Redes Definidas por Software
Control Plane - P4Runtime

Version: 1

Departamento de Informática
Universidade do Minho
May 2024

João Fernandes Pereira

1 Goal

In the realm of network programming, the Control Plane serves as the brain orchestrating the behavior and functionality of the underlying Data Plane. Pioneering technologies like P4Runtime [1] revolutionize how we interact with and manage network devices, offering unprecedented flexibility and programmability. This assignment delves into the intricate landscape of Control Plane programming for P4Runtime, where students will explore the fundamental concepts, principles, and practices essential for crafting dynamic, efficient, and scalable network control applications.

The purpose of this document is two-fold. Firstly, it outlines the steps involved in upgrading the work environment to support P4 [2], ensuring compatibility and functionality for P4Runtime development. Secondly, it provides a detailed example illustrating the implementation of a network fabric using Mininet, accompanied by the development of both Data Plane and Control Plane code. The document concludes with an assignment to reinforce understanding and application of the concepts covered.

This Work Environment was tested in Ubuntu 22.04 and 23.04, but it should work for the following versions: 20.04, 21.04 and 21.10.
Check your OS version:

```
$ lsb_release -a
```

2 Work Environment Upgrade

Having already learned Data Plane programming using P4, it's time to enhance our work environment to support P4Runtime. This section is about installing the necessary software and tools so you can delve into Control Plane programming effectively. It's important to view this section as a complement to the first tutorial of the semester; assuming you've completed that tutorial, your system should be in accordance with its requirements.

The following list identifies the software needed for your work environment upgrade.

- gRPC
- protobuf
- p4runtime
- p4lang-PI

2.1 Add P4Lang Repo

It's worth noting that the method for adding the repository has changed since the Data-Plane-Tutorial at the beginning of the semester. The use of `apt-key` is now fully deprecated, and instead, we should utilize `gpg --dearmor` for this purpose.

```
$ . /etc/os-release
$ curl -sS \
  "http://download.opensuse.org/repositories/home:/p4lang/xUbuntu_${VERSION_ID}/Release.key" \
  | gpg --dearmor | sudo tee /usr/share/keyrings/p4lang.gpg
$ echo "deb [signed-by=/usr/share/keyrings/p4lang.gpg] \
  http://download.opensuse.org/repositories/home:/p4lang/xUbuntu_${VERSION_ID}/ /" \
  | sudo tee /etc/apt/sources.list.d/p4lang.list
$ sudo apt update
```

2.2 Install Software

Now, with the repo added (check the output of `sudo apt update`), we shall install the required software.

```
$ pip3 install grpcio==1.51.3
$ pip3 install protobuf==4.21.6
$ pip3 install p4runtime==1.3.0
$ sudo apt install p4lang-pi
```

3 Workspace

The following repository serves as the main resource in your P4Runtime journey. This repository provides you with a hands-on example that is explained throughout this document. The objective is to empower you with the knowledge and skills needed to develop your own P4Runtime controller.

```
$ cd ~
$ git clone https://github.com/jfpereira-uminho/RDS-ControlPlane.git
$ rm -rf RDS-ControlPlane/.git # remove this git repo, you may want to create your own
```

The structure of this repo can be simplified as follow:

- `build` - p4 compiled files (.json and .p4info)
- `commands` - legacy, we don't need this anymore
- `controller` - code for the controller
- `logs` - controller log files, one for each p4 device
- `mininet` - code for network fabric
- `p4` - code for your p4 devices
- `tools` - passive debug tools (nanomsg_client.py) and lib install script (cp-python-libs.sh)
- `utils/p4runtime_lib` - wrapper lib to help the controller with bmv2-p4 devices

3.1 Testing

In this section we want to test our Work Environment, to do that we just run the scenario present in the Workspace.

