Packet Tracing with P4: Implementation, Challenges

Saketh Vishnubhatla
School of Computing and Augmented Intelligence
Arizona State University
svishnu6@asu.edu

***Abstract*—This project explores an implementation of packet tracing using a P4 implementation. Current packet tracing algorithms have limitations in terms of readability, protocol-dependence, and flexibility which network programmability can address. A P4-based working implementation on FABRIC is provided in this project, which is tested on two different topologies. Some challenges in scaling this implementation are discussed with possible future extensions. Code snippets for the implementation are provided for reproducibility [1].**

***Index Terms*—P4, INT, FABRIC, traceroute, packet tracing**

I. Introduction

It is often important to trace the route a packet takes to understand the network infrastructure and addresses. Network administrators and penetration testers use packet tracing algorithms to identify the routes along which they observe data transmission failures. This also helps in understanding the delays across different routers and also points of failure responsible for the packet losses. Many operating systems have network diagnostic tools like traceroute, tracert for running packet route tracing routines. However, there are multiple limitations with these tools [2].

The emergence of the P4 [3] (Programming Protocol-Independent Packet Processors) paradigm has ushered in a new era of flexibility and programmability. The paper proposes three goals for switches to process packets: reconfigurability, protocol-independence, and target independence. The core idea is that headers on top of existing packets can be modified to leverage programmability in packet processing.

One of the applications of P4 is INT [4] (In-Band Network Telemetry). With programmable switches, it is possible to get fine-grained telemetry about congestion, latencies, and queue occupancies. While conventional monitoring tools fail to provide accurate per-packet analysis, INT has been one of the most promising applications of programmable networks. In the context of route tracing INT could be used to modify packet headers. When a packet is delivered, the INT source and the following switches will add entries corresponding to the interface addresses that the packet traverses through to the packet header. Finally, at the INT sink, the headers can be deparsed and the packet will be sent to the destination.

In this project, we aim to qualitatively investigate the advantages of using P4 for route tracing, and compare it with traditional approaches like traceroute. The biggest advantages are in terms of flexibility, protocol-independence, and interpretability.

II. Related Work

A. Background Readings

FABRIC [6] is a test bed providing users with support to request custom compute nodes (with GPUs if needed) to create networks for experimentation, through the FABLib library programmatically. FABRIC also is planning to enable integration with cloud providers like Azure so that experiments can be run in a hybrid fashion, which already exists for AWS.

INT [4] refers to the capability of providing a network state directly from the data plane without any involvement of the control plane. One possible way of implementing this is using programmable routers and switches. In [7], an extensive discussion on how P4 could be used for INT applications is presented.

P4 provides a pathway for creating programmable switches. To use P4 on FABRIC, tutorials [8] provided by the University of South Carolina were very helpful. Also while FABRIC does not provide a physical implementation of the P4 switches (which will be added soon to FABRIC), it is easy to install BMv2 [9]: a software P4 implementation is installed on the switch nodes.

The basic architecture of P4 programs comprises: programmable parser, match-action tables, and a programmable deparser. At the ingress, when a packet enters, a parser enables the user to have custom logic to accept packets based on user-defined headers and their values. Then, match-action tables are used to map an action for the packet with some user-defined header values. Finally, at the egress, a deparser helps modify the packet headers if needed and assembles the headers back onto the packet.

B. Contextual Readings

In the context of FABRIC, monitoring capabilities can be leveraged by MFLib [10]. MFLib is a measurement framework for timing latency, congestion, etc. by adding a measurement node to the existing network for monitoring. MFLib is mostly helpful in creating error dashboards in integration with tools like Kibana. One possible way to monitor packets on FABRIC is by creating MFLib services on measurement and experiment nodes, and continuously passing information out-of-band to the measurement node as the packet traverses the network. In comparison to INT-based methodologies, this approach introduces intricate complexities.

