Technical Report[1]

2017.10.01 – 2017.12.31

Author: Márton Rosanics, BSc student

Supervisor: Sándor Laki, head of subproject Advanced Networks

Disclaimer (2018/08/30):

I edited this technical report retrospectively, in hopes that it would provide sufficient information about my project – *in English*. I added, removed, and changed several parts of the original report.

That said, there is a Hungarian version of this documentation which is much more detailed; here I focused only on the developer's side of things, showcasing how this project has come to life. The subject of this technical report was later on submitted as my thesis for my Bachelor's degree, under the title of *NS-3 szimulátor kiegészítése P4 programozható switch támogatással*, which roughly translates into:

P4-programmable switch support for NS-3 simulator

Contents

1This work was supported by EFOP-3.6.3- VEKOP-16- 2017-00002 project.

Motivation, Problem Description

Motivation, goals

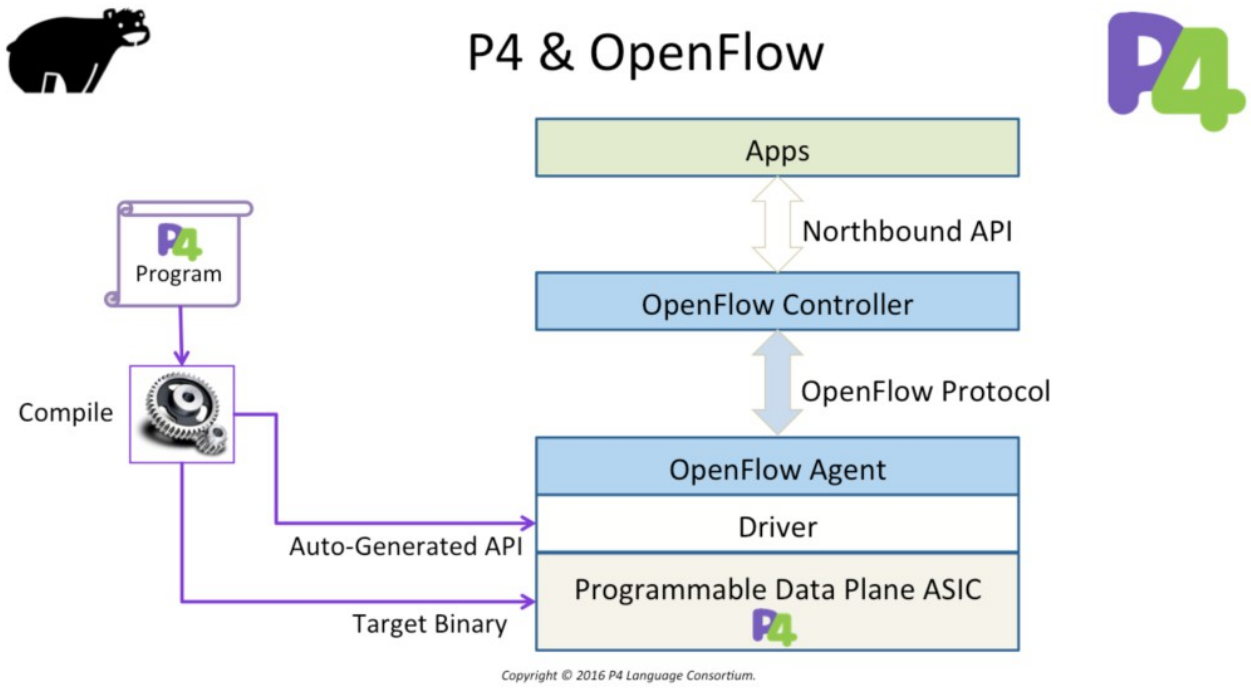
NS-3 is one of the most wide-spread network simulators. It's an open-source project and has a modular structure, which makes it easy to expand with new network protocols and solutions. Then, there is P4: a domain specific programming language, designed to allow programming of packet forwarding planes.

> The motivation behind my project is to connect these two subareas of network programming, which will hopefully prove to be useful for both NS-3 and P4 communities.

A P4-programmable switch has several functionalities that could come useful in an NS-3 simulation:

- Protocol independency: P4 programs specify how a switch processes packets.
- Target independency: P4 is suitable for describing everything from high-performance forwarding ASICs to software switches.
- Field reconfigurability: P4 allows network engineers to change the way their switches process packets after they are deployed.

This third point is what would make a P4 switch support stand out from the current alternative for NS-3, the OpenFlow switch support.