Run lib installation script - This script is intended to be run successfully only once

```
$ cd ~/RDS-ControlPlane
$ sh tools/cp-python-libs.sh
```

Start mininet

```
$ cd ~/RDS-ControlPlane
$ sudo python3 mininet/r-topo.py
```

Start Controller

```
$ cd ~/RDS-ControlPlane
$ python3 controller/dummy-controller.py
```

Test

```
mininet> pingall
```

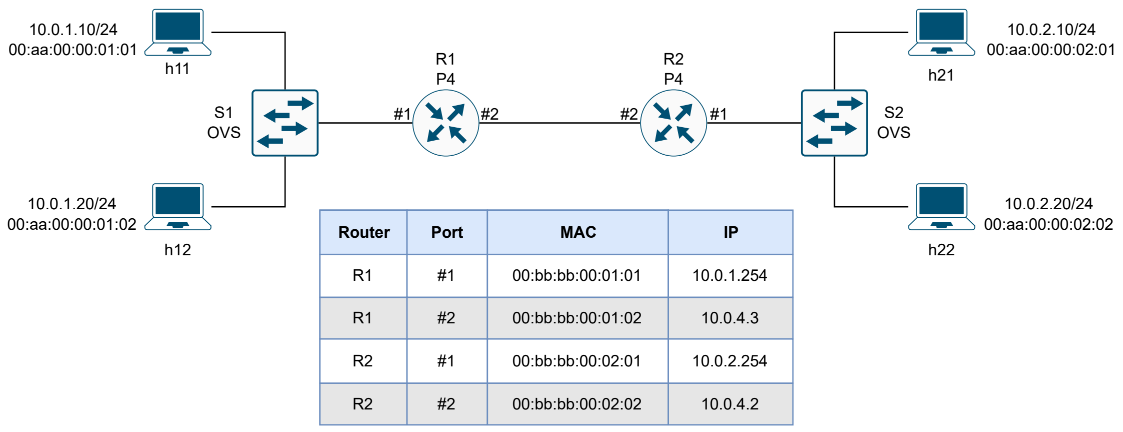
If you encounter any issues, please describe the problem and notify the instructor via email.

4 Example

In this section, we will delve into the development of the example provided in the repository. The purpose of this example is to implement two simple routers, involving the addition of rules to the Open vSwitch (OVS) and P4 devices to enable traffic flow between two networks.

4.1 Network Topology

As you must already know by now, a good first step is to design the topology. Fig. 1 shows the topology of the example.



Router	Port	MAC	IP
R1	#1	00:bb:bb:00:01:01	10.0.1.254
R1	#2	00:bb:bb:00:01:02	10.0.4.3
R2	#1	00:bb:bb:00:02:01	10.0.2.254
R2	#2	00:bb:bb:00:02:02	10.0.4.2

Figure 1: Example topology.

4.2 Network Fabric - Mininet

The mininet script is defined in the *r-topo.py* file. In this section we will highlight the differences that allow the implementation of P4Runtime.

Adding Switchs

We have a new file under the mininet dir, *p4runtime_switch.py*. In line 10 of the *r-topo.py* we import the class P4RunTimeSwitch, this class allows us to instantiate P4 switch with P4Runtime capabilities. The code sample below shows how to use this class.

```
r1 = self.addSwitch('r1',
                  cls = P4RuntimeSwitch,
                  sw_path = sw_path,
                  #json_path = json_path,
                  thrift_port = thrift_port,
                  grpc_port = grpc_port,
                  device_id = 1,
                  cpu_port = 510)
```

Has you can see, we no longer give the json (output of the p4 compiler) to the switch, we also have a new server on top of our switch, a gRPC (g Remote Procedure Call) [3]. Finally we manually define the device_id. The device_id will be useful to identify the switch in the controller, the gRPC will be responsible for the connection with the controller. You can still use the thrift server to connected with *nanomsg* in order to see what is happening in the P4 switch in real time. The cpu_port is the port used to send packets to the controller [4].