Traceroute [5] by Van Jacobson is a well-known and widely used network diagnostic program. Essentially each

time traceroute sends a number of ICMP echo requests with increasing TTL values. Once the TTL drops to zero on a node a reply is sent back to the source. An intelligent manipulation of incrementing TTL through the process, helps us estimate the number of hops it takes to reach any node. Two of the biggest challenges to using traceroute to determine paths are a) difficulty in interpreting paths when three or more possible IPs are listed for each hop (see Figure 1) and b) ICMP packets might report more delay than actually perceived by normal data plane networks. [2] also provides other limitations of traceroute: a) asymmetric paths cannot be seen, b) layer-3 implementation hiding layer-2 details, and c) poor error messaging system.

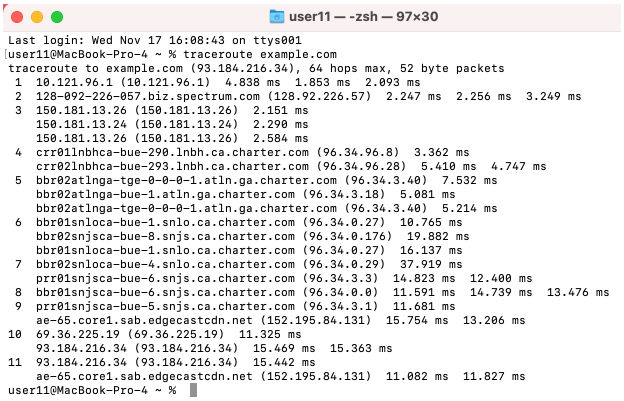

Fig. 1. A sample output of traceroute.

There have been a few works that seek to improve upon traceroute. In [11] the authors account for the limitation that traceroute only reports the path ICMP probes (or UDP probes) take. But to get the route from the destination to the source, one would need to run a traceroute from the destination server. Or assume that a similar path would be taken assuming symmetric costs. However this is rarely shown to be the case in the real-world. In this project, we would want to leverage programmability to highlight the advantages of P4 tracing over conventional approaches and have a working implementation for the same.

III. Project Description and Milestones

The project aims to implement a custom packet-tracing algorithm on the FABRIC testbed with P4. For this using P4, customized packet headers would be attached to each of the packets being sent from a node into the network. Using INT we ensure the collection of information regarding nodes traversed is embedded in the packet headers.

The basic timeline of the project was as follows:

1) Basic setup of network topology on FABRIC [**Done**]
2) Adding P4 switches on the network [**Done**]
3) Implementation of P4-based tracing routine on a small topology (2-4 weeks) [**Done**]
4) Scaling up the topology to simulate real-world scenarios [**Challenges**]
5) Future extensions

Some observations of interest in the experiment would be: a) running both P4-based-tracing and traceroute over the network topology, b) outputs as indicated with both the methods and c) monitoring switch information (like ingress/egress ports the packets traverse) from each intermediate hop at the sink.

IV. Experimental Setup and Design

Most of the implementations online were using the Mininet emulator, where multiple nodes would be on the same underlying hardware. But to account for latencies between different hops and their resemblance to real-world scenarios, nodes from different sites have been chosen. For the experiments, two topologies have been used: a small topology with two server nodes and one intermediate switch node, and a large topology consisting of six server nodes with six switches.

Each of the servers is created with 'default_ubuntu_20' images with 8 GB RAM, 20 GB disk space, and 4 cores. Similarly, each of the switches also is created with 'default_ubuntu_20' images but with 16 GB RAM, 40 GB disk space, and 32 cores. This is because software implementation of P4 switches has higher requirements of memory and disk space. Also, it is worth mentioning that very large topologies of size greater than 20 nodes, took very long and returned errors. Hence the current topology with a total of 12 nodes has been chosen for the large topology.