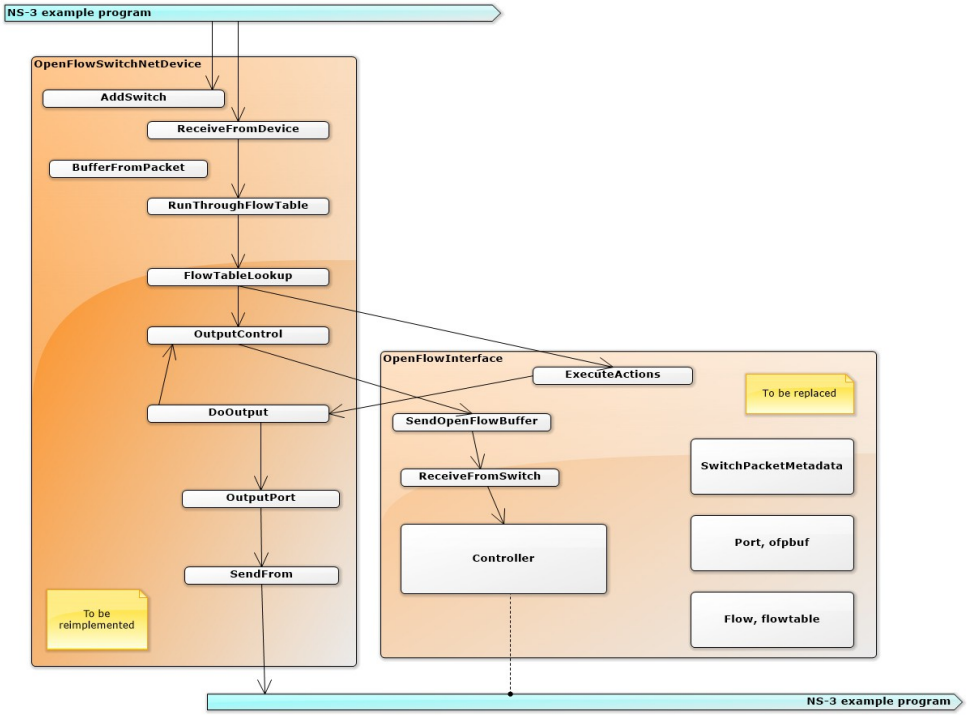
Problem description

In the following section, I start out by describing the problem in the broadest sense, gradually getting into the smaller, more technical parts that I had to overcome.

First off, I needed to figure out how to connect the switch (written in P4) to the NS-3 simulation (written in C++). To make this interface between NS-3 and the P4 switch possible, I used P4@ELTE team's retargetable compiler for P4 (also known as *t4p4s*). This compiler takes the P4 code and generates C code from it, which in turn the NS-3 simulation should be able to interpret and use. The first hurdle was to make this connection possible. The approach I chose to do was to take a single P4 switch, make it work within the NS-3 simulation first, then expand other functionalities from there.