The following listing shows the arguments that our script receives, and their default values.

```
parser = argparse.ArgumentParser(description='Mininet demo')
parser.add_argument('--behavioral-exe', help='Path to behavioral executable',
               type=str, action="store", default='simple_switch_grpc')
parser.add_argument('--thrift-port', help='Thrift server port for table updates',
               type=int, action="store", default=9091)
parser.add_argument('--grpc-port', help='gRPC server port for controller comm',
                  type=int, action="store", default=50051)
```

We have some changes here, the default value of --behavioral-exe is now simple_switch_grpc instead of the previous simple_switch. We don't have any argument for the json file, and there is a new argument --grpc-port with the default value 50051, it defines the first port of the gRPC servers, one per P4 switch.

4.3 Data Plane

The code present in *p4/s-router.p4* is identical to the resulting code from the data-plane tutorial conducted at the beginning of the semester. The only change is that a counter [5] was added and is incremented upon the ipv4_fwd action. Basically, this new version of the simple-router counts every packet that forwards.

```
counter(8192, CounterType.packets) c;
```

```
action ipv4_fwd(ip4Addr_t nxt_hop, egressSpec_t port) {
      meta.next_hop_ipv4 = nxt_hop;
      standard_metadata.egress_spec = port;
      hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
      c.count((bit<32>) 1);
   }
```

4.4 Compiling

The way we use the p4 compiler (p4c) is slightly different. We need to instruct the compiler to give one more output, besides the *json* file. This file is known as a *P4info* and its primary function is to provide a structured representation of the capabilities and features supported by a P4 program. In more detail:

1. **Specification of Capabilities:** The *P4info* file specifies the capabilities of the P4 program, including the tables, actions, and their parameters. It outlines the functionality that can be controlled and configured dynamically through the P4Runtime API.

2. **Runtime Configuration:** It enables runtime configuration of the P4 program by defining the structure and attributes of tables and actions that can be manipulated during runtime. This allows for dynamic reconfiguration of network behavior without requiring recompilation of the P4 program.

3. **Interoperability:** The *P4info* file facilitates interoperability between the P4 program and the control plane applications using the P4Runtime API. It provides a standardized format for communicating the capabilities and requirements of the P4 program to the controller.

4. **Automation and Tooling:** The structured information contained in the *P4info* file can be leveraged by automation tools and development environments to simplify the development and deployment process. It enables automatic generation of client libraries, configuration interfaces, and documentation.

```
$ cd ~/RDS-ControlPlane
$ p4c-bm2-ss --p4v 16 --p4runtime-files build/s-router.p4.p4info.txt -o build/s-router.
    json p4/s-router.p4
```

4.5 Control Plane

In our example, the controller's code is in *controller/dummy-controller.py*, lets start by exploring the starter block of the script.

Here we have the argument parser of our script. As you can see we have two arguments the *p4info* file, and the *json* file, basically the output files of the compiler. If you look carefully, you can see that the default values for `--p4info` is *build/s-router.p4.p4info.txt* and for `--bmv2-json` is *build/s-router.json*. This means that for this default values to work the script must be run from the workspace parent dir, otherwise the default values will not work. The script goes on to do some controls and then calls the main function.

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='P4Runtime Controller')
    parser.add_argument('--p4info', help='p4info proto in text format from p4c',
                        type=str, action="store", required=False,
                        default='build/s-router.p4.p4info.txt')
    parser.add_argument('--bmv2-json', help='BMv2 JSON file from p4c',
                        type=str, action="store", required=False,
                        default='build/s-router.json')
    args = parser.parse_args()
```

Looking at the main function, it can be summarized like:

1. Instantiate an object that contains the representation of P4 device specification, based on the p4info file.

2. Define P4 devices and connections to them.

3. Define the controller as master for the P4 devices.

4. Installing the P4 program in the devices.
5. Writing rules.
6. Reading rules - not necessary but is here for you to see this capability.
7. Reading the counter of the P4 devices each 10 seconds.

4.5.1 Connection to P4 devices

The following code shows how to configure the connections to the P4 devices.

