A Modular Framework for End-Host Driven Network Measurements
COS561 Spring '23 Course Project Report

Tanu Batra (tanu.batra@rutgers.edu)[a], Jack Defay (jd6058@princeton.edu)[b], Varun Rao (varunrao@princeton.edu)[b], and Leon Schuermann (lschuermann@princeton.edu)[b]

[a]Rutgers University
[b]Princeton University

Abstract

Today's networks get more complex by the minute: increases in bandwidth requirements, low-latency links, ubiquitous Internet of Things deployments, and strengthened security requirements demand extremely reliable, scalable and intelligent networks. Unfortunately, many of these requirements are interconnected. For instance, new features are usually deployed using middleboxes, but those increase network complexity and makes it harder to diagnose issues. Additional hardware-assisted monitoring infrastructure on routers lowers the resources available to packet processing. Making matters worse is the fact that network operators such as Internet Service Providers have only limited insights into the resulting network behavior; they commonly rely on informal customer feedback.

In an effort to combat these issues, this report presents our work on developing a modular framework for end-host driven network measurements. In particular, we present our design for an expressive Domain Specific Language (DSL) to implement custom network measurements, alongside mechanisms to aggregate captured information in a graph-based data structure. Furthermore, we demonstrate our implementation of these concepts in an integrated hardware–software system. We demonstrate the applicability of our framework through a deployment on the Princeton University campus network.

1 Introduction

Today's Internet Service Providers (ISPs) face a challenge of improving their networks to handle an ever-increasing demand for performant and flexible Internet connectivity. At the same time, they also strive to provide a reliable service to their customers. In particular, to ensure reliability of their networks, ISPs often rely on informal customer feedback or complaints to diagnose network issues, which may involve direct interaction with customers or physically probing affected network hardware. Conventional networking hardware such as commodity routers and switches provides only limited insight into network behavior. This makes it difficult to identify issues related to routing, packet reordering, packet filtering, or middlebox behavior.

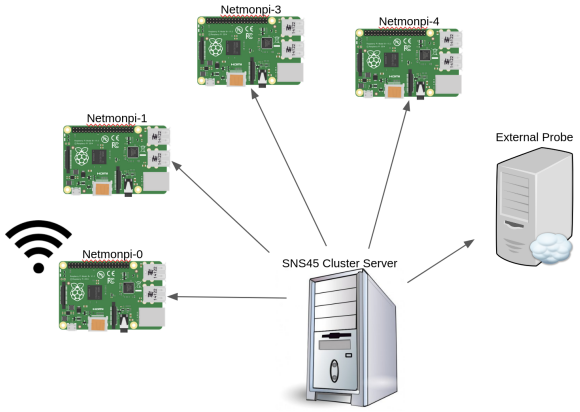

Figure 1: System Overview - 4 Ethernet Raspberry Pi, 1 Wifi Raspberry Pi, 1 External Probe

RIPE Atlas [10] is a global Internet-scale measurement network comprised of active network probe devices. It helps to increase network visibility and simplifies troubleshooting of common pathological network behaviors. While effective for diagnosing issues concerning routing, latency and censorship on a global scale, such a solution lacks the granularity and expressiveness required to cover integrated ISP-networks themselves. Furthermore, conventional and readily available diagnostic information such as raw link statistics and routing table dumps do not give significant insights into the actual network behavior.

To address these issues, we aim to build a measurement platform based on active network probes, integrated into a network as end-hosts. A centralized controller is used to coordinate measurements by instructing probe behavior, and collecting and analyzing results. We validate our system with a Mininet-based network simulation environment, and through a small-scale deployment on the Princeton University campus

data network.

Our work contributes to the space of network measurement solutions by providing a flexible and fine-grained measurement platform to network administrators, illustrated in Figure 1. This platform enables the user to define custom and arbitrarily complex measurements through a simple Python-based Domain-Specific Language (DSL). We hope that this platform will enable ISPs to improve and adapt their networks to cope with increasing demand while maintaining reliability.

2 Related Work

In this section, we contextualize our framework within the body of related work. In particular, we describe existing systems in the area of active network probing, as well as network monitoring.

2.1 Dedicated Hardware-based Network Measurement Systems

Prior work has used probes to perform both small and large scale network monitoring and measurements. One example is CheesePi, which uses a Raspberry Pi-based distributed measurement system for monitoring home internet connections (e.g. bandwidth and loss rate) [8]. Another such framework, the Archipelago Measurement Infrastructure, performs network monitoring using 120 Raspberry Pi probes deployed on major US broadband access provider networks. They estimated the topology and dynamics of the network by measuring the connectivity and congestion across network peerings through active measurements [2].