A. Small Topology

A small topology as shown in Figure 2 has been used to test the packet tracing experiment. The setup consists of two servers, a sender and a receiver along with one intermediate switch node. The subnets and network interfaces are assigned as shown in the figure. In this setup a packet is crafted on the sender using Scapy (a Python packet-crafting library) and the packet is received on the other server, via a hop over the switch. Details on the results are discussed in the section V.

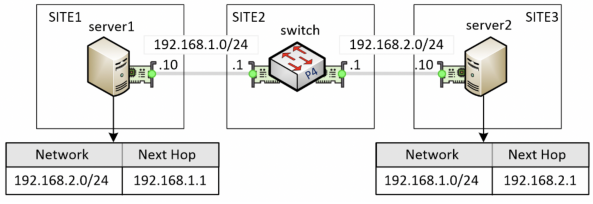

Fig. 2. A small topology is instantiated on FABRIC adapted from the UofSC tutorials [8].

B. Large Topology

For the large topology, a server-switch pair is created in six different sites: LOSA, SALT, STAR, ALTA, DALL, and KANS. Figure 3 shows how these nodes at different sites are connected. A more detailed snapshot of the network is given in Figure 4 and in the appendix. For testing path tracing implementation, we label site 1 (LOSA) as the source and site 3 (STAR) as the destination for observing the trace.

