**A Programmable Forwarding Plane Simulator for P4**

A Project Report Presented to

The faculty of the Department of Electrical Engineering

San José State University

In Partial Fulfillment of the Requirements for the Degree

Master of Science

By

*Aashav Panchal 011463361*

*EE297B Section 3, Spring’18*

*aashav.panchal@sjsu.edu +1(669)-271-5189*

*Swapnasheel Dattu Sonkamble 011452610*

*EE297B Section , Spring’18*

*swapnasheeldattu.sonkamble@sjsu.edu*

**Department Approval**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Prof. Morris Jones (signature)* Date

Project Advisor

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Prof. Thuy Le (signature)* Date

Project Co-Advisor

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Dr. Birsen Sirkeci (signature)* Date

Graduate Coordinator

**Department of Electrical Engineering**

Charles W. Davidson College of Engineering

San José State University

San Jose, CA 95192-008**4**

**Abstract**

*The project aims to develop a P4 application simulator with user-defined topology, capable of self-deploying P4 code on nodes for better debugging and testing purposes. The project also aims to understand the use of P4 language (programmable protocol-independent packet processors) in the forwarding plane and simulate the process for an enterprise level multi-layer architecture. This will allow P4 application developers to use the simulator tool prototype to simulate and debug their applications before hardware availability. Moreover, developers can also make use of the simulator tool to evaluate the actions of P4 based architecture for network design decisions. The P4 simulator environment developed in the project is based on the available software tools by the P4 community and other available environment sources. Blending all the resources together the project helps in reducing the complexity involved in simulating a P4 environment making it easy for users to test and analyse the effects of P4 code on a networking environment.*

**Acknowledgements**

We are extremely thankful to our project guide Prof. Morris Jones for providing valuable guidance and ideas throughout the project. He constantly encouraged us to explore new possibilities with the project implementation and guided us with possible test cases, ensuring the credibility of the project.

We are also thankful to Ms. Snehal Chavan and Mr. Omkar Desai for collaborating with us and giving us more insight into the P4 environment understanding. We would also like to thank Mr. Anurag Choudhary from Computer Engineering department for providing us insights on what is expected from a P4 simulator.

**Abbreviations and Acronyms**

**Table of Contents**

**List of Figures and Tables**

[Figure 1: Traditional Switch v/s P4 programmable 9](#_Toc513502782)

[Figure 2: Behavoiral Model (BMV2) switch architecture 13](#_Toc513502783)

[Figure 3: Compiler Data Flow[3] 16](#_Toc513502784)

[Figure 4: Compiler Structure[4] 17](#_Toc513502785)

[Figure 5: Mininet Emulator Environment 19](#_Toc513502786)

[Figure 6: Low-level API Template 20](#_Toc513502787)

[Figure 7: Mid-level API Template 21](#_Toc513502788)

[Figure 8: High-level API Template 21](#_Toc513502789)

[Figure 9: Converting File Format 23](#_Toc513502790)

[Figure 10: Defining Virtual Ethernet Interfaces 24](#_Toc513502791)

[Figure 11: Basic Work Flow 25](#_Toc513502792)

[Figure 12: P4 Runtime Architecture 30](#_Toc513502793)

[Table 1: OpenFlow Versions Timeline 27](#_Toc513502794)

[Table 2: APIs Features 30](#_Toc513502795)

# Overview

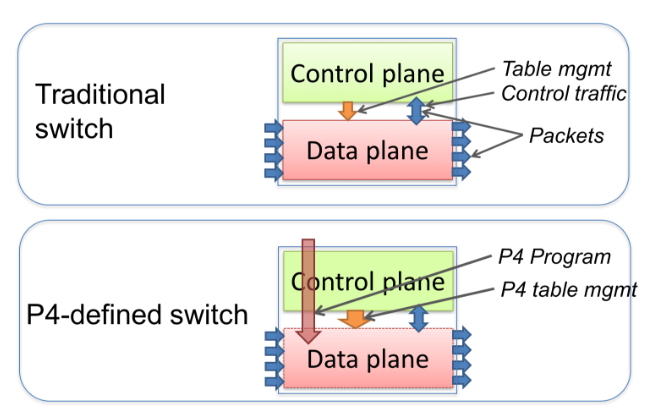
Over a decade the legacy network switches have over gone a makeover in its architecture. By the introduction of SDN (Software Defined Networking) the Networking industry experienced an era of decoupled software control plane. SDN provided a programmable control plane acting as a centralized brain to make decisions for the underlying forwarding plane. The essential parts of SDN called centralized control plane, northbound APIs and Southbound APIs offers a centralized view of the overall network and enables the network managers to handle the routing.  
  
SDN played an essential role in changing the perspective of the view of networking elements like switches, routers and its underlying architecture. It opened many possibilities by providing varied business benefits in the networking industry and helped provide network intelligence along with reducing capital and operational costs in laying out and managing enterprise networks. SDN came as an innovation which the networking industry needed. This was not enough to change the bottom-top approach of network industry had been following.   
  
Moving down to the data plane which acts as a forwarding element had been left blindly static over years with pre-programmed ASICs. With the emerging changes in Networking, there has been a need to make it easily adaptable to changes in match-action tables and packet headers. P4 language expresses how packets are processed by the data plane of a programmable forwarding element.  
  
The project aims to develop a P4 application simulator with simple topologies along with user-defined topology, capable of self-deploying and testing P4 code on nodes for better debugging and testing purposes. The project also aims to understand the use of P4 language (programmable protocol-independent packet processors) in the forwarding plane and simulate the process for general testing topology architectures. This will allow P4 application developers to use the simulator tool prototype to simulate and debug their applications by testing using custom-made packets and its behaviours.