RIPE Atlas [10] is a large scale network measurement platform which is directly and most related to our work. It performs active measurements using global network of 12800 probe devices deployed through volunteers. Although it provides an "unprecedented understanding of the state of the Internet in real time", there are several limitations associated with its use. First, it is difficult to correlate results between probes. Second, one cannot send arbitrary packet data. Third, one cannot easily define new measurement types. Fourth, and most importantly, it is hard to explicitly coordinate measurements, such as by sending packets between two probes directly. In contrast, our system is strictly more expressive and thus enables a strict superset of measurements to be implemented. Other comparable large scale platforms include SamKnows, BISmark and Dasu [1].

[1] https://stat.ripe.net/m/widget/atlas-ping-measurements#w.mode=condensed&w.measurement_id=2006&w.probe_id=1005701&w.starttime=2023-04-19T00%3A00%3A00&w.endtime=2023-05-07T00%3A00%3A00&w.resolution=1h

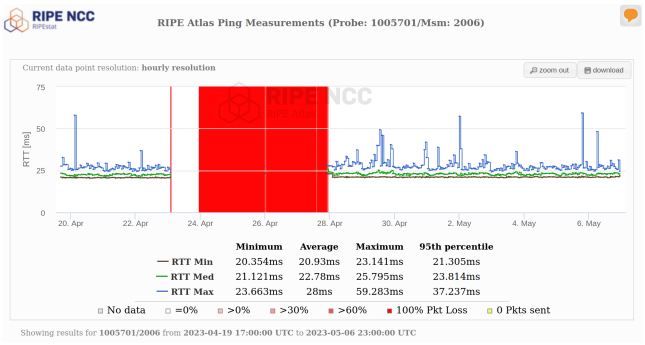

Figure 2: RIPE-Atlas can capture various types of network misbehavior: in this example, an active probe device on a commodity Internet uplink is unable to reach the IPv6-address of `m.root-servers.net` (`2001:dc3::35`) due to a routing error within the ISP's network (highlighted in red), while other hosts remained reachable over the same time span[1].

2.2 Software-based Network Measurement Systems

In contrast to the previously mentioned dedicated network monitoring systems, projects such as Microsoft PingMesh [5], Google Black Box Monitoring [4] and NetSight [6] integrate active probing software into existing systems, to acquire metrics such as latency and packet loss.

2.3 Network Monitoring Applications

Network measurements can be used in a variety of ways; identification of censorship devices and middleboxes are two such applications.

Prior work [9] has identified censorship devices through a novel traceroute method. On a much larger scale, [7] have identified censorship activities that block DNS, HTTP, or HTTPS requests within 122 countries through end-to-end network measurement frameworks.

Detal et al. [3] extended a basic traceroute measurement to detect middlebox interference. Specifically, they proposed *Tracebox* as a tool that detects modifications by middleboxes in network paths by sending IP packets with varying TTL values and analyzing the returned ICMP messages. Furthermore, it can pinpoint the specific network hop where the interference occurs using information provided in the ICMP message. So far our work has focused on basic network measurements such as ping and traceroute, and some more complicated measurements like NAT middlebox detection. However, our framework is extensible to other complex measurement tasks like detecting censorship devices and other middlebox hardware, as outlined above.

3 Design

Motivated by prior work on network measurement, in this section we continue to describe our system architecture. We start by providing a general overview of our framework, and proceed to discuss the design of our *measurement definition interface*. Influenced by this design, we further describe our software and hardware architecture, as well as our data aggregation strategy.

3.1 System Architecture

Our system employs a central controller. This controller is responsible for implementing the majority of the system's logic, and coordinates attached probe devices accordingly. To conduct a measurement, the central controller constructs packets and sends them to probe devices, to be injected into the network on the probe's network interface. Depending on the measurement definition, probes further record any incoming traffic, relaying received packets back to the controller. Time-sensitive measurements can utilize timestamps gathered by probe devices for sent and received packets, respectively. This inherently captures all end-host visible network behavior. For example if a packet header field is modified on the path between two probe devices, we can conclude that it must have been modified by an active network element. Similarly, blocked or throttled network flows can be identified. On a larger scale, individual measurements can be combined to learn attributes of more complex behavior, such as firewall policies.