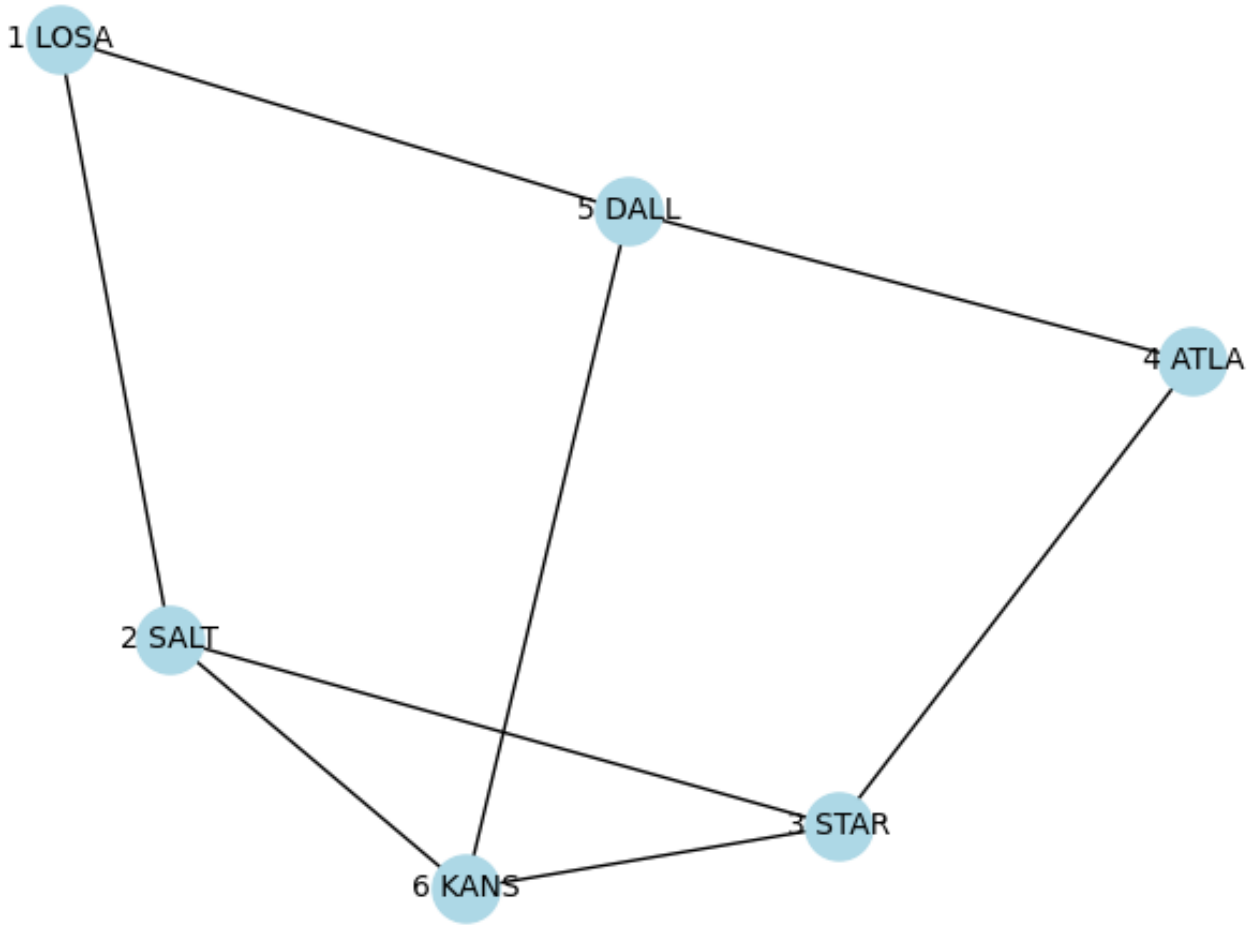

Fig. 3. Simplified diagram showing the connectivity between different sites. In each site, one server and one P4 switch have been created and connected with other sites as shown.

An outline of the proposed implementation is given in Algorithm 1. This program runs on all the six P4 switches. Whenever a packet passes, it is checked for corruption. Then a check is performed to see if this is the last hop in the traversal, in which case the path information is extracted from the header and returned. Otherwise, the packet header is simply modified to add the current interface address.

V. Implementation and Results

A P4 implementation of the setup is provided. As shown in Figure 8 the P4 headers consist of ethernet and ipv4 fields. It also contains ipv4 options field with an option value set to 31, for multi-hop route inspection (MRI) support. This enables us to have switch traces within the IPv4 options field.

The basic workflow of the P4 routine is as follows. A packet enters the parser, which parses each of the IPv4, ethernet, and MRI fields from the header. The trickier part is to recursively iterate through the packet searching for switch traces until there are no more traces left. This would be simpler if P4 routines allowed to initialize simple counter variables which is not the case. Hence an extra header field from the 'parser_metadata_t' is used to keep a count of the number of

Algorithm 1 P4 Packet Tracing

$Pckt \leftarrow$ Incoming Packet
$Pckt.nextAddr \leftarrow$ From match-action table
▷ Next IPv4 address to be filled from match-action table
if $Pckt$ is corrupt **then**
    return CORRUPT FLAG
else if $Pckt.dstAddr \neq Pckt.nextAddr$ **then**
    $Pckt.path+ = CurrIfaceAddr$
else if $Pckt.dstAddr == Pckt.nextAddr$ **then** ▷ This is INT Sink
    $Pckt.path+ = CurrIfaceAddr$
    return Pckt.path
▷ The switch trace could contain other metadata as well other than iface addresses
end if

