RU-CloudNet: P4-Virtualized Network Data Plane

CS 552 Computer Networks

Due March 18th, 2024

In this assignment, you will build your own P4-based "cloud" network data plane that leverages programmable switches to implement network virtualization among tenant hosts, including address space isolation and a degree of performance isolation among the hosts.

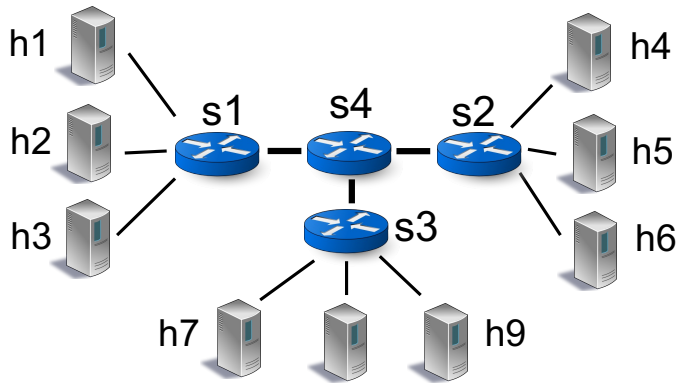
Let's build RU-CLOUDNET! First, we will walk through the tools you will need to run and understand the behavior of P4-programmable switches (§1), and then jump into the features you need to complete your implementation of RU-CLOUDNET (§2).

Getting set up with the VM. The VM for this project is the same as the last project. For project 1, we already provided instructions to access the VM in a separate document. Please consult the corresponding Canvas announcement.

1 Understanding and Controlling P4 switches

P4 is a language to program the data plane of switches [1]. There are different kinds of programmability. Primarily, P4 concerns itself with providing flexibility in packet header formats and table structures. While fixed-function routers of the late 2010s primarily supported parsing and forwarding standard packet header formats (such as Ethernet/IPv4/TCP), and standard table formats that supported forwarding depending on the fixed meanings of the fields in those headers, P4 switches can be programmed not only to parse and emit packets with header formats that you invent, but also forward packets depending on the meanings you assign to those invented packet fields.

The network topology. We will be playing with a fixed topology consisting of 3 *edge switches* S1–S3 and one *fabric switch* S4 shown below. The reason we call these switches edge or fabric will become clear in §2, but at this time it is sufficient to understand that edge and fabric switches do qualitatively different things. Each edge switch is directly connected to 3 hosts. The fabric switch is connected directly to the edge switches, but not to any hosts directly.



P4 programs for switches. Each switch S1–S4 is programmable using P4. We have provided identical starter code for both in the files `edge.p4` and `fabric.p4`. However, in §2 you will implement different functionality for each. You can compile these programs to an underlying representation that can be consumed by the switches using the following commands:

```
p4c -b bmv2 edge.p4 -o edge.bmv2
p4c -b bmv2 fabric.p4 -o fabric.bmv2
```

Spend some time understanding the structure of the P4 program provided. Each switch is defined by a combination of 5 things, a parser, an ingress pipeline, checksum update/verification modules, an egress pipeline, and a deparser (definition `main` at the end). In this assignment, we will mainly concern ourselves with the parser, deparser, and the ingress pipeline. The significance and meanings of the individual modules hopefully is (or will become) clear from lecture, but, at a high level: (i) the header definition and parser allow you to understand any packet format you want (including invented ones that differ significatly from regular TCP/IP), (ii) a deparser which will allow you to put parsed or constructed headers back together in an order you choose; (iii) the ingress pipeline implements match-action table processing on the headers extracted by the parser. You can also understand these concepts deeper from the papers on P4 [1], RMT [2], and P4 tutorials/talks such as `https://github.com/p4lang/tutorials/blob/master/P4_tutorial.pdf`. The starter code defines the structure of the Ethernet and IPv4 headers, and sets up a table that implements IPv4 longest-prefix matching and forwarding. Note that nothing is assumed about standard TCP/IP: even parsing and forwarding packets from "standard" protocols have to be defined explicitly in P4. P4 switches can be taught to understand these and other formats.