These measurements can be carried out by the central controller in an automated fashion. The data gathered through measurements is aggregated into appropriate data structures at the controller, and can further be used for purposes such as network visualization, alerting, or even vulnerability detection. Figure 3 illustrates the controller software architecture.

3.2 Measurement Definition Interface

A key component of this system is the ability to define arbitrary custom measurements, which capture data about the network behavior between multiple probe devices, or between a probe and an arbitrary remote host. Our centralized server design is further motivated by the ability to easily define custom measurement types. These measurements can coordinate the behavior of multiple physically distributed probe devices from a logically centralized point. Our system integrates an intuitive and powerful interface for network administrators to define such measurements, which we describe throughout this section.

On a fundamental level, end-host behavior in Ethernet/IP-networks can be reduced to two actions: transmitting and receiving a packet. All high-level network abstractions are composed out of a combination of these actions, such as reacting to received packets with corresponding responses, or combining multiple packets into *network flows*. To enable active probing through network packets and observing any resulting behavior, it is thus sufficient to provide an interface only for sending and receiving network packets.

```
def ping(probe_a, probe_b):
    probe_b_ip = await probe_b.primary_ip()

    # Send an ICMP echo-request from probe A to B
    answered, unanswered = await probe_a.sr(
        # Scapy packet manipulation DSL
        IP(dst=probe_b_ip)
        / ICMP(type="echo-request"))

    if len(ans) > 0:
        # We received a response, return the RTT:
        return (
            answered[0].answer.time
            - answered[0].query.time
        )
    else:
        # We did not receive a response in time:
        return None
```

Listing 1: An implementation of an ICMP-based *ping* measurement between two probe devices. The `sr` method sends packets and receives related packets of the same network flow.

While expressive, a simple interface to send and receive raw network packets is not a particularly intuitive method to interact with packet data, and to correlate related network traffic. To more easily support measurements which rely on correlating sent and received packets, we offer an interface to send and receive packets which belong to the same network flow. Finally, to enable the user to intuitively craft arbitrary network packets, we use the *Scapy* Python library, which implements an expressive domain-specific language (DSL) to create, parse and manipulate network packets.

Listing 1 outlines the definition of a simple, ICMP-based *ping* measurement. As part of a measurement definition (implemented through a Python class), a developer is provided with objects for all registered probe devices in the system. In this case, individual measurements are carried out on pairs of probe devices, `probe_a` and `probe_b`. Packets can be created as defined through the Scapy-DSL: for instance, an IP-packet with an accompanying ICMP payload can be synthesized by instantiating the `IP` and `ICMP` Python classes, and combining the objects through the *slash* operator. A probe-object can be used to interact with physical probe devices; it supports sending packets (`send`), sending and receiving packets of the same network flow (`sr`), or sniffing incoming packets on the probe's network interface (`sniff`). For each method, the controller abstracts all network communication and serialization with the probe. This ensures that developers have logically centralized control with the measurement definition, and can coordinate sent and received packets even across multiple probes deployed throughout their network.

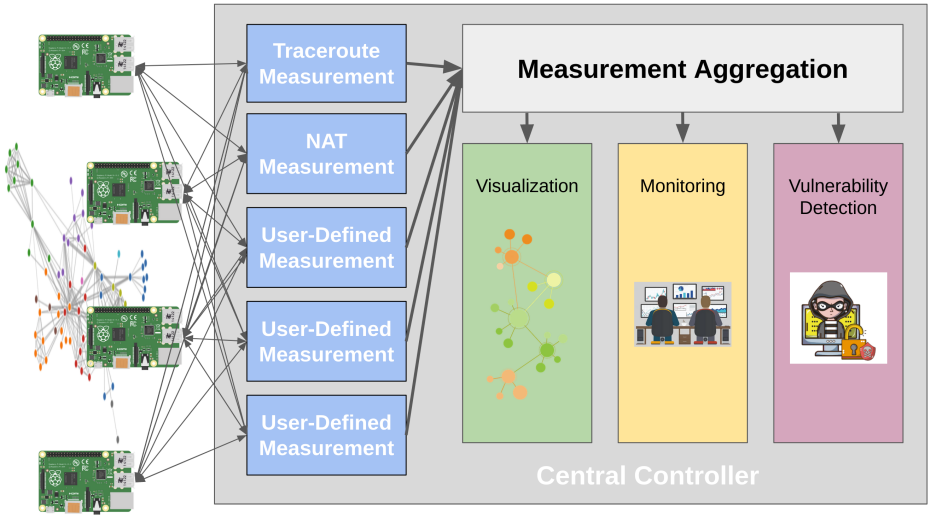

Figure 3: A diagram of our overall system architecture

3.3 Probe Software Architecture

The measurement definition interface motivates a simple software architecture for the physical network-probe devices. By avoiding the inclusion of measurement-specific functionality within the probe software itself, our system can reuse a single application deployed on every device. Generally, the probe software does not need to be modified to support new measurements.