**Chapter 1**

# Introduction

P4 language was developed out from the paper on “Programming Protocol-Independent Packet Processors” - https://arxiv.org/pdf/1312.1719.pdf [1]. Initially focused on designing programmable switches, it’s scope since then has expanded over a variety of networking devices and is now termed as a programmable language for multiple targets.

The targets defined to be programmed by P4 are composed of both control plane and the data plane. Although much of the control plane functioning has been managed by SDN recently, P4 aims at specifically designing the data plane of the targets. Furthermore, P4 also plays a role in defining the interface which can communicate with the new programmable control plane and P4 defined forwarding planes.



**Figure 1: Traditional Switch v/s P4 programmable**

P4 being relatively new in the field is in rapid developing phase with upcoming new version making the programming language more stable and adaptable to the industry in general. Aimed at changing the old bottom-top approach it promises a revolutionary change by defining a new top-bottom approach making it easier for network administrators to define new rules and protocols and making the transition to them faster and convenient process.

Although knowing its capabilities and industry revolutionizing power, it is very important to focus on the fundamentals of its mechanisms by which the P4 language can be compiled, integrated on targets and executed to perform its functionalities.

Being a hardware-oriented programming language, it is very important to ensure the validity of the code before implementing it directly on to the hardware. To ensure the reliability of the code, many test cases are required and thus it needs a stable environment to perform the test on.

The P4 community is growing at a rapid phase and is getting enormous contributions to support the platform. With multiple versions progressing towards the stability it lacks a generalized platform for the users to test and validate their code in a stable virtualized environment.

This project targets at building a generalized virtual simulation environment for P4 language, based on the aggregation of stable underlying compilers, software switch, and virtual emulator. It would help reduce the complexity of testing P4 code on varied virtual network topologies to help user debug the outcomes of the generated P4 code before implementing it on the hardware.

**Chapter 2**

# Software Tools for Simulator

As P4 language is defined to be hardware oriented and focussed on programmable targets, these programmable targets could be any hardware which could make use of P4 to tell how to process the packets. Some of the well-defined Programmable Network devices are [2] :

**• PISA Flexible Match+Action ASICs** - Intel Flexpipe, Cisco Doppler Cavium (Xpliant), Barefoot Tofino, …

**• NPU** - EZchip, Netronome, …

**• CPU** - Open Vswitch, eBPF, DPDK, VPP…

**• FPGA** - Xilinx, Altera, …

To be able to create a simulator tool for the users developing P4 code for these programmable targets, it is very important to know the types of target and ways of executing the P4 code on them. This would help to recreate the software-based environment enacting the actual scenario of the programmable targets.

Thus, the most important software tool required for the simulation environment is a software switch that could provide the exact functionality as that of the programmable targets the users are expected to work on and supports all the P4 functionalities. Just like OpenVswitch (OVS) which is a software switch running in hypervisor highly incorporated by SDN. OVS became so prevalent as it can support port mirroring, VLANs, LACP, NetFlow, sFlow, and various other protocols especially OpenFlow which is a fundamental part of SDN.

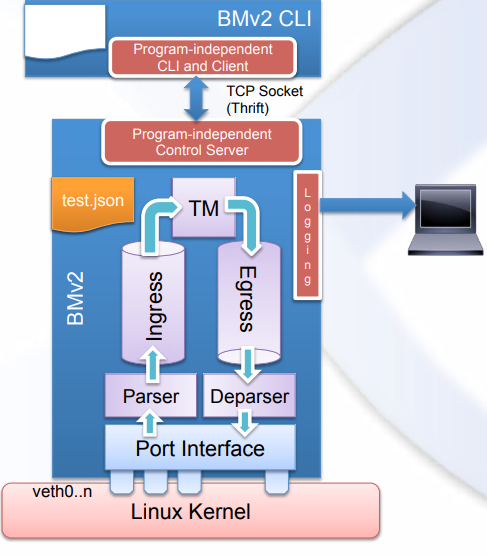
Also as every target acts differently and has different interfaces it is very important to have a compiler which could ensure the program execution on the target. Thus, a compiler is must in the environment for having the software switch perform required functions of the code.

To sum it all up a crucial part of the simulation is filled in by a proper network emulator which could put together all the components and offers a reliable virtual networking environment to perform tests on different scenarios. The P4 simulation environment is based on chosen stable software tools which could meet the expected requirements.

**2.1. Behavioral Model (BMV2)**

The behavioral model version 2 of the P4 termed as BMV2 is the defined software switch model enacting the PISA switch. The PISA switch is the Protocol Independent Switch Architecture model which does not have a defined set of protocols and works based on the user.

The new BMV2 is a static model designed to provide a virtual PISA switch functionality as a software model for the users to perform tests and use in a virtual environment. The switch carries a good architecture model as shown in figure2.



**Figure 2: Behavoiral Model (BMV2) switch architecture**

The BMV2 switch architecture comprises the following elements:

1**. Port Interface** – Port interface is responsible for creating a pair of virtual ports binding connection between the Linux kernel to the switch ports through which the traffic can be sent and received through the switch. It provides the essential binding to the switch required by the system software switch is operating upon. The number of interfaces can be defined by the user according to the requirement.

2**. Forwarding Elements** – Just like any other legacy switch forwarding plane, the behavioral model also consists of the basic forwarding elements structure including the parser sending the parsed headers to the ingress pipeline responsible for the match-action table functioning based on the rules defined by the input object json code, and forwarding it to the Egress pipeline for further action to finally deparsing all the actions attaching to payload for the responsible port to forward it.