Instantiating the topology. You can instantiate the network topology above in mininet using the command:

```
sudo python3 star_topo.py --behavioral-exe simple_switch \
  --edge_json edge.bmv2/edge.json \
  --fabric_json fabric.bmv2/fabric.json
```

This script starts edge switches S1–S3 with the program in `edge.p4`, and the fabric switch S4 with the program in `fabric.p4`. The script will print several useful pieces of information, like the IPv4 addresses of the host interfaces, and the commands used to invoke the various P4 switches. You can play around within mininet in the command line interface (CLI) that comes up. Mininet provides custom *network-level commands* executed over the entire network. For example, `net` will print the connectivity pattern in the network. In the output, `h1-eth0:s1-eth1` means that interface `eth0` on host `h1` is connected to interface `eth1` on switch `s1`. Mininet also allows you to execute arbitrary *host-level commands* on any host in this network. For example, `h1 ls -a` will list all files in the current working directory on `h1`, and `h4 ifconfig -a` will list information about the network interfaces on `h4`. Every host or switch-level command requires prefixing the name of the corresponding host/switch to the command, otherwise it would be interpreted as a network-level command: since those are limited in number, you would often get an error. You may find it helpful to walk through a basic Mininet tutorial at `https://mininet.org/walkthrough/` to get an idea of the commands and conveniences offered by Mininet in manipulating many network nodes from one CLI.

The P4 runtime: Control Plane. A P4 program describes the *possible* data plane behaviors of a P4 switch: the ultimate behavior of the switch is determined by the combination of the data plane and its control plane rules, which are defined through run-time configurations of the switch.

You can send rules to each P4 switch from the control plane through a CLI interface, also known as the P4 runtime. Our P4 software switch, `bmv2`, receives commands from the control plane on a port that is fixed for each switch. In our topology, these ports are configured by the Mininet script to be `9090 (s1), 9091 (s2), 9092 (s3),` and `10101 (s4)`. You can access the control plane of, for example `s2`, by running (in a separate terminal; do not quit mininet)

```
simple_switch_CLI --thrift-port 9091
```

You can access other switches similarly. Once in the control plane, you can see the full roster of commands by typing `help`. You can get help on specific commands by typing `help <command>`. For example, `table_add` is a particularly helpful command that is useful to program rules into a P4 table, and you can get specific help on this command by typing `help table_add`. The command `table_dump` will print all rules programmed into a given table, along with their "handles"; `table_delete` can delete specific entries using the corresponding rule handle as input; `table_clear` will delete all rules in a table. You can also query information about the switch through commands such as `show_ports` and `show_tables`. Play around with commands here to get familiar with the control plane interface. You will notice that all tables on all switches are empty, since we have not yet programmed any data plane rules.

Understanding packet behavior as processed by switches. The P4 software switch we are working with (`bmv2`) provides two additional very useful hooks to debug programs.

The first diagnostic is *event logs*, which is a detailed log of the most relevant packet processing in the P4 data plane for every packet. We can use the program `nanomsg_client.py` to access this log for each switch separately. You need to install the python3 module `nnp4` first (just once per VM) using the command `sudo pip3 install nnp4`. Then, you can access the detailed event log of the switch, say `s1`, by typing (in a separate shell)

```
sudo python3 nanomsg_client.py --thrift-port 9090
```

The second diagnostic is *per-interface packet captures.* You can optionally enable this through `--pcap_dump` option when starting the Mininet script, e.g.,

```
sudo python3 star_topo.py --behavioral-exe simple_switch \
  --edge_json edge.bmv2/edge.json \
  --fabric_json fabric.bmv2/fabric.json \
  --pcap_dump
```

This will produce logs containing PCAP-formatted packet captures from each switch interface in each direction. Logs are stored in the current working directory named in the format

`<switch>-<interface>_[in|out].pcap`

These logs can be viewed through tools such as Wireshark or tcpdump, e.g.,

```
tcpdump -XX -r s1-eth1_in.pcap
```

Sending test packets. Mininet allows you to send packets from the Mininet CLI. For example, you can use the `ping` tool, typically used to test network reachability (e.g., you can check reachability to `google.com` on a terminal on a Windows/MAC/Linux machine by typing `ping google.com`). Observe what happens when you type (on the mininet CLI)

```
h1 ping -c 1 h4
```

There will be no response, and `ping` will report 100% packet loss. We will fix this in §2. For now, note that despite the option `-c 1` supplied to `ping`, asking it to send 1 ICMP probe, the underlying network stack sent more than 1 packet, involving address discovery protocols such as ARP. Read up on the need and utility of the ARP protocol. You can see these packets in the packet capture on the interface `s1-eth1` on `s1` directly connected to `h1`. You can also see the packet handling on `s1` from the event logs, which reports that tables are missed on every packet, and consequently every packet is dropped.