We chose to implement our probe software in Python. To facilitate communication with the central controller, we employ the `aiomas`[2] asynchronous RPC-library. This library supports invoking multiple RPCs simultaneously, which is important to allow for a *multi-tenant* system with multiple measurements being run in parallel. Furthermore, it allows for invoking *reverse RPCs*: when probe devices are located at arbitrary points in a network, possibly behind Firewall- or NAT-devices, it may not be feasible for the controller to establish a connection to these devices directly. Instead, in our system, probe devices initiate TCP-connections to the controller, which are in turn used by the controller to perform RPC invocations on the probe devices.

A probe device offers the following methods to the central controller: `send_l2` and `send_l3` to transmit Ethernet- or IP-packets respectively, `sr_l2` and `sr_l3` to send and receive packets of the same flow, and `sniff` to collect incoming packets on a probe's physical network interface. A probe device also offers methods to interact with the device itself, such as accessing its routing table, or executing arbitrary commands. This allows the controller to launch more specialized software for measurements where serializing and transporting all packets towards the controller would be infeasible, such as `iperf3` for conducting bandwidth measurements.

[2] https://gitlab.com/sscherfke/aiomas

3.4 Probe Hardware Architecture

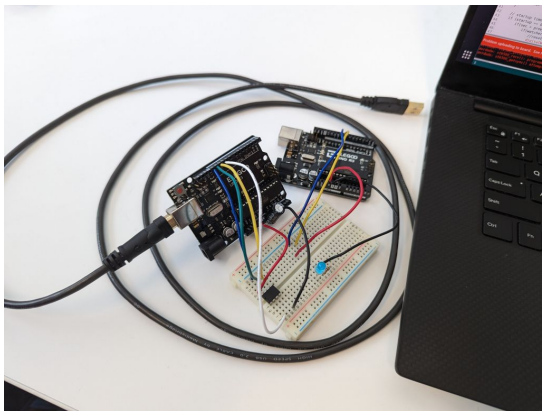

Figure 4: The prototype hardware watchdog

Our design prioritizes fully featured yet accessible hardware, with long term deployability in mind. We use Raspberry Pi 4B devices for our probes. These devices are very accessible compared to more specialized networking hardware, but are still a relevant benchmark with gigabit Ethernet. Additionally, since our code is all written in python, the probe software could easily be extended to other devices, making this solution generalizable. We then add two features to the Pi's to make them more robust to deployment. Rather than use an SD card which can fail, we net-boot the probes from an image they pull from the central controller. This incurs a larger startup cost, but eliminates a potential failure mode. We also add a hardware watchdog timer using an Attiny85, which is a low cost, Arduino compatible micro-controller. Combined with the net-booting feature, this allows a probe to automatically fully recover from a failure event. Weather it is caused

by bad code or a power surge, the hardware watchdog and net-boot allows any probe to return to a working state within minutes. This is especially important with devices deployed all across a network, so if something goes wrong they can be reset remotely rather than having to access each of them manually.

3.5 Data Aggregation

In network monitoring, it is often necessary to collect data from multiple sources, such as network probes or routers, and aggregate that data to gain a comprehensive view of network behavior. Aggregation allows us to identify patterns and trends, detect anomalies and performance issues, and make informed decisions about network management and optimization. Without aggregation, we would be limited to viewing each data source in isolation, which could lead to incomplete or inaccurate insights.

The code in Listing 2 represents a Network Graph that is constructed by gathering measurements from multiple probes. The graph contains information about IP addresses, neighbors, unreachable hops, and NAT state. The `ensure_node` method creates a node in the graph given an IP and/or hostname. The `process_traceroute_measurement` method aggregates multiple traceroute measurements and builds a graph of the network topology. This method adds neighbors and unreachable hops to the graph based on the traceroute measurements. The `traverse_node`, `add_neighbor`, and `add_unreachable_hop` methods are helper methods used by `process_traceroute_measurement` to add nodes and edges to the graph.