3. **Thrift Port** – Each software switch consists of a thrift RPC server running on it which creates a thrift port at a default value of port number 9090. It is important to have a different port number if using multiple switches together. The thrift port allows running the CLI instance connecting the user to the switch interface.

1. **Debugger/Logging** – It is very important to be able to log the events to debug and explore them. The BMV2 has a very well defined systematic formalized module which can allow seeing all the tables and the packet parsers flow which go in. This is done by a very generic message bus called logger which can be initialized to view the logging events.
2. **Switch CLI -** BMv2 uses python based CLI to provide an interactive environment. It can be used to input programmable scripts in the running environment. The CLI contains many available generic commands for P4 objects which makes it very easy to interact with match-action tables, counters etc. There can be multiple instances that can be started simultaneously. CLI instances are initiated via a thrift port and it is very important to have a unique thrift port for every CLI instance.

Along with all the above-defined Architecture parameters the P4 faces challenges to have control plane talk to the programmable data plane managed by the P4. As P4 is just designed to program the forwarding plane it needs to correlate with the available control plane for the network operations to function. The P4 switch needs to have a well-defined protocol model which allows programming the tables, retrieve the counters, set the meter etc. so the switch can function as programmed even with the control plane interaction.

## **2.2. P4 Compilers for BMV2**

P4 has recently introduced its new version P4\_16, which was earlier using the version P4\_14. Though the new version has extensively new functionalities and built in libraries to offer, currently both the versions are being used. Thus, with the new versions there has also been change in the compilers for the P4.

After the introduction of the P4\_16 a new reference compiler was introduced with it. The new compiler is made capable to both the versions of the P4 language and is thus a convenient option to use for the P4 simulator environment.

**P4\_16 Reference Compiler - p4c**

Compile both P414 and P416 programs, P416 is incompatible with P414 and requires a migration path for the transition. The new compiler can convert the rich sets of P414 code into P416 when compiling the code.

Free and open source with vibrant community issuing multiple commits and contribution for providing a stable compiler.

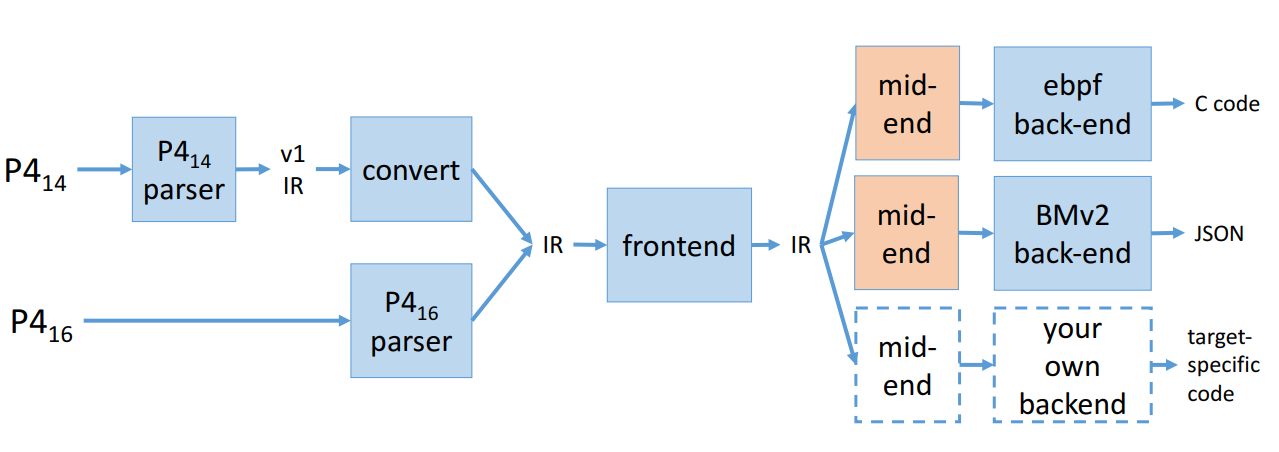
The new compiler was required to meet upcoming compiler goals, the most important being supporting upcoming versions of the P4 language. It was built simultaneously when the P416 was developed. The compiler should also can support multiple back-ends like ASICs, FPGAs, NICs and the software switch used for testing P4 codes.

Generating codes which different targets can support with a common compiling interface tends to be an important goal for designing the new compiler.

It is also very important for the compiler to support other software tools being commonly used with P4 in developing programmable forwarding plane like debuggers, control-plane support, IDEs etc. This could help save the complexity in providing a testbed for the users implementing P4 in their environment.

The p4c compiler is designed with an open-source front end, with ability to provide the user defined backend along with the backends provided by the developers. P4c is defined by extensible architecture to add new transformation easily and is built using modern compiler techniques like immutable intermediate representation, visitor patterns, string type checking etc.