To avoid additional packets (that we didn't initially intend) to be sent out while testing, the project also supplies a program `send_and_receive.py`, that sends one custom crafted packet per configurable interval of time. This script uses `scapy`, an extremely helpful library that may be used to craft custom packets. Familiarize yourself with this script and the ways in which packet fields can be customized using `scapy`. The script also allows you to send or receive packets on specific interfaces. You can test its behavior over the mininet CLI using

```
sh python3 send_and_receive.py 1
```

which transmits one customized IP packet every 1 second. With such customized transmissions, you will be able to precisely craft inputs to your P4 switch, and then use the event logs and packet captures to debug your data plane and control plane.

2 Building RU-CLOUDNET

RU-CLOUDNET requires building up a virtualized cloud network for hosts running with their own IP subnets in our Mininet-based network. At a high level, this requires building three features:

(1) *One Big Switch:* Each host is in a specific IPv4 network (prefix), and can communicate with other hosts in that IPv4 network without worrying about how packets outside the two endpoints are forwarded or formatted. We will assume hosts in each IPv4 network are from a single tenant/customer. Hence, with the provided topology, `h1;h4;h7` belong to one tenant; `h2;h5;h8` belong to a second tenant; and `h3;h6;h9` are from a third tenant.

(2) *Cross-tenant Reachability Isolation.* Packets from a host belonging to one tenant may not communicate with hosts belonging to another tenant, even though there is network connectivity between them (in terms of links forming a path from one host to another).

(3) *Cross-tenant Performance Isolation.* We will guard tenants from affecting each other's performance over the network by adding rate-limiting-based packet scheduling to the network.

Together, these features implement what is termed *network virtualization,* providing each tenant an illusion of its own dedicated isolated network with endpoint IP addresses of their choosing. In RU-CLOUDNET (and in some real cloud networks), we implement network virtualization by combining two qualitatively-different kinds of network switches: edge and fabric. In our network, edge switches are in charge of enforcing reachability. Fabric switches are in charge of forwarding between edge switches, and don't care about specific host-facing ports. In real clouds, edge switches often enforce rate limits, but in RU-CLOUDNET, we will use fabric switches to enforce aggregate rate limits over all traffic of each tenant.

How to build network virtualization in RU-CLOUDNET?

(1) One big switch: We require every packet going between hosts to go over the fabric switch. The fabric performs forwarding between edge switches using *encapsulated* packets. Encapsulation is the idea of attaching an extra set of header fields on a packet to indicate additional metadata, in this case information about the destination edge switch. You must program the P4 data plane tables, packet header formats, parser/deparser modifications, and control plane rules to make all of this happen.

When an edge switch receives a packet from a host-facing port (i.e., `s1-eth1, s1-eth2, s1-eth3` for edge switch `s1`), this (source) edge switch should assign a destination edge switch for that packet using the information available about the physical location of each tenant IP address (e.g., `h4` is on `s2`). Further, this information must be carried as a label which is a part of a custom header field added to the packet. The source edge switch will forward the packet (with the new custom header) to the fabric switch over the (unique) fabric-facing port of the edge switch (e.g., `s1-eth4` for `s1`). Once the packet reaches the fabric switch, it should use the destination edge switch label to forward the packet to the correct destination edge switch. When an edge switch receives a packet over a fabric-facing port, it should decapsulate the packet (i.e., remove the extra headers, so that the receiving endpoint can correctly understand the packet as the sending endpoint intended it) before forwarding the packet to the correct output port.

Your data plane and control plane logic must not only forward IPv4 packets between hosts, but also those of supporting protocols such as ARP. After implementing this step, we want a command like `h1 ping -c 1 h4` to work without packet loss. Also, **only the port corresponding to the destination host must receive the (IPv4 or ARP) packet;** do not simply flood all packets out of all ports.

(2) Address space reachability isolation. Implementing a one-big switch abstraction allows any host to talk to any other host simply using their (normal) TCP/IP stack, without worrying about any custom headers or forwarding logic. The trouble is, it also allows hosts *across tenants* to communicate with one another freely, a situation we would like to avoid when the tenants do not necessarily trust each other. We achieve reachability isolation by encapsulating packets with a tenant identifier, and, at the destination edge switch, forwarding packets using both the destination address and the tenant identifier on them.