4 Evaluation

We begin this section with an expressiveness evaluation, followed by a discussion of the deployment of our framework on the Princeton campus data network.

4.1 Expressiveness

In this section, we demonstrate the expressiveness of our system by illustrating how it can used to implement a number of integrated network measurements. The illustrated measurement types make use of various system aspects, such as the system's flexible measurement definition interface, the ability to coordinate measurements between probe devices, as well as the powerful mechanism of executing arbitrary high-level applications directly on the probe devices.

Listing 3 outlines the complete code for the definition of an iterative ICMP-based traceroute implementation, between all pairs of probes registered with the system. It manages to express advanced features, such as a parallel execution of all measurements on all probes through the use of asynchronous APIs. Furthermore, it avoids issuing excessive spurious network packets which probe for TTLs larger than the hop-count up to the final destination, by performing step-wise increases of the sent-packets' TTL values (in this case, increasing by 5 hops each iteration). As apparent from this code excerpt, our framework is able to concisely express measurements of significant complexity.

To demonstrate the ability to coordinate measurements between multiple probe devices, we devise a simple scheme to detect the presence of Network Address Translation (NAT) devices between two probe devices. As shown in Listing 4, this measurement utilizes the fact that a packet can be exchanged between two probes, where one probe transmits the packet and another records incoming network traffic. Holding a copy of both the sent and received packets, the software can determine whether a NAT was present on this end-to-end path; in this case, the source IP or the source port of the received packet would have been changed.

Finally, we show the capabilities enabled by being able to execute arbitrary commands on probe devices. Listing 5 shows a measurement which executes the `avahi-browse` command on probe devices to discover announced services by Multicast-DNS (mDNS) DNS Service Discovery (DNS-SD) enabled devices. The controller can collect this information from multiple probe devices, acting as network vantage points. From this information, we can detect the present of mDNS *zoning* in a network: for performance and isolation reasons, networks may filter mDNS Multicast traffic even within a single broadcast domain. By correlating information gathered from multiple probe devices in a single broadcast domain but multiple mDNS zones, we can deduce the presence of such mechanisms.

4.2 Princeton Campus Deployment

We currently have 4 probes deployed on Princeton's network. One in Friend Center, one in EQuad, one in Sherrerd Hall, and one in the Graduate College. Additionally, we have a probe external to Princeton's network in a datacenter located in Virginia. The central controller is hosted on a server in the Computer Science building's datacenter. Each of these probes is provisioned through a process called *net-booting*. We found this to be a sufficient number of devices, as it allowed us to perform measurements between 3 subnets on Princeton's campus, and investigate the mDNS *zoning* infrastructure on the shared subnet between Friend Center and EQuad.

4.3 Data Visualization

We make use of the Jaal [3] framework for network visualization. Jaal is a Python-based interactive network visualizing tool built on top of two underlying data visualization frame-

[3] https://github.com/imohitmayank/jaal

works, `Dash`[4] and `Visdcc`[5]. `Dash` provides basic visualization functionality through a web server. `Visdcc` on the other hand provides additional visualization components, such as interactive maps, and advanced styling options, that can be used to create dynamic and engaging visualizations.

Jaal provides a simple Python-based interface for network visualization. The input is provided as 2 CSV files, one each for the attributes of the nodes and edges, which form the overall network. The node CSV file requires an `id` attribute, which identifies the node. In our setting, the `id` is the IP address of the network device. The edge CSV requires two attributes, `from, to`, which defines the connections between the nodes. Both `from, to` attributes must be a part of the set of `id`'s defined in the nodes CSV. The node and edge CSV files can define any number of additional numerical or categorical attributes associated with a node or edge. The CSV files need to be serialized into 2 distinct `Pandas DataFrame` objects before being instantiated with the Jaal framework.

Jaal creates an interactive network visualization of the nodes and edges, provided as an input. There are 4 features exposed which allows the user to interact with the network:

Search : search for a node or an edge based on attributes using Pandas syntax

Filter : only visualize select nodes and edges

Color: nodes and edges can be colored based on categorical attributes

Size: nodes and edges can be sized based on numerical attributes

We instantiate Jaal using the aggregated network measurements described in Section 3.5. In addition to the required attributes for the nodes and edges, we define the following additional attributes to aid visualization:

nodes: `nat ("unknown", "likely"), probe (0,1)`

edges: `type ("direct", "unreachable")`

We provide a sample visualization in Figure 5. The 6 large yellow nodes represent our probe devices - 3 probes connected to Ethernet, 1 probe connected to Ethernet and Wifi, 1 external probe on the Hetzner server. The purple nodes indicate likely NAT devices. The small yellow nodes represent intermediate routers or end hosts.