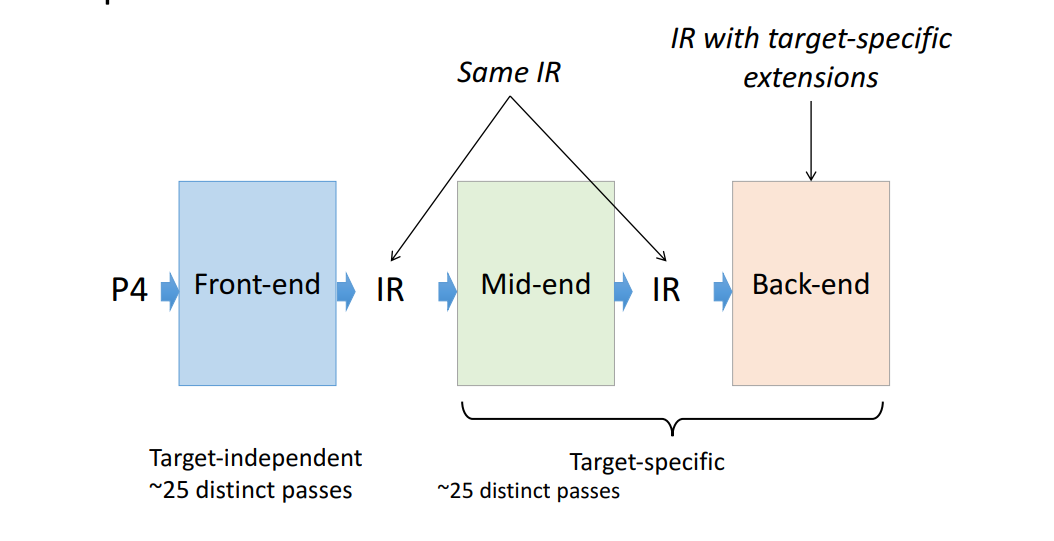
Though the only place where the compiler lags is the comprehensive testing phase. For any compiler to be declared healthy and completely stable it must go through at least 10,000 test cases whereas p4c just seems to have 1000 test cases available on repository. Though with widespread community and developers it would not take longer to reach the healthy phase.



**Figure 3: Compiler Data Flow[3]**

The p4c has 2 parsers one parsing the p4\_14 version and the other parsing the new p4\_16 version. the p4\_14 version parser converts the p4 16 representation and then on the same representation is used for further process. P4c provides with pluggable system where the mid-end and the back-end can be written by the users. Currently the p4c provides with 4 defined public back-ends:

* **ebpf back-end** – Generates c code which can be further compiled to ebpf and run on Linux kernels.
* **BMv2 back-end** – Generates the JSON representation which can be ingested by the behavioral model to test the P4 programs on the software switch.
* **P4c-graphs back-end** – Used to generate the graphical visual representation of the top-level control flows of the p4 program.
* **P4test back-end** – Used for testing, learning and debugging the internal of the compiler using the P4 translator.



**Figure 4: Compiler Structure[4]**

The p4c compiler is divided into 3 different blocks making it a unique structure:

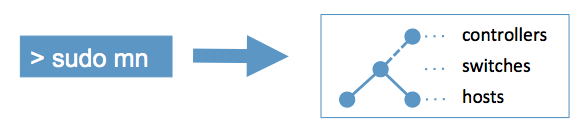
* **Front-end –** It is completely target independent block and takes care of validating program, conducting type checks on the program and makes clear program semantics, translating code in to homogeneous simple cases. It is a service provided as a framework to users.
* **Mid-end –** Has 25 distinct passes, and is a target specific block. It comprises of library of pre-built passes. It lets the user mix-match these policies to create their owns.
* **Back-end –** This target specific blocks allows user to define their own backend based on the provided framework.

The compiler is based on C++11 using the garbage-collection, with a very clean independently defined front-end, mid-end and back-end. The IR’s acting as data-structure defining the objects used are designed to be immutable and can be shared safely making them easily extendable.

## **2.3. Mininet Network Emulator**

## 

Mininet is a widely known network emulation system. It runs on Linux kernel environment deploying virtual hosts, switches, routers, and links between them. Mininet provides a complete virtualized network environment with hosts that can be accessed and programmed by the user using ssh. The lightweight virtualization makes the functioning very fast even on normal Linux machine having low configurations making it an ideal choice for research and testing purpose.



**Figure 5: Mininet Emulator Environment**

Mininet delivers following features making it an ideal choice for the P4 simulation environment:

• Starting and configuring simple network is very fast and easy making it simple for the user to run and debug the environment.

• It supports multiple real programs used in enterprise network environments like Wireshark, web-servers, iperf and other Linux based tools.

• It is well adapted to SDN controllers making it easy to customize packet forwarding capabilities from various controllers to choose from.

• The environment can be packaged easily to share and deploy on other systems in a very efficient way.

• Being an open source project mininet a wide community platform with access to a variety of software and tools compatibility, making it easily compatible with the Behavioral Model software switches used for the P4 test.

• Mininet has very well-defined Python APIs which are extensively used in the networking environment.

Though Mininet lags in many other areas like not being compatible with any other platform than Linux, making it single platform oriented language. It is also very important to create your own controller in mininet as it does not have an auto-define controller, making it very important for the user to have basic controller knowledge to choose suitable controllers and define rules.

As P4 simulation environment is primarily focussed on working on forwarding plane and control plane compatibility is not a crucial requirement and the programs can be tested with static rules, mininet emulation environment remains a valid choice with all the given benefits and BMV2 software switch compatibility.

The most important aspect of Mininet incorporated into the P4 simulation environment is the Mininet API. It is very important to understand the rich options offered by the python classes comprising the Mininet APIs. Mininet’s API is built under 3 primary levels [5]:

* **Low-level API:** The low-level API consists of the base node and link classes (such as Host, Switch, and Link and their subclasses) which can be instantiated individually and used to create a network, but it is a bit unwieldy.

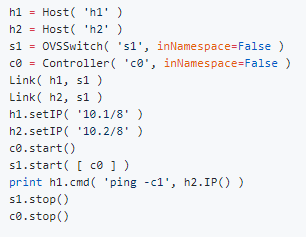
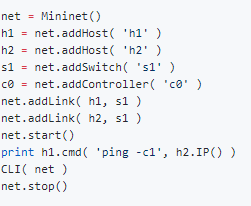


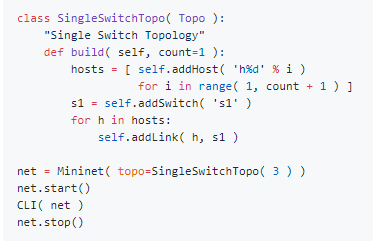
Figure 6: Low-level API Template

* **Mid-level API:** It adds the Mininet object which serves as a container for nodes and links by offering many methods (such as addHost(), addSwitch(), and addLink()) for adding nodes and links to a network. Mid-Leve API also manages network configuration, startup, and shutdown (notably start() and stop().)