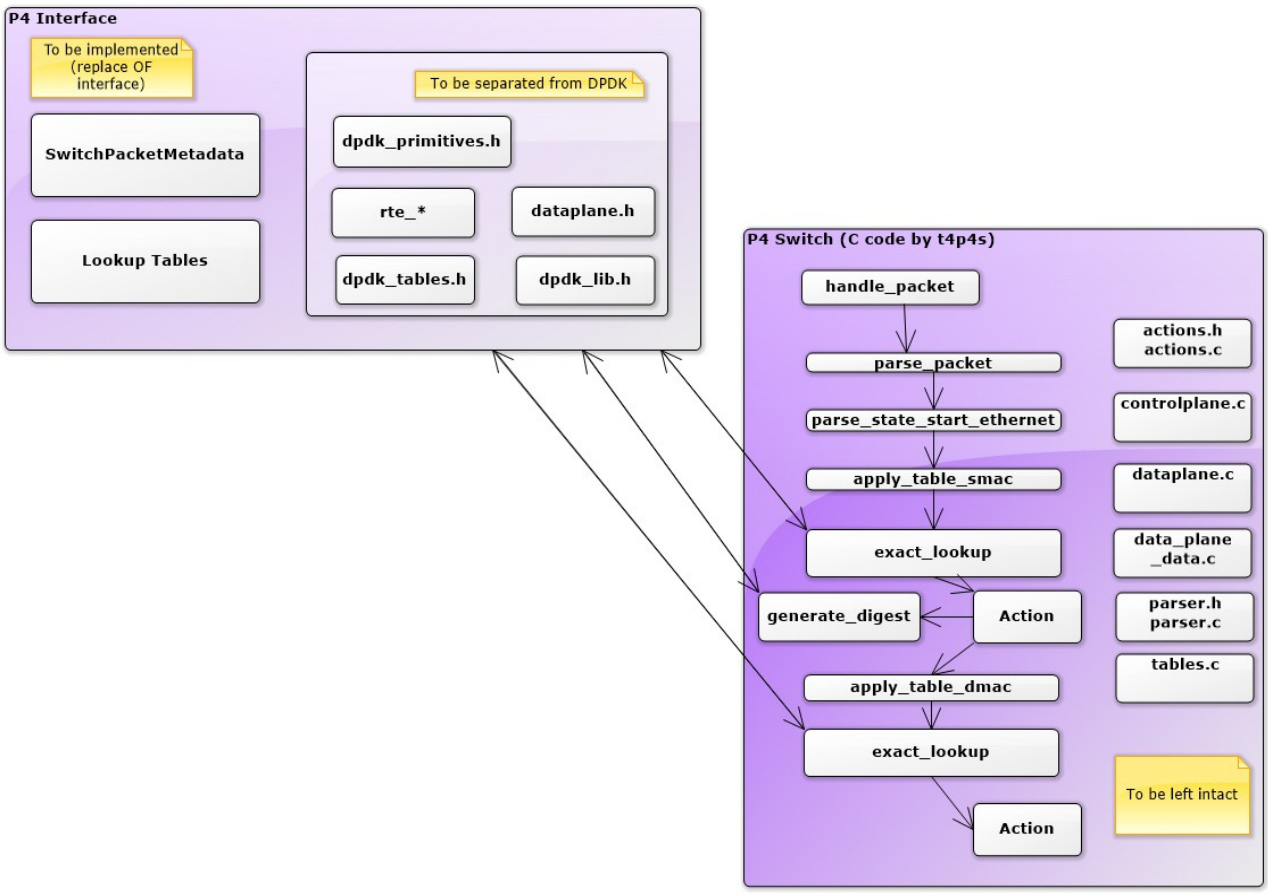
I took the L2 example from the ELTE P4 compiler (8 files in C language) and used it as my sample switch. At the same time, in NS-3 I created a new module (P4SwitchNetDevice), which would function as the switch itself within the simulation. For this module, I took the OpenFlow switch module as a template; it's structure is similar to what I needed, so it was a good starting point for my switch.

OpenFlow switch has a sample NS-3 simulation as well, which helped me tremendously. I examined it and mapped out the call order of the functions within the simulation and the switch, in order to understand its workings better. This way it was easier to plan out which parts needed to be reimplemented, and which parts weren't necessary for a basic P4 switch implementation. Below is a rough representation of the call order within the OpenFlow switch.



The call order, or more specifically, the path that the packet takes can be quite convoluted with OpenFlow; I had to figure out how to simplify it for my P4SwitchNetDevice. Fortunately, I didn't need every component from the OpenFlow switch module; my module – at least in its earlier versions – was going to be much simpler. I also didn't need to implement controllers and flow tables directly into the module, as they would be defined by the P4 switch, in my case.

The following image shows what I considered the most important parts of the sample P4 switch.



The “P4 Switch” box in the picture represents all those files and functionalities that had been generated by the ELTE P4 compiler. (I listed the eight C files on its right side.) The goal here was to leave these files intact, but implement every function call that they have (or would have).

The P4 interface, on the other hand, is something that was already implemented in the ELTE P4 compiler. It isn't part of the generated C files, and the code itself is partially dependent on the backend it is used with. Here, I had hardware-dependent parts that were depending on DPDK (a set of libraries and drivers for fast packet processing). I needed to find alternate solutions (i.e. write my own implementation) for those parts of the switch to work.

There were also several type-declarations, functions, variables and macros used by the sample L2 switch that weren't within the mentioned eight generated files. I had to gather up these dependencies from various source files within t4p4s (or had to reimplement them) to be able to build the project. To understand this better, I studied another sample project within t4p4s, one with freescale backend (instead of DPDK). It's implementation contained several methodologies that I could use.

Further problems:

To build the NS-3 project, WAF (a Python-based build tool) is used. Each NS-3 module has a file (called wscript) that informs WAF about the components of the module. To be able to build and run the project, it is necessary to list every file and every other module-dependecies here. Build configurations can be also specified in this file.