When an edge switch receives a packet on a host-facing port, it must add (as part of a custom header) a label that indicates the tenant from which the packet arrived. It is possible to determine which tenant sent a packet simply by using the input port of the edge switch that the packet arrived on. Do not simply use source addresses, as malicious hosts can spoof them. The fabric switch leaves the label information alone. When the unmodified packet arrives at the destination edge switch, the packet is forwarded to the correct destination port only if the packet's tenant identifier matches the tenant identifier of the output port.

After implementing this part, hosts of the same tenant should be able to reach each other with no trouble (e.g., `h1 ping -c 1 h4` should succeed without packet loss), but hosts of different tenants should be isolated from each other (e.g., `h1 ping -c 1 h2` should fail with 100% packet loss). In this assignment, it is acceptable for any edge switch or the fabric switch to drop packets which attempt to violate reachability isolation.

(3) Performance Isolation. There are different ways to achieve performance isolation. In RU-CLOUDNET, we will use rate limiting at the tenant level (not host-pair or host level). We will achieve this through *metering*, specifically running a two-rate three-color marker (TR-TCM `https://datatracker.ietf.org/doc/html/rfc2698`) per tenant at the fabric switch. If a tenant *in aggregate* sends traffic at a rate higher than a specified maximum rate, its packets should be dropped.

You must implement the corresponding data plane modifications and add the necessary control plane rules to the fabric switch. The metering and dropping action for out-of-profile tenants is implemented purely at the fabric switch, since this switch sees all packets. You must implement a separate meter for each tenant, enabling the specification of different rate limits for different tenants. After implementing this, if a tenant exceeds its rate limit (in aggregate), you must start noticing packet drops.

We need to fix an average and peak rate for each TR-TCM. In this assignment, we use packet-based rates, and use an average rate of 1 packet/second with a burst of 1 packet, and a peak rate of 2 packets/second with a burst of 1 packet, in aggregate for each tenant.

3 Suggested steps to work through the assignment

Important References. The `P4_16` language specification is documented at

`https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html`

You may refer to the document above for thorough information on all the P4 language constructs, i.e., on what is permissible, and what is not.

The control plane interface (P4 runtime) is documented at

`https://github.com/p4lang/behavioral-model/blob/main/docs/runtime_CLI.md`

Many features may be specific to the target switch running P4 code. Our switch target (`bmv2`) is documented at

`https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md`

You will likely use scapy extensively to craft packets for debugging. Scapy is documented at

`https://scapy.readthedocs.io/en/latest/usage.html`

(1) Get familiar with the control plane by programming forwarding rules for IPv4 (not ARP) packets between two fixed hosts. Get IPv4 forwarding to work correctly between two fixed hosts in the network, without modifying the data plane program. This will allow you to get familiar with the control plane interface and give you experience in debugging packet flow over the network. Test your forwarding rules using `send_and_receive.py`, the event logs, and the packet captures on each switch interface along the path. Note that the edge and fabric need not use packet encapsulation here; the goal is to program a fixed forwarding rule for a single destination host on each switch along the path from source to destination host.

(2) Handle ARPs between two fixed hosts. Get ARP packets to be forwarded correctly. You should not flood ARP packets (unlike how ARPs are normally flooded in layer-2 networks), you need to parse ARP headers on incoming packets at each edge switch by adding to your data plane the header definition of an ARP header. Consult Internet references on the ARP protocol to understand its format and encode it in a P4 header. You must also add the corresponding parse transitions in the parse graph for Ethernet (i.e., for `ethertype 0x0806`). Once you've understood where an ARP packet is destined to (using the *target protocol address* field), there are two ways to handle it. Either you could create a new ARP-specific table which contains a copy of the rules you used for IPv4 forwarding. Alternatively, you could add a separate table that copies either the IPv4 destination address or the ARP target protocol address into a *P4 metadata field* that can then be used as the key for your (unified IPv4+ARP) forwarding table. We recommend the second approach as it is more elegant and eventually simpler.

If you parse ARP, your deparser must also emit ARP. Deparsers only emit fields that are *valid*, i.e., previously parsed correctly or explicitly set to be valid in a table action. If your packet contains only either a valid IPv4 header or a valid ARP header but not both, `emit(ipv4); emit(arp);` will do the right thing and only produce one header in the deparsed packet.