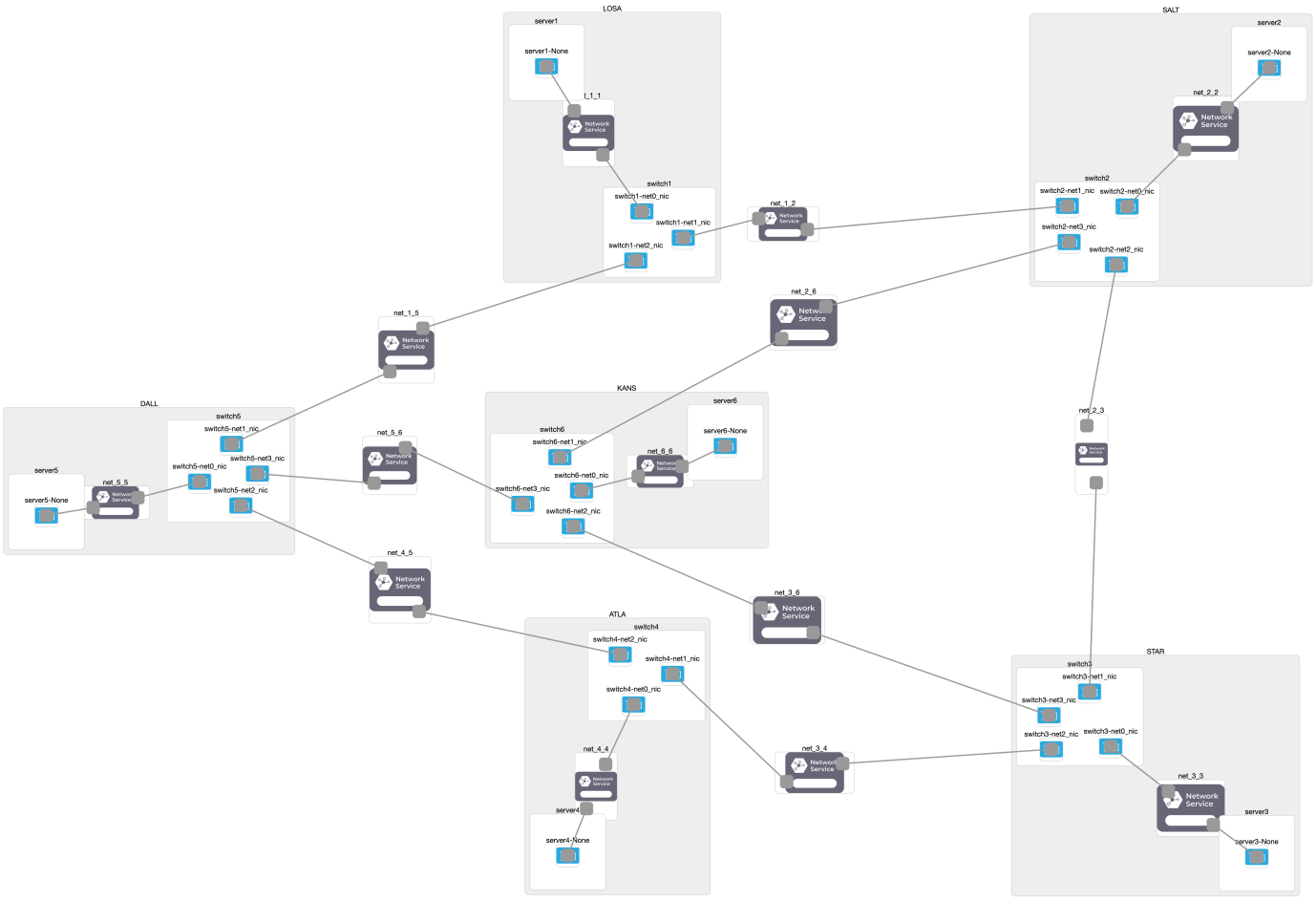

Fig. 4. Large experiment topology created on FABRIC comprising six switches and six servers from six different sites.

traces extracted by the parser. Once it hits zero, the parsing is completed.

Next is the ingress pipeline where an exact and longest-prefix matching is used to forward the packets. In this step header fields like TTL, and egress port are modified. After ingress, the packet traverses to the egress pipeline where the MRI count (indicating the number of switch hops) is incremented, along with writing the switch traces like ingress and egress port addresses to the packet headers. The code for each of these pipelines is provided in the repository.

For crafting the packets, Scapy has been used. The snapshots shown indicate the packet that has been crafted on the sender (Figure 5) and the received packet on the receiver (Figure 6). We can notice that while at the sender there is an empty switch trace it is populated at the receiver's end. While tcpdump can be used to capture packets, to display the header information in the required format a sniff tool provided by Scapy is used on the receiver's end to monitor for the incoming packets.