```
def main(p4info_file_path, bmv2_file_path):
   # Instantiate a P4Runtime helper from the p4info file
   p4info_helper = p4runtime_lib.helper.P4InfoHelper(p4info_file_path)

   try:
      # this is backed by a P4Runtime gRPC connection.
      # Also, dump all P4Runtime messages sent to switch to given txt files.
      r1 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
         name='r1',
         address='127.0.0.1:50051',
         device_id=1,
         proto_dump_file='logs/r1-p4runtime-request.txt')
      r2 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
         name='r2',
         address='127.0.0.1:50052',
         device_id=2,
         proto_dump_file='logs/r2-p4runtime-request.txt')
      print("connection successful")
```

The object `r1` represents the connection to the P4 device that is running in the localhost (127.0.0.1) and has the gRPC server in port 50051. It has *r1* has a given name (it needs to be the same name given in mininet), and the *device id* is 1. The file for dumping all messages sent to `r1` can be anything, but needs to exist. The `Bmv2SwitchConnection` does not create the log file, if the file does not exist it will return an error. You need to define connections to every P4 define present in your network fabric.

The following code is also mandatory in every P4Runtime controller, and it send an *arbitration update message*, that defines this controller as he main controller of the device. In this case, your controller will be the *master* of device *r1* and *r2*.

```
r1.MasterArbitrationUpdate()
r2.MasterArbitrationUpdate()
```

4.5.2 Installing P4 program in P4 devices

At this point our P4 devices are still empty, meaning that they don't have a P4 program running on them. It's the controller responsibility to install these P4 programs. This is the reason that we don't pass the *json* file to mininet, but instead the *json* and *p4info* file are passed as arguments to this controller. As you can see in the following code, the method `SetForwardingPipelineConfig` is responsible for this step.

```
r1.SetForwardingPipelineConfig(p4info=p4info_helper.p4info,
                               bmv2_json_file_path=bmv2_file_path)
r2.SetForwardingPipelineConfig(p4info=p4info_helper.p4info,
                               bmv2_json_file_path=bmv2_file_path)
```

4.5.3 Writing rules in P4 devices

The main function goes on to call functions to write and read table entries (rules), let us focus on those functions now.

The strategy to write a table entry is quite simple:

- use the `p4info_helper` object to write a table entry, this object ”knows” your P4 device structure.
- a table entry can be described by:
  - `table_name`
  - `match_fields`
  - `action_name`
  - `action_params`
- use the method `WriteTableEntry` of the object representing the connection to the desired P4 device to write the table entry.

Remember, you are not writing tables, that was done when you installed the P4 program, you are writing table entries. In other words you are populating the tables. This can be a little more verbose than the rules defined in the *commands* dir (that we don’t need anymore), but they mean the same thing. The main advantage is that we can write and delete table entries more dynamically, and in response to network events (if the controller is sophisticated enough).

Therefore, the following function writes entries in the *src_mac* table. It receives as arguments the `p4info_helper` object, the object that represents the connection to a P4 device `sw`, and a dictionary containing pairs `port:mac`. The purpose of this table is to map the interfaces (ports) of the P4 devices to mac addresses, basically we are giving mac addresses to the interfaces of our device.

```
def writeSrcMac(p4info_helper, sw, port_mac_mapping):
   for port, mac in port_mac_mapping.items():
      table_entry = p4info_helper.buildTableEntry(
         table_name="MyIngress.src_mac",
         match_fields={
            "standard_metadata.egress_spec": port
         },
         action_name="MyIngress.rewrite_src_mac",
         action_params={
            "src_mac": mac
         })
      sw.WriteTableEntry(table_entry)
   print("Installed MAC SRC rules on %s" % sw.name)
```

You can read the above code like: for the table name `src_mac` defined in the control block `MyIngress`, `MyIngress.src_mac`, when the outgoing port, `standard_metadata.egress_spec`, is equal to `port` execute the action named `MyIngress.rewrite_src_mac`. Upon execution, the action argument named `src_mac` should be equal to `mac`.The values for `mac` and `port` are defined in each entry of the dictionary.

The last function to be discussed in this document is the `writeFwdRules`. This function writes, creates entries, in two different tables: `MyIngress.ipv4_lpm` and `MyIngress.dst_mac`. This happens because the two tables are deeply connected. The *ipv4_lpm* table and action, define the selection of the *next hop* and the *outgoing port*. The selection of the *next hop* will define the *mac address* that is going to be set as *destination mac*, it need to be the mac addrees of the next hop.

