we collected beginning April 11th 2024, we were able to observe some interesting patterns, one of which we highlight here as an illustrative example. We observed how traffic load appeared to influence which OCA servers Netflix would return to the VP. Figure 6 shows RTT values to speedtest servers returned to a VP located in Bhutan during four days in May 2024. Those servers contemporaneously selected by Netflix are noted with black circles. Netflix generally returned servers in Hong King and Singapore, and those had the lowest observed latency of ≈100ms. However, those servers occasionally had significant latency spikes to ≈500ms. During those latency spikes, Netflix directed the Bhutan VP to servers in the U.S., which had a latency of ≈250ms.

B Router Alias Inference and Fingerprinting

	2023-02	2024-02
Input:		
IPv4 addresses probed:	2.64M	3.58M
Traceroute data window:	2 weeks	3 weeks
Ark VPs w/ traceroute data:	93	142
Number of countries:	37	52
Alias Resolution:		
Ark VPs used for MIDAR:	55	101
Ark VPs used for iffinder:	46	101
Ark VPs used for SNMP:	-	7
MIDAR + iffinder Graph:		
Nodes with at least two IPs:	75,660	107,976
Addresses in nodes with at least two IPs:	284,479	425,964
SNMP Graph:		
Nodes with at least two IPs:	-	48,899
Addresses in nodes with at least two IPs:	-	208,313
MIDAR + iffinder + SNMP Graph:		
Nodes with at least two IPs:	-	125,370
Addresses in nodes with at least two IPs:	-	516,867

Table 1. Properties of ITDKs collected before (2023-02) and after (2024-02) we developed our solution.

These measurements, briefly described in §5, consisted of multiple related scripts that we integrated into an automated workflow for building the 2024-02 ITDK. Table 1 provides statistics illustrating the growth of the ITDK between February 2023 and February 2024, driven by the expansion of Ark VPs. Overall, we increased the number of Ark VPs providing topology data from 93 to 142, the number of addresses probed from 2.64M to 3.58M, doubled the number of VPs that we use for alias resolution probing, and found aliases for 50% more addresses in 2024-02 than we did for 2023-02. These 3.58M addresses were observed in the middle of a traceroute path, and are most likely router interface addresses. We use the term "node" to distinguish between our router inferences, and the actual routers themselves. By definition, all routers have at least two IP addresses. Our "nodes with at least two IPs" are the subset of routers we were able to observe with that property.

For 2024-02, we also evaluated the gains provided by SNMPv3 probing, following the work by Albakour *et al.* published in 2021 that showed many routers return a unique SNMP Engine ID in response to a SNMPv3 request [3]. The basic idea is that different IP addresses returning the same Engine ID, number of boot counts, and inferred boot time in response to SNMPv3 queries are likely

aliases. Of the 3.58M addresses we probed, 669K returned an SNMPv3 response. We inferred that IP addresses belonged to the same router when they return the same SNMP Engine ID, the size of the engine ID was at least 4 bytes, the number of engine boots was the same, and the router uptime was the same; we did not use the other filters in section 4.4 of the IMC paper [3]. This inferred 48,899 nodes with at least two IPs, many of which were shared with existing nodes found with MIDAR + iffinder. In total, when we combined MIDAR, iffinder, and SNMP probing, we obtained a graph with 125,370 nodes with at least two IPs, covering 516,867 addresses.

Finally, we also implemented the methodology published in 2023 by Albakour *et al.* that described a way to infer router vendors [2]. An SNMPv3 response embeds a vendor identifier in the SNMP Engine ID. The basic idea is that the subset of routers that return an SNMPv3 response allow us to learn fingerprint rules for other routers that do not respond to SNMPv3 probes, but will send responses to TCP, UDP, and ICMP probes. For the 2024-02 ITDK, we sent TCP, UDP, and ICMP probes to the 3.58M router interface addresses, using the probing strategy described in [2], implemented in 170 lines of code. We inferred 96 rules to infer vendors from TCP, UDP, and ICMP response patterns, implemented in 483 lines of code (which also includes logic to infer router aliases from the same SNMP responses). Our inference script followed the same approach as in [2] except that it also considered byte-swapped IPID values for TCP responses. When we used these rules to map response patterns to vendors, we inferred vendors for 248K router interface addresses, in addition to the 669K SNMPv3-responsive addresses that directly returned a vendor identifier.

Acknowledgments

This work started with a suggestion from Bill Herrin during a CAIDA AIMS workshop that a Domain Specific Language (DSL) could accelerate discovery with active measurement. Alexander Marder suggested that we start with Python bindings for scamper. We thank the anonymous reviewers for their comments. This research was supported by National Science Foundation (NSF) grants OAC-2131987, CNS-2120399, CNS-2323219, and CNS-2212241.

References

1. Adhikari, V.K., Guo, Y., Hao, F., Hilt, V., Zhang, Z.L., Varvello, M., Steiner, M.: Measurement study of Netflix, Hulu, and a tale of three CDNs. IEEE/ACM Transactions On Networking **23**(6), 1984–1997 (2014)
2. Albakour, T., Gasser, O., Beverly, R., Smaragdakis, G.: Illuminating router vendor diversity within providers and along network paths. In: IMC. pp. 89–103 (Oct 2023)
3. Albakour, T., Gasser, O., Beverly, R., Smaragdakis, G.: Third time's not a charm: Exploiting SNMPv3 for router fingerprinting. In: IMC. pp. 150–164 (Nov 2021)
4. Beltiukov, R., Chandrasekaran, S., Gupta, A., Willinger, W.: PINOT: Programmable infrastructure for networking. In: ANRW. pp. 51–53 (Jul 2023)