**Figure 7: Mid-level API Template**

* **High-level API:** The high-level API adds a topology template abstraction, the Topo class, which provides the ability to create reusable, parametrized topology templates. These templates can be passed to the mn command (via the --custom option) and used from the command line.



**Figure 8: High-level API Template**

For providing the user with a simulation environment based on mininet emulation network it is very valuable to understand all the API levels in order to write scripts for building general networks according to user requirements. Amongst the above APIs the High-Level API providing the topology template abstraction and Mid-level API creating objects for nodes and links seems a suitable option to use for the P4 simulator.

**Environment Setup**

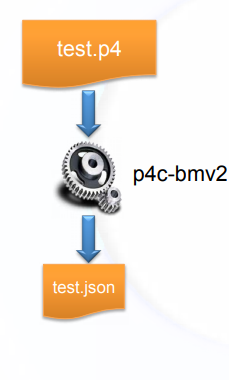
**Basic Work Flow for the BMv2 switch model:**

To build a simulation environment it is very important to understand the basic work flow to use a BMv2 switch. It defines the steps and the software tools required to fulfil the assigned task and get the test results. Understanding the basic work flow would help put together all the essential elements together into an automated framework to complete the expected tasks.

The basic work flow for implementing a P4 program on a simple BMv2 switch comprises of the following tasks:

* **Step 1: Compile the file**

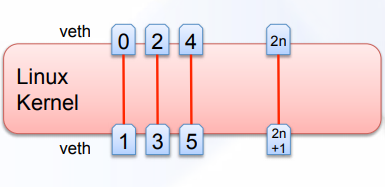
Based on the compiler version being used, the first step is converting the P4 program file into a format which can be fed to the underlying switch. The BMv2 software switch uses the JSON format and thus the file with the extension “. p4” is converted to the “.json” format by the compiler.



**Figure 9: Converting File Format**

**Step 2: configuring the virtual ethernet interfaces**

The BMv2 switch runs on a Linux environment and thus it can be provided the virtual ethernet ports connecting to the Linux environment. The user can define required number of virtual ethernet port pairs. The user also needs to make sure nothing unexpected is done by Linux on these ports like sending discovery packets etc which could hinder the P4 process running on the simple BMv2 switch architecture.



**Figure 10: Defining Virtual Ethernet Interfaces**

**Step 3: Run the simple switch model**

The next step is to start the switch model with the specified json file with the logger console by binding the virtual ethernet interfaces as defined in the previous step. Also, it is important to specify the thrift ports which enables the access of the switch via a CLI. It is very important to define all the parameters correctly while starting the switch model.

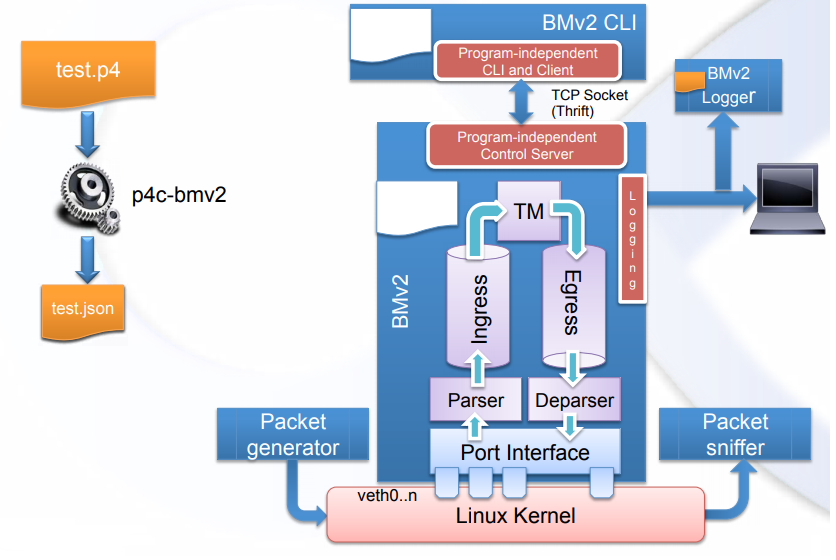
**Step 4: Starting CLI for accessing switch during runtime**

The python based CLI can be used to access the switch during runtime via the thrift server which is running on the switch. The thrift server assigns a port through which the switch actions can be monitored or changed during the runtime via a CLI interface. This port is called the thrift port and its default value is 9090.

**Step 5: Implementing traffic generator and capturing O/P packets**

Once the switch is running with the user defined P4 program and the available CLI interface, the user can now test the environment and validate the P4 code by sending in the random packets via a veth interface generated by the traffic generator using the scapy module. The packets flow through programmed forwarding plane and are captured on another terminal flowing out via a veth interface. The captured packets would help the user learn the working of the P4 code and the switch behaviour when the code is implemented.

The figure shown below defines the basic work flow of the simple switch model with all elements defined in an orderly manner.



**Figure 11: Basic Work Flow**

**P4 and SDN**

**Overview**

Targeting two very different elements of networking devices, P4 and SDN have evolved over the time to give a very transformed perspective of handling the routing and forwarding elements. Along with the novelty the transformed decoupled perspective of the control plane and forwarding plane also comes along with new challenges to the industry compelling some new standard protocols to corelate with the new decoupled architecture.