VI. Discussion

Traceroute only provides information about interface addresses. An output from a traceroute usually contains the IP addresses of all the network interfaces through which a packet has traversed. However, in some cases, we might benefit from retrieving more information. For example, the management IP of the node is useful so that a developer can easily SSH into the node and further debug. This is easy with P4 given that this would mean one additional packet header modification adding management IP.

Traceroute is tightly integrated with transport protocols. Given that tracert on Windows uses ICMP, while traceroute on linux uses mostly UDP/TCP segments depending on the routers configuration and firewall settings some of the datagrams might be dropped. This is not the case with P4 tracing, given that they are programmable and not closely tied to the underlying protocols.

Traceroute output is not very readable. In some cases, it is difficult to interpret the path when we send two or more probes with a traceroute. Often we would want to get one specific path, which isn't the exact structure followed by traceroute output.

The discussions in the class after the project presentation provided many interesting insights. For example one student's question: "What would happen to the packet size if the packet traverses many switches? Wouldn't the increase in packet size be a problem?". Usually, the number of hops a normal packet

```
sending on interface enp6s0 to 192.168
###[ Ethernet ]###
  dst       = 00:00:00:00:00:02
  src       = 00:00:00:00:00:01
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 6
     tos       = 0x0
     len       = 56
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xd656
     src       = 192.168.1.10
     dst       = 192.168.2.10
     \options   \
      |###[ MRI ]###
      |  copy_flag = 0
      |  optclass  = control
      |  option    = 31
      |  length    = 4
      |  count     = 0
      |  \swtraces  \
###[ TCP ]###
        sport     = 1234
        dport     = 4321
        seq       = 0
        ack       = 0
        dataofs   = 5
        reserved  = 0
        flags     = S
        window    = 8192
        chksum    = 0xc694
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      = '192.168.1.10'
```

Fig. 5. The packet header structure at the sender's end.

```
ubuntu@server2:~$ sudo python3 receive.py
sniffing on enp7s0
got a packet
###[ Ethernet ]###
  dst       = 00:00:00:00:00:04
  src       = 00:00:00:00:00:02
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 8
     tos       = 0x0
     len       = 64
     id        = 1
     flags     =
     frag      = 0
     ttl       = 63
     proto     = tcp
     chksum    = 0xf452
     src       = 192.168.1.10
     dst       = 192.168.2.10
     \options   \
      |###[ MRI ]###
      |  copy_flag = 0
      |  optclass  = control
      |  option    = 31
      |  length    = 12
      |  count     = 1
      |  \swtraces  \
      |   |###[ SwitchTrace ]###
      |   |  ingress_port= 0
      |   |  egress_port= 1
###[ TCP ]###
        sport     = 1234
        dport     = 4321
        seq       = 0
        ack       = 0
        dataofs   = 5
        reserved  = 0
        flags     = S
        window    = 8192
        chksum    = 0xc694
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      = '192.168.1.10'
```

Fig. 6. The packet header structure at the receiver's end.

takes over the internet is less than 20 to 30 hops. However, this won't be the case if the network configuration is erroneous, where the packet might be stuck in a loop. In this case, the packet traverses until the TTL reaches zero. Given that the TTL usually is set to 64 or 256. However, given that the metadata is usually a few bits this intuitively should not cause an issue.

Also, the execution of the routines on the larger topology proved to be challenging. The biggest challenges include manual subnetting of large networks which is error-prone, and difficult error detection when operating on multiple nodes. Also, the header fields in the P4 program should be a multiple of 8 bits. These have created some challenges for the project. For creating larger topologies, in the class discussion, it was suggested to use the "auto-configuration" option on the FABRIC testbed. This is to be checked to see the ease of instantiating larger topologies on FABRIC, which in itself could provide another interesting direction for future exploration.

ACKNOWLEDGEMENT

We would like to thank Dr. Violet Syrotiuk for providing the students with a great opportunity to run experiments on the FABRIC testbed and facilitating continuous feedback throughout the project development. Also, the comments from anonymous reviewers and feedback from the students were invaluable in improving the project.