5 Conclusion

In this project, we have developed and deployed a modular framework to perform active network measurements through the use of Raspberry Pi probes. Our current framework supports traceroute and ping as primitive measurements. Further, we demonstrate the ability to perform coordinated measurements by composing multiple traceroute measurements for the detection of NATs. By doing so, our work builds on and contributes to literature which aims to provide granular and flexible network measurements.

We envision future work and extensions of our work to involve the following:

- Reverse engineer arbitrary network topologies by identification of further types of network devices.
- Make the aggregation strategy more flexible, expressive and intelligent to reduce data redundancy.
- Real-time updating of network visualization and integration with our existing framework
- Automated fault detection. By running routine measurements and comparing against baselines, a network administrator could automatically detect bottlenecks or link failures as it pertains to the user.
- Automated vulnerability testing. The system can be used to automatically test the presence of firewalls, and verify that security-relevant middleboxes work as intended.

[4] https://plotly.com/dash/

[5] https://github.com/jimmybow/visdcc

References

[1] Bajpai, V., Eravuchira, S. J., and Schönwälder, J. Lessons learned from using the ripe atlas platform for measurement research. *ACM SIGCOMM Computer Communication Review 45*, 3 (2015), 35–42.

[2] Clark, D. D., Bauer, S., Lehr, W., Claffy, K., Dhamdhere, A. D., Huffaker, B., and Luckie, M. Measurement and analysis of internet interconnection and congestion. In *2014 TPRC Conference Paper* (2014).

[3] Detal, G., Hesmans, B., Bonaventure, O., Vanaubel, Y., and Donnet, B. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 conference on Internet measurement conference* (2013), pp. 1–8.

[4] Guilbaud, N., and Cartlidge, R. Google—localizing packet loss in a large complex network. *Feb 5* (2013), 1–43.

[5] Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 139–152.

[6] Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., and McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)* (2014), pp. 71–85.

[7] Jin, L., Hao, S., Wang, H., and Cotton, C. Understanding the practices of global censorship through accurate, end-to-end measurements. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 5*, 3 (2021), 1–25.

[8] McNamara, L., Marsh, I., and Forslin, S. Cheesepi: A raspberry pi based measurement platform. In *IRTF & ISOC Workshop on Research and Applications of Internet Measurements (RAIM 2015), October 31, 2015, Yokohama, Japan* (2015).

[9] RAMAN, R. S., WANG, M., DALEK, J., MAYER, J., AND ENSAFI, R. Network measurement methods for locating and examining censorship devices. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies* (2022), pp. 18–34.

[10] RIPE NCC STAFF. Ripe atlas: A global internet measurement network. *The Internet Protocol Journal 18*, 3 (2015), 2–26.

A Appendix

```python
class NetworkGraph:
    graph = {}

    # create node given an IP and/or hostname
    def ensure_node(self, ip,    probe_hostname=None):
        if ip not in self.graph:
            self.graph[ip] = {
                "ip": ip,
                "probe_hostname": probe_hostname,
                "neighbors": set(),
                "unreachable_hops": set(),
                "nat_state": 0,
                "traversed": False,
            }
        else:
            self.graph[ip]["probe_hostname"] =
            probe_hostname

    # aggregation strategy of traceroute measurements
    def process_traceroute_measurement(self, m):
        # Add src probe as first hop:
        self.ensure_node(m.tracert_hops[0],
                         m.src_probe)

        prev_hop = m.tracert_hops[0]
        skipped_hops_count = 0
        for ip in m.tracert_hops[1:]:
            if ip is not None:
                self.ensure_node(ip)
                self.add_neighbor(prev_hop, ip,
                                  skipped_hops_count)
                self.traverse_node(ip)
                prev_hop = ip
                skipped_hops_count = 0
            else:
                skipped_hops_count += 1

        self.ensure_node(m.dst_ip, m.dst_probe)
        if m.dst_reached:
            self.add_neighbor(prev_hop, m.dst_ip,
                              skipped_hops_count)
        else:
            self.add_unreachable_hop(prev_hop,
                                     m.dst_ip)

    def render_csv(self):
        ...
```