SDN is solely responsible for programming control plane and providing a centralised controlling brain to the underlying forwarding plane infrastructure. It has its own set of rules and protocols providing this transition. The question arises at the point of recognizing the need of a new programable protocol independent packet processor language termed as P4.

The sole purpose of SDN is to provide the users with a programable brain so they can create new rules, define new policies and operate the underlying network in a redefined way. Though it has met its requirements to a large extent there are domains which SDN cannot justify. Thus, it is very important to have an innovative technology obliging at taking care of those domains.

Any technology by itself might be superior to the other one but would never be enough to fulfil the standard requirements of a complete networking environment and thus would have defined in a way that it could collectively work to solve some challenges faced in the ever-growing networking environment.

**Alternative Future: P4**

SDN has a set of well-defined protocols through which it interacts with the underlying forwarding plane and helps the user define the new set of rules. Open Flow is the most well-defined communications protocol which gives the access of the forwarding plane to the centralized control plane to impart the user-defined rules in the underlying network.

Over the years the number of headers in the OF specifications has significantly increased as shown in Table [1]. With the change in the OF specification based on the new requirements over a period of time, the changes require changing the controllers implementing OF and also the forwarding element.

|  |  |  |
| --- | --- | --- |
| **Version** | **#Headers** | **Release date** |
| OpenFlow V1.0 | 12 | Dec 2009 |
| OpenFlow V1.1 | 15 | Feb 2011 |
| OpenFlow V1.2 | 36 | Dec 2011 |
| OpenFlow V1.3 | 40 | Jun 2012 |
| OpenFlow V1.4 | 41 | Oct 2013 |

**Table 1: OpenFlow Versions Timeline**

Changing the forwarding plane and the controller to work with the newly introduced OF version makes the transition a prolonged process. Even though after that the control plane and the data plane are not sufficiently decoupled and does not provide an easy way to modify the packet headers.

This makes it difficult to evolve the SDN switches and control protocols independently from the hardware. For the SDN switches to evolve further it is required to define the control protocols based on the requirements of the control program and not being constrained by the underlying switch hardware.

To make this possible we require the capabilities of:

• A user configurable packet parser which is not constrained to any specific header format.

• Offering flexible match-action tables in the forwarding plane, having the ability to assign multiple tables in series or in parallel matching in all the user-defined fields.

• Ability to copy, add, remove and modify the packet fields.

The new hardware chips are now allowing to make all the above requirements possible. Thus, it is very important to have a programming language for these chips which could provide the customer with a low-level microcode programmable interface irrespective of the vendor-specific interfaces.

P4 offers these hardware oriented requirements and accomplishes 3 important goals:

• Protocol Independent framework

• Target-independent language

• Field reconfigurability

**Solution**

In terms of SDN, OF 1.X protocols provides with vendor-independent APIs but they are limited to only the available legacy fixed function switches. So, for any updates to be made in the protocol itself there is a required resulting change in the supporting controller and the underlying forwarding elements. Thus, the process takes forever to put a new policy into actual implementation.

Whereas, the P4 on the other hand provides the capability of a protocol independence with no target dependencies and allows the reconfigurability in the fields. Though being just limited to forwarding targets it cannot fulfil the complete requirements of the networking environment.

Thus, to fetch the best results for the upcoming networking challenges it is important for the concepts of SDN and P4 to corelate with each other by defining some standard protocols that both the controller and the programmable forwarding element can understand and make use of for the fruitful new network architecture.

**P4 Runtime**

P4 Runtime API is specifically designed to provide runtime control of the programmable forwarding plane switches. Once the programme is compiled and loaded onto the hardware then you have the same tables you defined into the P4 program but it’s important to have the ability to change the tables and other parameters during the runtime, this is where the P4 runtime API comes into the picture.

There are many other approaches that provide the runtime control such as:

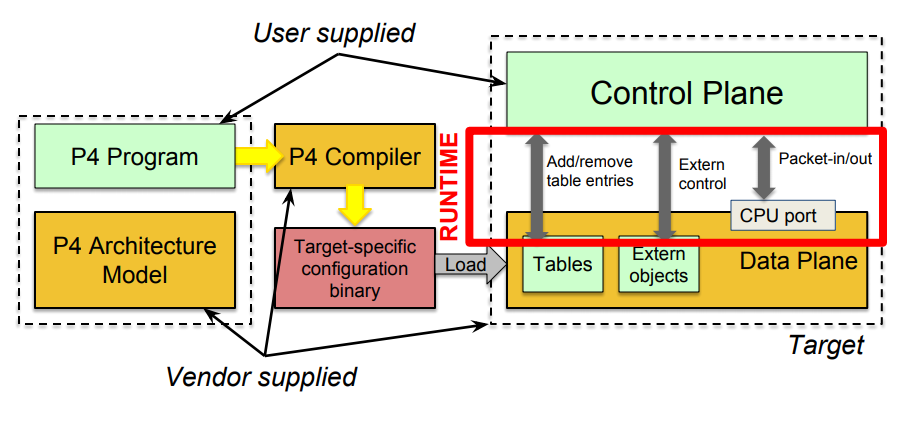
* P4 compiler runtime APIs – These are program dependent and makes it difficult to provision a new P4 program without restarting the control plane.
* BMv2 CLI – It is a program-independent entity but are target specific.
* OF - OF are target independent but are protocol-dependent entities.