There were a couple of caveats that I needed to deal with:

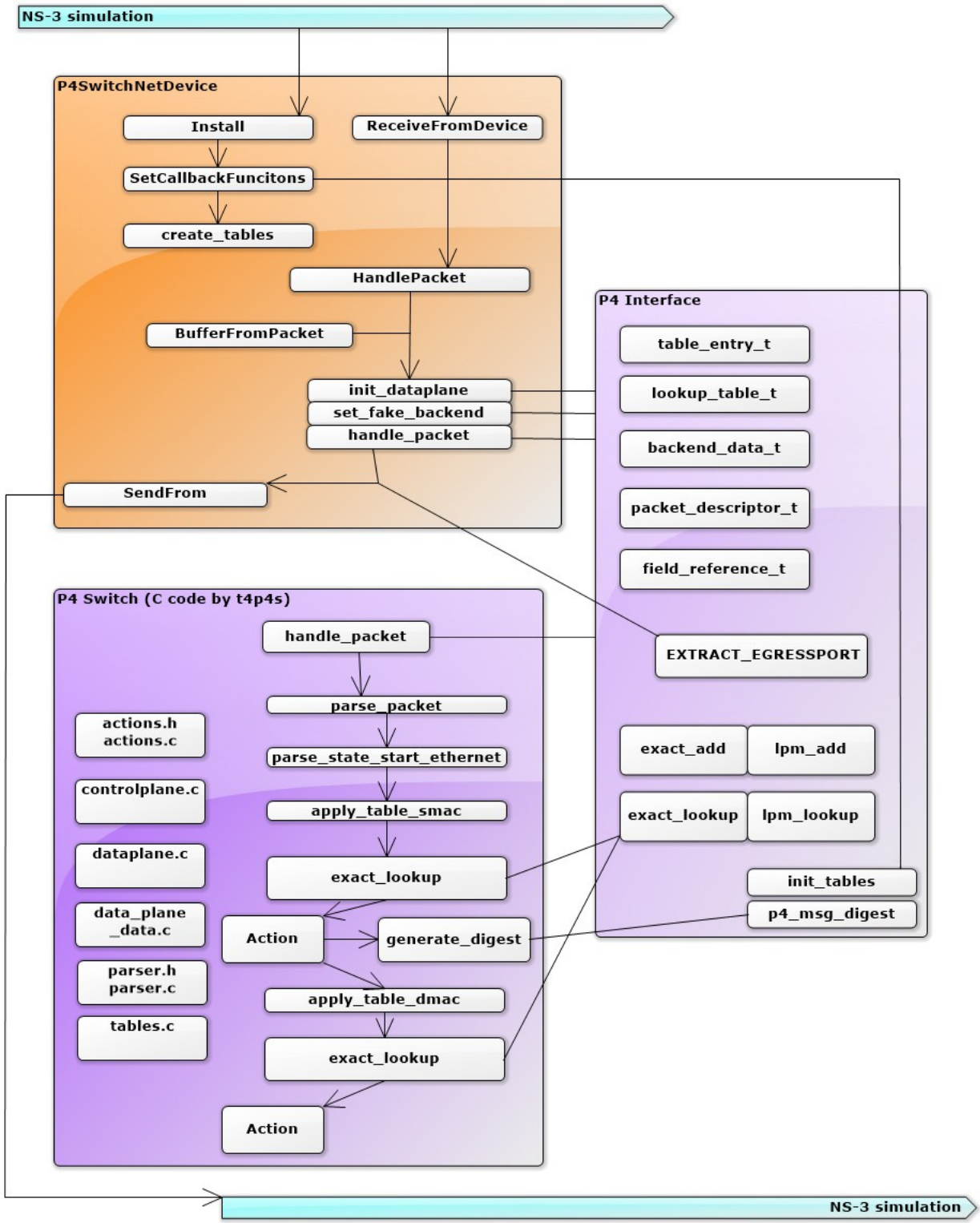
- The simulation uses both C and C++ source files. While building the project, this caused linking errors prior appropriate configuration.
- All of the “includes” have to be listed in the wsript file. I found it difficult to use large libraries (containing multiple files) in the module, because all of the file names need to be listed. This is why I refrained including larger libraries (like Judy array for the longest prefix match tables, detailed later on) and opted to do my own implementations instead.

Also worth to mention:

Everything described so far is just one way to approach for this problem. There are other possible solutions too, like integrating the already existing Bmv4 (P4 interpreter) into NS-3. In the end however, that would have required even more work than the current approach. Within the (time and workload) boundaries of this project, this current solution seemed more feasible.