```
def writeFwdRules(p4info_helper, sw, dstAddr, mask, nextHop, port, dstMac):
    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.ipv4_lpm",
        match_fields={
            "hdr.ipv4.dstAddr": (dstAddr, mask)
        },
        action_name="MyIngress.ipv4_fwd",
        action_params={
            "nxt_hop": nextHop,
            "port": port
        })
    sw.WriteTableEntry(table_entry)

    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.dst_mac",
        match_fields={
            "meta.next_hop_ipv4": nextHop
        },
        action_name="MyIngress.rewrite_dst_mac",
        action_params={
            "dst_mac": dstMac
        })
    sw.WriteTableEntry(table_entry)
```

As the syntax of a table entry was alredy desribed in this document we will focus in the arguments of the function:

- `p4info_helper` - the object that allows the construction of table entries.
- `sw` - the object that represent the connection to the P4 device.
- `dstAddr` - the destination IP address
- `mask` - netmask of the destination IP address
- `nextHop` - IP address of the next hop
- `port` - id number of outgoing port
- `dstMac` - mac address of the next hop

The way that the code can be read, is exactly the same as discussed before.

5 Assignment - P4Runtime Controller Development

Objective:

In this assignment, you will build upon your knowledge and skills acquired in the previous assignments to develop a P4Runtime controller. The primary objective is to transition from manual rule injection via simple_switch_CLI to dynamic, programmable control of P4 devices using P4Runtime. You will be tasked with creating a controller capable of managing the most complex network topology developed throughout the semester.

Tasks:

1. **Topology Configuration:**
    - Utilize Mininet to instantiate the most complex network topology developed, by our group in this semester, which includes multiple LANs, OVSSwitches, and P4 simple-routers.

2. **P4Runtime Controller Development:**
    - Develop a P4Runtime controller capable of connecting to P4 devices within the network topology.
    - Implement functionality to dynamically inject forwarding, firewall and icmp rules into the P4 devices using P4Runtime protocol.
    - (Extra) Ensure that the controller can handle rule updates, additions, and deletions in response to network events or policy changes. [4]
3. **Testing and Verification:**
    - Test the functionality of the P4Runtime controller by sending traffic through the network and observing the behavior of P4 devices.
    - Verify that forwarding and firewall rules are correctly installed and enforced by the controller.
    - Verify that the simple-routers can respond to icmp messages directed to their interfaces.
    - (Extra) Ensure that the controller can respond to network events and dynamically adapt to changes in network conditions. [4]
4. **Report:**
    - Document the design and implementation of the P4Runtime controller, including any challenges encountered and solutions implemented.
    - Provide a detailed report outlining the testing procedures, results, and observations.
    - Include any recommendations for future improvements or enhancements to the controller.

Deliverables:

- All the developed code: network fabric (mininet), data plane, control plane.
- Report detailing the design, implementation, testing, and results.

Submission Guidelines:

- Submit all the developed code along with the report via blackboard group-file-exchange.
- The submission must be a .zip archive, **RDS-G"X"-TP3.zip**. Replace **"X"** with your group number. Your group number contains 2 digits.

Deadline:

13th June 2024

References

[1] *P4Runtime Spec.* [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html (visited on 05/08/2024).

[2] *P4 Spec.* [Online]. Available: https://staging.p4.org/p4-spec/docs/P4-16-v1.2.4.html (visited on 05/08/2024).

[3] *Grpc*. [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/ (visited on 05/08/2024).

[4] *P4 Packet in : Packet out*. [Online]. Available: https://github.com/jafingerhut/p4-guide/blob/master/ptf-tests/packetinout/packetinout.p4 (visited on 05/08/2024).

[5] *P4-cheat-sheet.pdf*. [Online]. Available: https://github.com/p4lang/tutorials/blob/master/p4-cheat-sheet.pdf/ (visited on 05/08/2024).