(3) Label packets with destination edge switch for switch-to-switch forwarding. Define a new custom header that includes a destination edge switch identifier. You must also create a new table with an action that writes a value into this new custom field.

A crucial question is, where will this custom header be located relative to the existing headers (Ethernet, IPv4/ARP) on the packet? We recommend placing your new header right after Ethernet, because you must carry destination edge switch information for both IPv4 and ARP packets, which "share" only an Ethernet header in the parse. **We recommend using a new, custom ethertype 0xFFFF** to denote that your custom header follows Ethernet on the packet (rather than IPv4 (0x0800) or ARP (0x0806)), and add a corresponding new parse transition and parser state to your parser.

It is important to preserve the previous ethertype somewhere on the packet, since you will need to parse IPv4 and ARP on the destination edge switch to forward to the correct destination host. Add an ethertype field to your new header, and copy the previous ethertype into this custom field before you erase the packet's original ethertype by writing into it. Note that any new header must first be set valid before writing into it can take effect. You can use the `setValid()` call on the new header before writing into its fields.

Since your packet contains a new header, also ensure that your deparser emits the new header in the correct order relative to other headers.

The fabric switch must use the destination edge switch to determine where the forward the packet. At the destination edge switch, decapsulate the custom header we added by (i) copying its `ethertype` back into the Ethernet header's `ethertype`; and (ii) setting the custom header invalid.

(4) Add reachability isolation by labeling packets with tenant information. Use the packet's input port (available as part of `standard_metadata`) at the source edge switch as a match key to determine which tenant an incoming packet belongs to. **Do not use a packet's source address to identify a tenant** as those addresses can be spoofed by malicious hosts. Encode the packet's corresponding tenant into a new field in your custom header. At the destination edge switch, use both the tenant identifier and the destination address to determine the correct output port.

(5) Meter based on the tenant identifier. At the fabric switch, add a metering table that marks packets into one of three "colors" based on their rates (measured in aggregate over all hosts of each tenant, using the tenant identifier), and another table that drops packets that are marked the RED color (out of profile). Meters are external objects which are not part of the official P4 language specification, but rather the specific target running the P4 program. For `bmv2`, meters are documented here:

`https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4#L201`

(see large blocks of comments starting at that point). Further, use the `direct_meter`, documented under `bmv2`:

`https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md#bmv2-direct_meter-implementation-notes`

See the rate specifications for each tenant in §2.

(6) Collect all your control plane commands into switch-specific files. All your commands to `s1` (i.e., the ones you sent to `s1` using `simple_switch_CLI`) should be collected in a file `s1-commands.txt`, commands to `s2` in `s2-commands.txt`, and so on. These files should be formatted in such a way that they program the intended rules when running the following terminal commands:

```
simple_switch_CLI --thrift-port 9090  < s1-commands.txt
simple_switch_CLI --thrift-port 9091  < s2-commands.txt
simple_switch_CLI --thrift-port 9092  < s3-commands.txt
simple_switch_CLI --thrift-port 10101 < s4-commands.txt
```

4 What you must submit

You must submit seven files: the updated `edge.p4` and `fabric.p4`, as well as the four switch command files `s1-commands.txt, ..., s4-commands.txt`, and a report (`report.pdf`) in a single zip file. The report must be in PDF format only; we do not accept word (docx) or text formats. The report must answer the following questions:

1. The top of the file should provide the names and netIDs of one or both team members. **Only one team member needs to submit**
2. Collaborators and References: If you consulted resources other than the references listed in this document, or discussed the project with people outside of your teammate and the course staff, please note them here and describe the nature of your consulation.
3. What functionality works currently, and what doesn't?
4. If you have any other thoughts to share about this assignment, please note them here.

We will test your code for the specification of the functionality in the assignment (e.g., reachability tests using `ping`, efficacy of rate limiting), as well as ensure that you have avoided incorrect or inefficient shortcuts (e.g., avoid forwarding packets out of unnecessary ports). We will run packet captures to check the existence and contents of custom header fields.

Your code must build using the commands provided above, and run on the course VM. If we cannot build your code, or if our VM crashes when running your software, you will lose a significant fraction of the points.

Please start early, and ask many questions on Piazza to help you move forward and get this project submitted on time.

References

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 2014.

[2] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 2013.