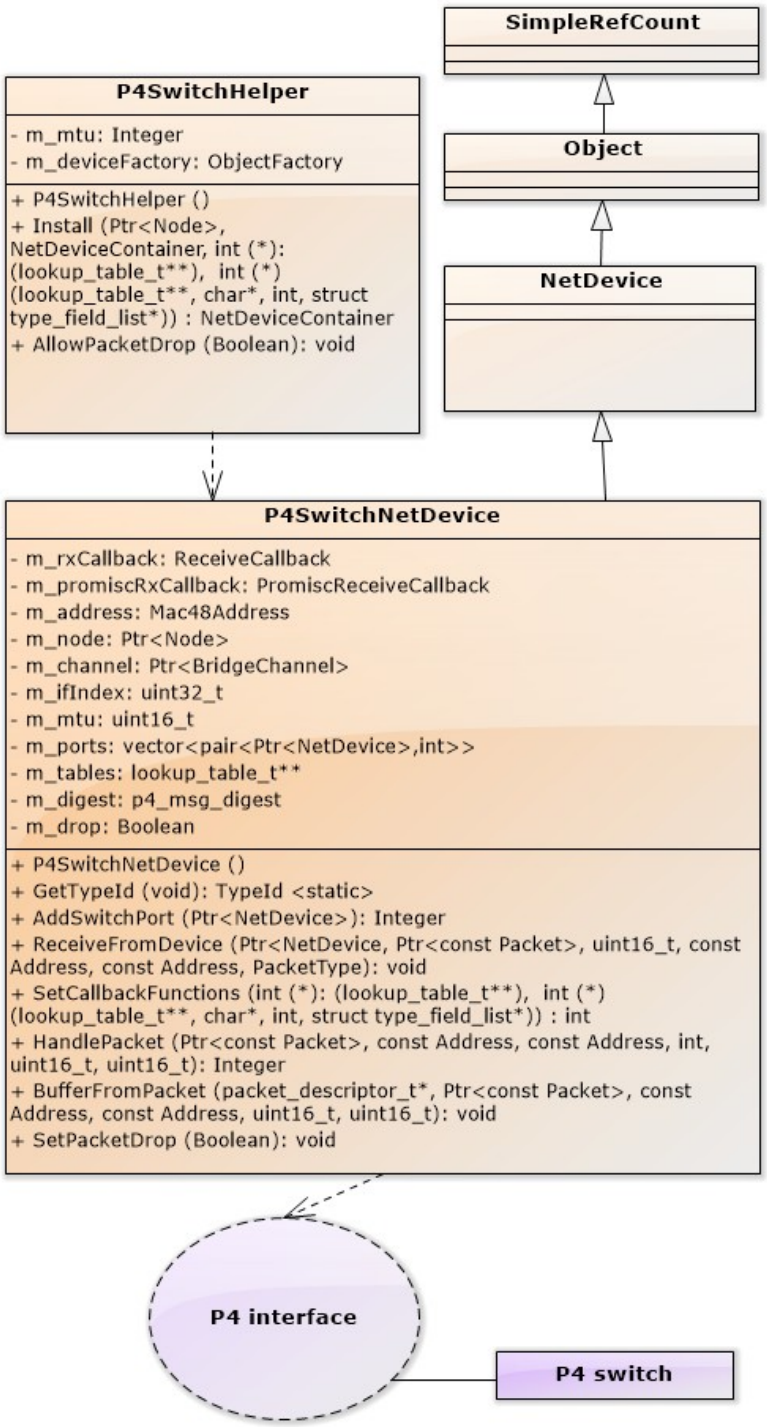
Implementation, Results

Just like for the planning, I made a rough representation about the call order within the simulation. By no means is this an accurate illustration of the P4SwitchNetDevice, P4 interface and P4 switch – my only purpose here was to highlight the most important parts.



The previous image depicts the sample L2 switch from t4p4s, but I ran the simulation on the sample L3 switch as well (meaning that provided the user writes the appropriate P4 switch, Layer 3 routing is also possible within the NS-3 simulation with the use of this module).

The structure of the module



For the most part, both the P4SwitchHelper and P4SwitchNetDevice classes were quite straightforward, their essential methods implemented on the OpenFlow switch's example. There were a few things I needed to be careful about though, like storing the lookup tables, assigning port-numbers, and adding features to take care of “dropped” packets. The most complicated part in this section was the packet handling, however.

Packet handling

After the P4SwitchNetDevice receives a packet, it will soon be passed on to its HandlePacket method. This is where the actual work is going to happen. The P4 switch's data buffer and the data plane will be initialized, most of it already defined within the compiled P4 switch. For my part, I didn't need to do much with the data plane as it was already implemented by t4p4s, though the *ingress port* field of the packet descriptor was needed to be set manually.