REFERENCES

[1] Vishnubhatla, S. (n.d.). Sak-18/ASU-CSE-534: P4 based packet tracing on fabric. GitHub. https://github.com/sak-18/ASU-CSE-534

[2] Fabien Viger, Brice Augustin, Xavier Cuvellier, Clémence Magnien, Matthieu Latapy, Timur Friedman, Renata Teixeira, "Detection, understanding, and prevention of traceroute measurement artifacts", Computer Networks, Volume 52, Issue 5, 2008, Pages 998-1018, ISSN 1389-1286, https://doi.org/10.1016/j.comnet.2007.11.017.

[3] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J. L., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D. P. (2014). P4: Programming protocol-independent packet processors. Computer Communication Review, 44(3), 87-95. https://doi.org/10.1145/2656877.2656890

[4] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J.Wobker, "Inband network telemetry via programmable dataplanes," in Proc. ACM SIGCOMM, 2015, pp. 1.2.

[5] V. Jacobson, Traceroute https://datatracker.ietf.org/doc/html/rfc1393

[6] I. Baldin et al., "FABRIC: A National-Scale Programmable Experimental Network Infrastructure," in IEEE Internet Computing, vol. 23, no. 6, pp. 38-47, 1 Nov.-Dec. 2019, doi: 10.1109/MIC.2019.2958545.

[7] Elie F. Kfoury, Jorge Crichigno and Elias Bou-Harb: An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. CoRR abs/2102.00643 (2021).

[8] Cybertraining: Virtual platform deployed for cybertraining purposes," http://ce.sc.edu/cyberinfra/cybertraining.html, accessed April 8th, 2023.

[9] Bas, A. (2022, February 10). Behavioural-modelVersion (v2). BEHAVIORAL MODEL (bmv2). Retrieved October 22, 2023, from https://github.com/p4lang/behavioral-model.

[10] Carepenter, C. (2023, January 31). MFLib - Measurement Framework on FABRIC. MFLib. https://github.com/fabric-testbed/mflib

[11] Ethan Katz-Bassett, Harsha V. Madhyastha, Vijay Kumar Adhikari, Colin Scott, Justine Sherry, Peter Van Wesep, Thomas Anderson, and Arvind Krishnamurthy. 2010. Reverse traceroute. In Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI'10). USENIX Association, USA, 15.