Listing 2: An implementation of an aggregation strategy for traceroute measurements. The `process_traceroute_measurement` method aggregates traceroute measurements. The `ensure_node` method creates a node given an IP or hostname. The class may be extended to support arbitrary number of measurements and aggregation strategies.

```python
async def traceroute(probe_a, probe_b):
    if probe_b[0] is not None:
        print(f"Traceroute {probe_a[0]} -> {probe_b[0]}...")
        probe_b_ip = await probe_b[1].primary_ip()
    else:
        print(f"Traceroute {probe_a[0]} -> {probe_b[1]}...")
        probe_b_ip = probe_b[1]

    # Perform an iterative traceroute to avoid flooding
    # the network:
    max_ttl = 32
    current_ttl = 1
    ttl_step = 5
    tracert_hops = [None for _ in range(0, max_ttl)]
    dst_reached = False
    while not dst_reached and current_ttl < max_ttl:
        ttl_range = (current_ttl, current_ttl + ttl_step - 1)
        ans, unans = await probe_a[1].sr(
            IP(dst=probe_b_ip, ttl=ttl_range) / ICMP(),
            as_thread=True, timeout=2, retry=3)
        current_ttl += ttl_step

        for qa in ans:
            idx = qa.query[IP].ttl
            src = qa.answer[IP].src
            tracert_hops[0] = qa.query.src
            if src == probe_b_ip:
                tracert_hops = tracert_hops[:idx]
                dst_reached = True
                break
            else:
                tracert_hops[idx] = src

    return TracerouteProbeMeasurementResult(
        src_probe = probe_a[0],
        max_ttl = max_ttl,
        dst_ip = probe_b_ip,
        dst_probe = probe_b[0],
        tracert_hops = tracert_hops,
        dst_reached = dst_reached,
    )

# Build all 2-permutations of probes currently registered
# and make them traceroute towards each other:
measurement_perms = list(
    itertools.permutations(self.probes.items(), 2))
measurement_jobs = [
    traceroute(probe_a, probe_b)
    for (idx, (probe_a, probe_b)) in enumerate(measurement_perms)
]

# Await all measurements to execute them simultaneously:
results = await asyncio.gather(*measurement_jobs)

for r in results:
    self.coordinator.ng.process_traceroute_measurement(r)
```

Listing 3: Implementation of an interative ICMP-based traceroute between probe devices. This measurement makes use of the asynchronous APIs to run measurements of individual probes in parallel.

```python
async def check_nat(probe_a, probe_b):
    print(f"Check NAT {probe_a[0]} -> {probe_b[0]}...")
    probe_b_ip = await probe_b[1].primary_ip()
    probe_a_ip = await probe_a[1].lookup_ipv4_route(probe_b_ip)

    # Start a sniff task on probe B:
    sniff_task = asyncio.create_task(
        probe_b[1].sniff(filter="udp port 1337", timeout=5))

    # Send a packet from probe A to B to UDP port 1337 with
    # a known payload to identify this measurement:
    await probe_a[1].send(
        IP(src=probe_a_ip, dst=probe_b_ip)
              / UDP(sport=udp_port, dport=udp_port)
            / "netmonpi-nat-probe")

    # Wait for the sniff to end
    sniffed = await sniff_task

    for p in sniffed:
        if UDP in p \
           and p[UDP].dport == udp_port \
           and isinstance(p[UDP].payload, Raw) \
           and p[UDP].payload.load == b"netmonpi-nat-probe":
            return NatMeasurementResult(
                src_probe = probe_a[0],
                src_ip_snd = probe_a_ip,
                src_ip_rcv = p[IP].src,
                dst_probe = probe_b[0],
                dst_ip = probe_b_ip,
                method = "udp",
                udp_src_port_snd = udp_port,
                udp_src_port_rcv = p[UDP].sport,
                udp_dst_port = udp_port,
                nat_detected = (
                    p[IP].src != probe_a_ip \
                    or p[UDP].sport != udp_port
                )
            )

        # Packet not received, no information
        return None

# Build all 2-permutations of probes currently registered
# and check whether there is a Nat on the path in between:
measurement_perms = list(itertools.permutations(self.probes.items(), 2))
measurement_jobs = [
    check_nat(probe_a, probe_b, method="udp")
    for (idx, (probe_a, probe_b)) in enumerate(measurement_perms)
]

# Await all measurements to execute them simultaneously:
results = await asyncio.gather(*measurement_jobs)

for r in results:
    self.coordinator.ng.process_nat_measurement(r)
```