None of the above APIs fulfil both the requirements of target-independence and protocol independence and thus P4 runtime API was developed.

|  |  |  |
| --- | --- | --- |
| API | Protocol-Independent | Target-Independent |
| P4 compiler APIs | X |  |
| BMv2 CLI |  | X |
| OF | X |  |
| P4 Runtime |  |  |

**Table 2: APIs Features**

P4 runtime provides a framework for the runtime control of P4 targets. It is a protobuf based API which means that the code is generated in CLI and CLI does the messaging in other different languages. The runtime API does not change with the P4 program. This makes it possible to push new P4 programs without recompiling deployed switches.



**Figure 12: P4 Runtime Architecture**

P4 runtime API enables interaction between the control plane and the forwarding plane. This is done by P4 info which is a contract between the control plane and the data plane. It captures the P4 program attributes required at P4 runtime such as table ID’s, actions, counters etc.

By learning these parameters, the control plane can add or delete the table entry into the forwarding plane.

This enables the SDN’s OF and P4’s programmable forwarding plane to work together in synergy.

**Chapter 3**

Testing the working of the simulation.

Unit testing using Python-scapy

Scapy is a python program used to generate packets to test working of the P4 application. Scapy is used to send and sniff packets on interfaces of the host. It is a tool used for manipulating interactive packets. This tool is based on Python language. Python’s syntax is simple and dynamic in nature, incorporated with features for interpretation; making an ideal choice for scripting and rapid development. Usage of scapy for unit testing was proven to be very essential to the testing phase of the project.

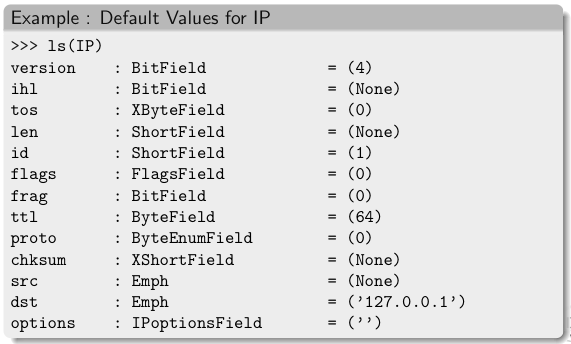
Why Scapy?

P4 gives you the ability to define packet headers. Therefore for unit testing, it is very important to have packet’s that can be understood by the P4 switch. With scapy, you can build exactly the packets you want. Below snippet of the code shows the creation of new header. With scapy, you can stack up 802.1q layer on top of TCP – and it makes no sense, but it does for someone working on some product of their own, because they can make sense out of it. You are free to put any value you want in any field you want and stack them on layers you want; scapy gives that flexibility.



Scapy uses some default values for packet fields. Some of which are

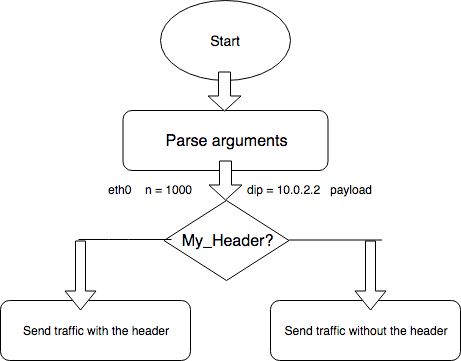
* IP source is chosen according to the destination and routing table
* Checksum is computed and added in packet
* Source MAC address is chosen according to the source interface
* Ether type values are decided according to the upper layer configuration



**Figure 13 Some common default values for scapy packets**

**Traffic generation using python-scapy**

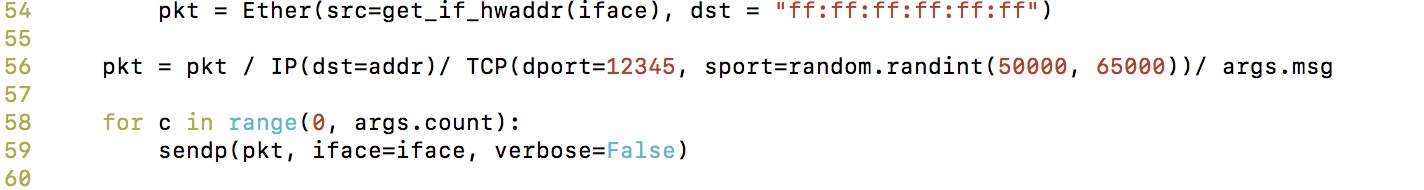
Scapy’s flexibility and scalability is used in generating *n* number of packets with and without the special header. We designed a script that could take in arguments like *destination IP address, port number, payload, and the count of packets* and send *n* number of packets to the destination IP address. Flow chart for traffic generator is as shown below:

****

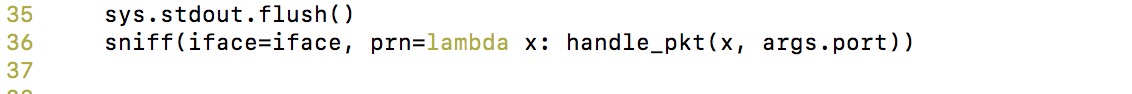
To verify the integrity of the P4 code and the topology build, both with and without the special header packets were sent to the destination.   
  
Similarly, a sniffing script was designed at the receiver end that sniffed and recorded incoming packets on a given interface. One can also use Linux tool *tcpdump* to record traffic packets on the receiving end. Tcpdump along with the *-w* flag allows one to store the recorded packets in a pcap file which can be used to inspect raw packets in more detail using a packet capture tool like ‘Wireshark’.

Some snippets below show few of the operations of scapy. Operations like creating packets, sending packets over the interface, and sniffing packets of the interface.