But the data buffer needed some additional work. By the time the NS-3-defined Packet arrive to this point, the simulator already got rid of the ethernet header – the P4 switch expects it, so I needed to add it manually. Not only that, but it turned out that NS-3 uses its own packet-format, so I basically had to convert the whole thing, for the P4 switch to use it. Once the conversion is done, the data is

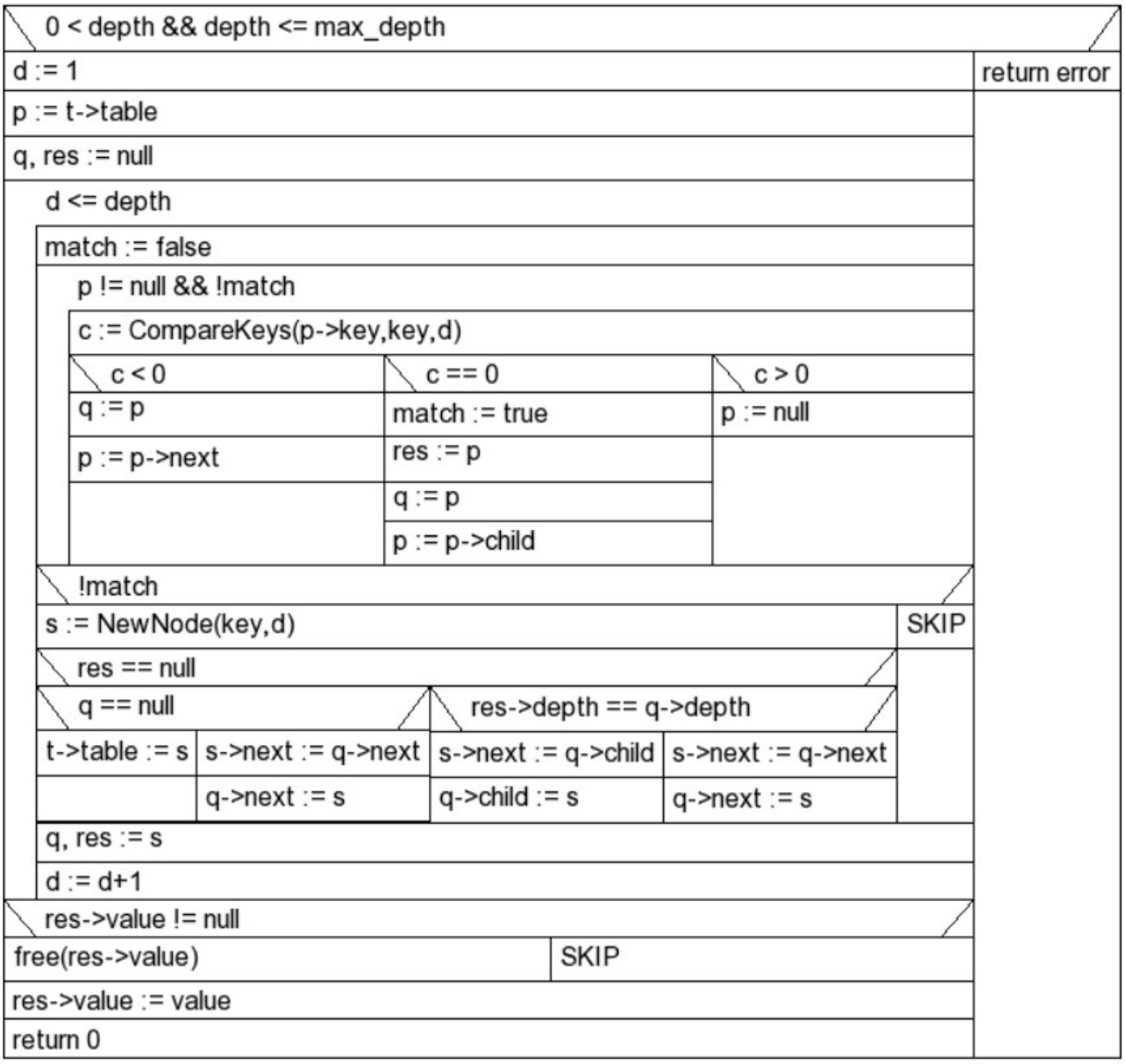
Lookup tables

Currently there are three supported types of lookup tables: *EXACT* (where the key needs to match perfectly during lookup), *LPM* (longest prefix match) and *TERNARY* (where the lookup-key can be masked freely). All of these tables have simple structures, since the actual runtime of the simulation isn't important; it's the simulated runtime that really matters.