Listing 4: Implementation of a primitive Network Address Translation (NAT) detection mechanism. This example illustrates how measurements can be carried out and coordinated between probes connected to the system: first, a recipient probe records incoming network traffic through the `sniff` method. Then, the sending probe transmits a UDP packet with known headers and payload. Finally, the received packet's header values are inspected. If either the source address or port are changed, this likely indicates an on-path NAT device.

```python
async def query_mdns(probe):
    retcode, stdout, stderr = \
        await probe[1]._c.remote.exec_command(
            "avahi-browse -pt _adapi-http._tcp " \
            + "| cut -d';' -f4")

    announced_services = list(map(
        lambda record: record.strip().lower(),
        stdout.strip().split("\n")
    ))

    return MDNSZoningResult(
        src_probe=probe[0],
        src_ip=(await probe[1].primary_ip()),
        dnssd_query="_adapi-http._tcp.local",
        announced_services=announced_services,
    )

measurement_jobs = [
    query_mdns(probe)
    for probe in self.probes.items()
]

for probe in self.probes.items():
    res = await query_mdns(probe)
    self.coordinator.graph.\
        process_mdns_zoning_measurement(res)
```

Listing 5: Implementation of a measurement to capture surrounding mDNS DNS-SD announced services offered by other network devices. This measurement demonstrates the capability of the controller to execute arbitrary applications on the probe devices themselves. In practice, this measurement is used to detect mDNS *zoning* in networks: for performance and isolation reasons, Multicast mDNS traffic may be limited even within a single broadcast domain. Our system can capture mDNS announcements from multiple vantage points, and identify such isolation mechanisms by correlating data from different probe devices.

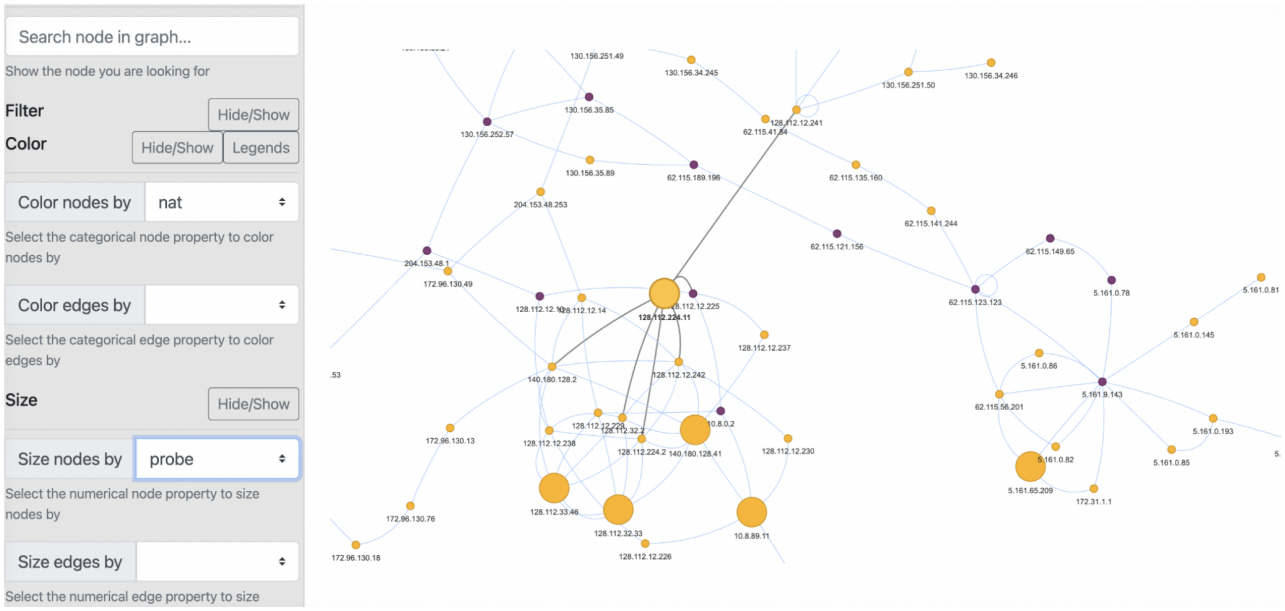

Figure 5: Sample Network Visualization consisting of 6 probes (large yellow nodes), likely NAT devices (purple nodes), intermediate nodes or end hosts (small yellow nodes)