5. Böttger, T., Cuadrado, F., Tyson, G., Castro, I., Uhlig, S.: Open connect everywhere: A glimpse at the Internet ecosystem through the lens of the Netflix CDN. ACM SIGCOMM Computer Communication Review **48**(1), 28–34 (2018)
6. Cappos, J., Hemmings, M., McGeer, R., Rafetseder, A., Ricart, G.: EdgeNet: A global cloud that spreads by local action. In: SEC. pp. 359–360 (Oct 2018)
7. Şenel, B.C., Mouchet, M., Cappos, J., Fourmaux, O., Friedman, T., McGeer, R.: EdgeNet: A multi-tenant and multi-provider edge cloud. In: EdgeSys. pp. 49–54 (Apr 2021)
8. Darwich, O., Rimlinger, H., Dreyfus, M., Gouel, M., Vermeulen, K.: Replication: Towards a publicly available Internet scale IP geolocation dataset. In: IMC. pp. 1–15 (Oct 2023)
9. Doan, T.V., Bajpai, V., Crawford, S.: A longitudinal view of Netflix: Content delivery over IPv6 and content cache deployments. In: IEEE INFOCOM (Jul 2020). https://doi.org/10.1109/infocom41043.2020.9155367
10. Dovrolis, C., Gummadi, K., Kuzmanovic, A., Meinrath, S.D.: Measurement lab: Overview and an invitation to the research community. Computer Communication Review **40**(3), 53–56 (Jul 2010)
11. Fiebig, T.: Crisis, ethics, reliability & a measurement.network: Reflections on active network measurements in academia. In: ANRW. pp. 44–50 (Jul 2023)
12. Huffaker, B., Plummer, D., Moore, D., k claffy: Topology discovery by active probing. In: SAINT. pp. 90–96. Nara City, Japan (Jan 2002)
13. Jonglez, B.: drakkar-lig scamper-pywarts (Mar 2021), `https://github.com/drakkar-lig/scamper-pywarts`
14. Kalidindi, S., Zekauskas, M.J.: Surveyor: An infrastructure for Internet performance measurements. In: INET. San Jose, CA (Jun 1999)
15. Keahey, K., Feamster, N., Martins, G., Powers, M., Richardson, M., Schrubbe, A., Sherman, M.: Discovery testbed: an observational instrument for broadband research. In: eScience (Oct 2023)
16. Keys, K., Hyun, Y., Luckie, M., k claffy: Internet-scale IPv4 alias resolution with MIDAR. IEEE Transactions on Networking **21**(2), 383–399 (Apr 2013)
17. Levchenko, K., Dhamdhere, A., Huffaker, B., kc claffy, Allman, M., Paxson, V.: PacketLab: A universal measurement endpoint interface. In: IMC. pp. 254–260 (Nov 2017)
18. Luckie, M.: Scamper: a scalable and extensible packet prober for active measurement of the Internet. In: IMC. pp. 239–245 (Nov 2010)
19. Luckie, M.: Scamper (Nov 2024), `https://www.caida.org/catalog/software/scamper/`
20. Luckie, M.: Scamper Python module documentation (Nov 2024), `https://www.caida.org/catalog/software/scamper/python/`
21. Luckie, M.: Understanding the deployment of public recursive resolvers (May 2024), `https://blog.caida.org/best_available_data/2024/05/06/understanding-the-deployment-of-public-recursive-resolvers/`
22. McGregor, T., Braun, H.W.: Balancing cost and utility in active monitoring: The AMP example. In: INET. Yokohama, Japan (Jul 2000)
23. Medina, A., Allman, M., Floyd, S.: Measuring the evolution of transport protocols in the Internet. Computer Communication Review **35**(2), 37–52 (Apr 2005)
24. NLNOG: Ring, `https://ring.nlnog.net/`
25. Paxson, V., Mahdavi, J., Adams, A., Mathis, M.: An architecture for large-scale Internet measurement. IEEE Communications Magazine **36**(8), 48–54 (1998)
26. Paxson, V., Adams, A., Mathis, M.: Experiences with NIMI. In: PAM. Hamilton, New Zealand (Apr 2000)

27. Peterson, L., Bavier, A., Fiuczynski, M.E., Muir, S.: Experiences building PlanetLab. In: OSDI. pp. 351–366. Seattle, WA (Nov 2006)
28. Project, C.: Cython: C-extensions for Python (Oct 2024), https://cython.org/
29. Randall, A., Liu, E., Akiwate, G., Padmanabhan, R., Voelker, G.M., Savage, S., Schulman, A.: Trufflehunter: Cache snooping rare domains at large public DNS resolvers. In: IMC. pp. 50–64 (2020)
30. RIPE NCC: RIPE Atlas - coverage, https://atlas.ripe.net/coverage/
31. Shavitt, Y., Shir, E.: DIMES: let the Internet measure itself. Computer Communication Review **35**(5), 71–74 (2005)
32. Spring, N., Wetherall, D., Anderson, T.: Scriptroute: A public Internet measurement facility. In: USITS. pp. 225–238. Seattle, WA (Mar 2003)
33. Yan, T.B., Chen, Y., Chen, A., Zhang, Z., Huffaker, B., Mok, R., Levchenko, K., kc claffy: Poster: PacketLab - tools alpha release and demo. In: IMC. pp. 766–767 (Oct 2022)
34. Yan, T.B., Zhang, Z., Huffaker, B., Mok, R., Levchenko, K., kc claffy: Poster: Empirically testing the PacketLab model. In: IMC. pp. 724–725 (Oct 2023)