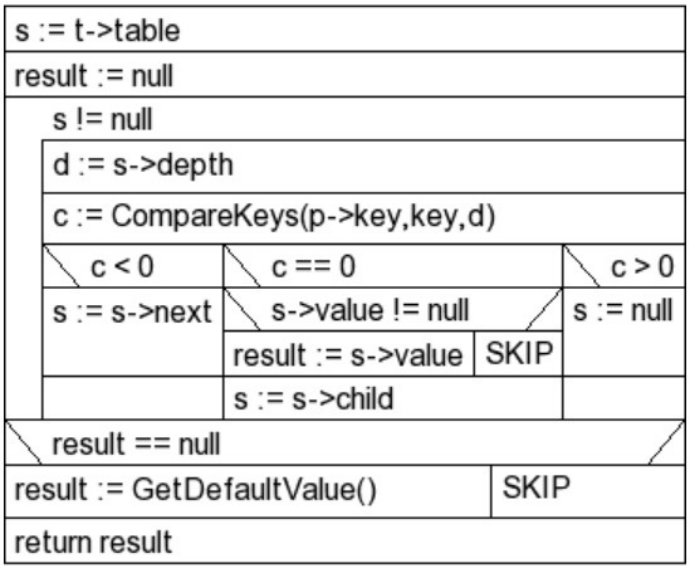
Adding entries to the lookup tables happens the way the user implements it in the controller. The user has to define their own keys and values, then call the *exact_add, lpm_add* or *ternary_add* functions. The rest is handled by the interface.

In this technical report, I'd like to refrain from getting into the lookup tables' details; for the sake of brevity I'd leave only the structogram of the lpm_add and lpm_lookup functions here. (They were created to provide a visual representation of my code's workings, since *lpm_add* is actually the most complicated function in the entire project.)

int lpm_add(lookup_table_t* t, uint8_t* key, uint8_t depth, uint8_t* value);



int lpm_lookup(lookup_table_t* t, uint8_t* key);



Getting the results

The packet can go through multiple *actions* of the switch before it is completely processed. The switch doesn't call any functions when it's finished, but updates the appropriate *field* (on the data plane) instead. Once it is done, the P4SwitchNetDevice will then acquire the port number on which the packet should be forwarded, provided that the packet hasn't been dropped.

Testing

There were multiple ways to test the correctness of my code during development. The one I've used most frequently was to simply run the same simulation with OpenFlow switch along with the (functionally same) P4 switch I was working with, and compare the results.

I could print out certain lines to know which part of the code was being executed...

OpenFlow switch

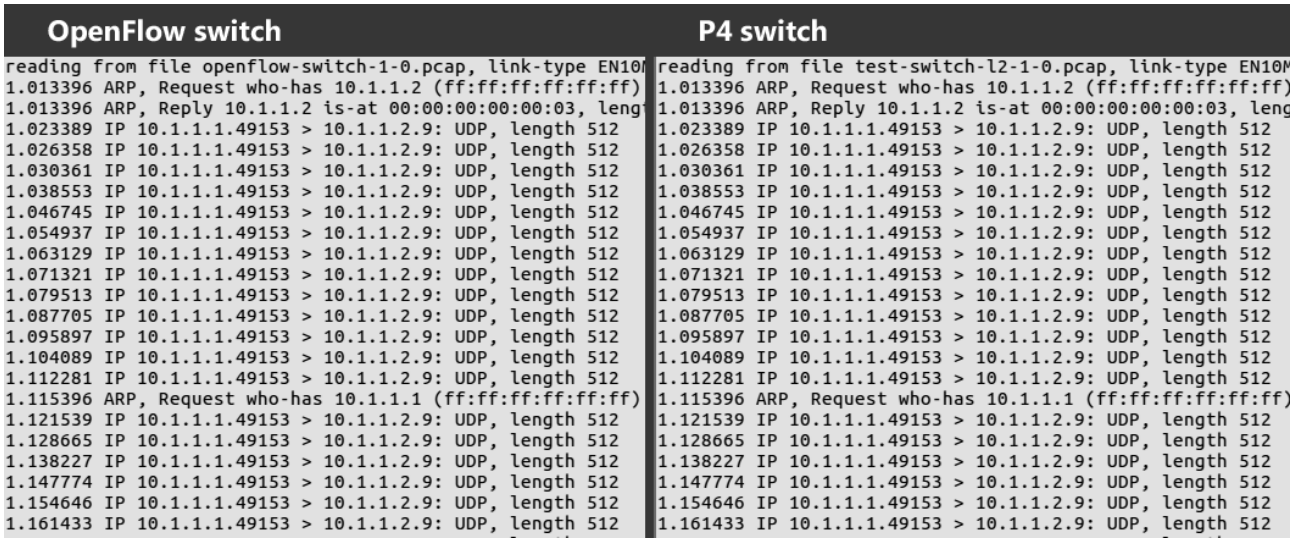
```
Create nodes.
Build Topology
RegisterProtocolHandler for ns3::CsmaNe
RegisterProtocolHandler for ns3::CsmaNe
RegisterProtocolHandler for ns3::CsmaNe
RegisterProtocolHandler for ns3::CsmaNe
Assign IP Addresses.
Create Applications.
Configure Tracing.
Run Simulation.
---------------------------------------
UID is 1
Received packet from 00:00:00:00:00:01
Creating Openflow buffer from packet.
Parsed EthernetHeader
Parsed ArpHeader
Matching against the flow table.
Flow not matched.
Sending packet to controller
Setting to flood; this packet is a broa
Added new flow.
Flooding over ports.
Sending packet 1 over port 1
Sending packet 1 over port 2
Sending packet 1 over port 3
Learned that 00:00:00:00:00:01 can be f
---------------------------------------
UID is 3
Received packet from 00:00:00:00:00:03
Creating Openflow buffer from packet.
Parsed EthernetHeader
Parsed ArpHeader
Matching against the flow table.
Flow not matched.
Sending packet to controller
Added new flow.
Sending packet 3 over port 0
Learned that 00:00:00:00:00:03 can be f
---------------------------------------
```