APPENDIX

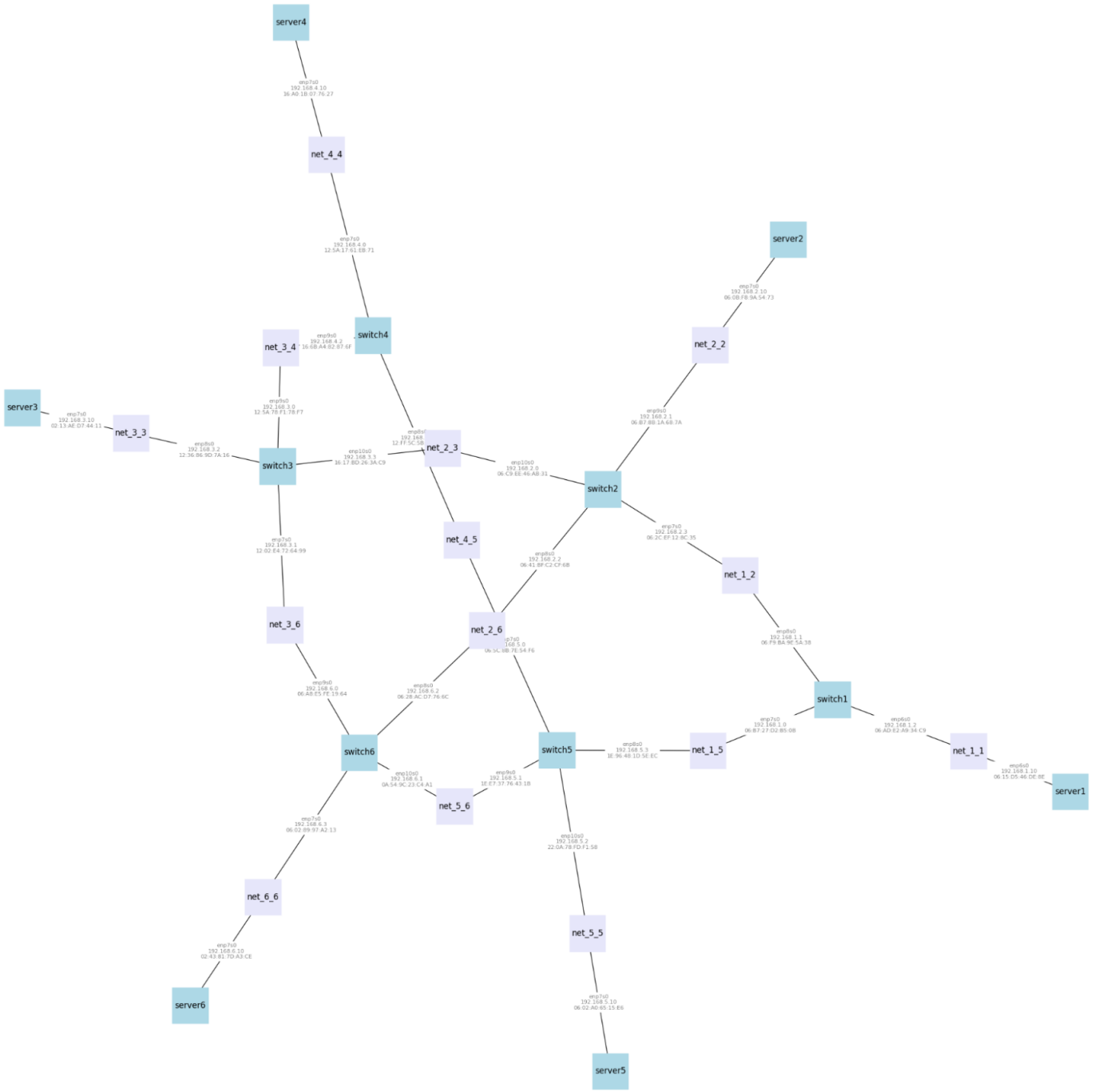

Fig. 7. Complete Topology of servers and P4 switches with their addresses.

```
// ethernet header fields
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

// ipv4 header fields
header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

// ipv4_options
// Set option value to 31 for embedding switch traces as part of the ipv4 header
header ipv4_option_t {
    bit<1> copyFlag;
    bit<2> optClass;
    bit<5> option;
    bit<8> optionLength;
}

// MRI- Multi-Hop Route Inspection
// This field maintains a count of the number of switch hops
header mri_t {
    bit<16>  count;
}
```

```
//switch_t header: The details to be collected from each switch
header switch_t {
    ingress_port_t ingress_port;
    egress_port_t egress_port;
}

// The below two headers are used as counter variables
// P4 doesn't allow a counter variable to be initialized within a routine
struct ingress_metadata_t {
    bit<16>  count;
}

struct parser_metadata_t {
    bit<16>  remaining;
}

//contains metadata from the ingress and parser
struct metadata {
    ingress_metadata_t   ingress_metadata;
    parser_metadata_t   parser_metadata;
}

// FINAL parent header structure to be crafted on Scapy
struct headers {
    ethernet_t         ethernet;
    ipv4_t             ipv4;
    ipv4_option_t      ipv4_option;
    mri_t              mri;
    switch_t[MAX_HOPS] swtraces;
}

//an error definition
error { IPHeaderTooShort }
```

Fig. 8. Implementation: P4 headers