P4 switch

```
Create nodes, 4+1.
Build Topology.
Installing switch device on node 4.
Setting properties for table 0.
Setting properties for table 1.
Adding SwitchPort 0x25b6ab0.
RegisterProtocolHandler for ns3::CsmaNetDevice
Adding SwitchPort 0x25bca90.
RegisterProtocolHandler for ns3::CsmaNetDevice
Adding SwitchPort 0x25ba8a0.
RegisterProtocolHandler for ns3::CsmaNetDevice
Adding SwitchPort 0x25750e0.
RegisterProtocolHandler for ns3::CsmaNetDevice
Setting callback functions.
Assign IP Addresses.
Create Applications.
Configure Tracing.
Run Simulation.
-------------------------------------------
UID is 1
Received packet from 00:00:00:00:00:01 looking for ff:ff:ff:ff:ff:ff
Creating P4 buffer from packet.
Handling packet arriving at port 0.
  :::: executing table smac
   ::: exact_lookup didn't find match for key [0,0,0,0,0,1]
    :: executing action mac_learn...
     : Learned that port 0 belongs to MAC address 00:00:00:00:00:01
  :::: executing table dmac
   ::: exact_lookup was successful with value 3 and key [255,255,255,2
    :: executing action bcast...
(broadcast) Sending packet 1 over port 1
(broadcast) Sending packet 1 over port 2
(broadcast) Sending packet 1 over port 3
-------------------------------------------
UID is 3
Received packet from 00:00:00:00:00:03 looking for 00:00:00:00:00:01
Creating P4 buffer from packet.
Handling packet arriving at port 1.
  :::: executing table smac
```

...or I could use NS-3's built in tracking tools; ASCII tracking and pcap tracking.

I've also tested the different “table-operators” (*_add*, *_lookup*, and *_remove* functions) I've created, the test simulation for that is also included to this project's files.



How to use the P4 switch in NS-3

Finally, I'd like to write a few words about how to make this P4 module work in practice. This isn't going to be a step-by-step user manual – though I've detailed it in the Hungarian documentation appropriately – just a quick overview of the process.

To be able to run a simulation with a P4 switch, the user first needs to create a new module, and copy the eight t4p4s-generated files (that represent the switch itself) into the *model* folder of the p4 module. These files need to be modified a little, by changing the *#include "dpdk_lib.h"* into *#include "p4-interface.h"* (the current version of t4p4s doesn't have this option).

After this, the programmer (i. e. the user) can write their own NS-3 simulation and create their P4 switch via the P4SwitchHelper class. The Install method of this class requires two functions to be passed down as pointers: one of them will initialize the lookup tables, while the other is the controller (or the so-called digest). It is a callback function through which the user can define how the switch behaves. For example, the user may define here a custom MAC-address learning method for incoming packets.

For demonstration purposes, I've attached to this project samples for both the (compiled) P4 switch and the entire NS-3 simulation.

Conclusion

All in all, though the project definitely has some areas it could be improved on, its basic functionality is achieved. It can forward packets on L2 and L3 as well, and the user can define custom switch actions and make them work within the simulation.

References

P4SwitchNetDevice source files: https://github.com/mrosan/P4SwitchNetDevice.git
NS-3: https://www.nsnam.org/
P4: https://p4.org/
P4@ELTE (t4p4s): http://p4.elte.hu/