* Create and send packets

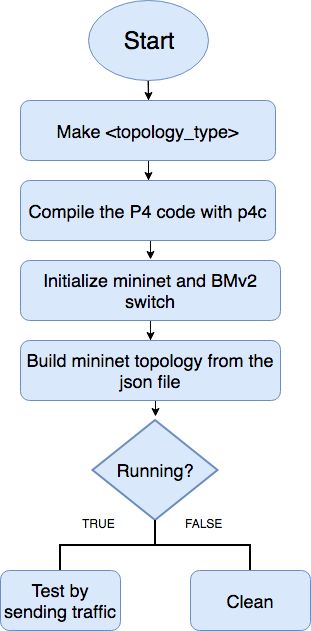


* Sniff packets



**Makefile**

The *make* utility in the Linux is a software tool used to maintain and manage computer programs with many component files. The *make* utility automatically detects pieces of large programs that need to be recompiled and issues commands to recompile them. Make reads from the Makefile (called the file descriptor) by default and compiles the scripts. Makefile was developed to orchestrate the overall working of the simulator. Any program that is capable of running in the shell can be written in a Makefile. Implementation of the Makefile in the project can be shown as per the flow chart below:



* Running *make <topology\_name>* will create a *topology.json* file
* After creating the topology file, we compile the p4 code in the directory
* As discussed before, we then create a mininet topology making the use of the BMv2 switches and the topology file
* Now we will have the mininet console and we can send in traffic to test the P4 environment and topology
* To end, run *make clean.* This will run *sudo mn -c* and remove associated dump files

Makefiles have a certain handful targets that are useful in some projects, like:

* develop: to set up a development environment
* serve: to start a development web server
* build: to build distributed artifacts
* test: run test scripts or automated test suite
* coverage: to produce a test coverage report
* deploy: automatically push to a staging area or production environment

**Code Implementation**

1. Importing libraries

At present, the BMv2 switch is only compatible with mininet tool. Therefore, like mininet libraries, we can import BMv2 libraries, and p4runtime\_switch libraries provided by the P4.org community <link to the p4runtime lib>. These libraries define classes and methods of emulating a virtual PISA switch architecture in the mininet tool.

1. Argument parser

The script requires a number of arguments to simulate the environment. For instance, two important arguments required are the P4 code and path to the behavioral model binary (discussed above) to simulate the environment.

Other arguments involve the topology.json file - to map the final topology to be created in the mininet simulation, pcap and logs directory paths, in order to store the pcap and log files generated during the simulation.

1. The topology file

The topology file is responsible for creating the simulation topology in the P4 simulator. This is a JSON file which is read in the python script and used a data structure to extract information and create simulation topology. When you run the simulator, the first thing that runs is the topology creation script that outputs in creating a file named topology.json responsible for the creation of topology in the simulation.

We have defined three most commonly used mininet topologies for testing, like:

* The Simple switch topology
  + This consists of a single switch with an option to add a number of hosts
  + To run this topology for your simulation, execute *“make simple\_switch HOST=4”.* This command will run the simulation creating one switch and 4 hosts topology
* The Linear topology
  + Running the linear mode will create ‘n’ number of switches and hosts
  + To run linear topology, execute *“make linear SW=3”*. This command will run simulation creating 3 switches connected linearly and having 1 host connected to each switch
* The tree topology
  + The tree topology takes in two arguments, the depth and the fanout of the topology
  + To run the tree topology, execute *“make tree DEPTH=3 FANOUT=2”.* This will create a topology that has 7 switches arranged with a depth of 3 and leaves switches will have 2 hosts

The *topology.json* file can also be used to create custom topologies. We have also added a feature for the user to input number of hosts, switches and their links (connections) to be able to simulate topology as per requirements.

1. The cli-commands.txt file (*s1-commands.txt*)

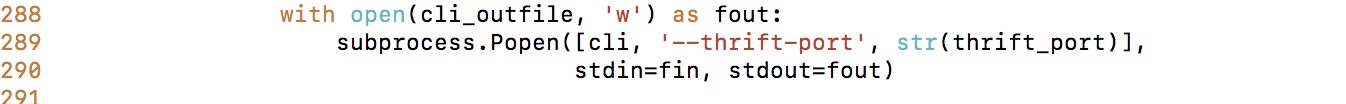
All the static rules for the forwarding of data are stored in the file named as the *“CLI-commands.txt”*. Each switch has its own cli-commands.txt file and this is also included in the topology.json file.

1. The exercise runner class

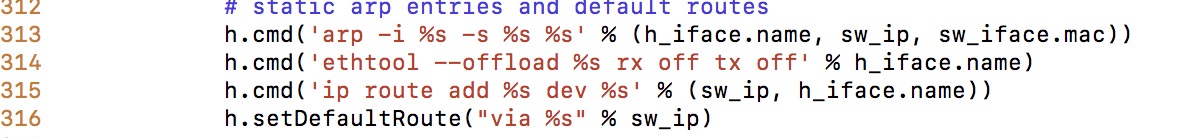
This class is responsible to parse the topology.json file, that we discussed above, and make lists of switches, hosts, and links among them. Moreover, the class is also responsible to run methods like create\_network, program\_hosts, program\_switches, and activate\_cli. Below we discuss these methods in detail.

"Create\_network" method: This method is responsible for creating the mininet simulation network object. It inherits the mininet generic topology class to sequentially assign IP and MAC address to the hosts. Furthermore, we configure these virtual switches with the P4 runtime and assign "thrift-ports" sequentially. As discussed, thrift-ports are the ports number to communicate with the P4 switch.

“Program\_switches” method: Once switches are configured with the P4 runtime, we call this method to program switches with static rules found in the “cli-commands.txt” file. This file can be located from the topology.json file as the switch. In addition, mapping of thrift port with the switches is also programmed in this method.



“Program\_hosts” method: We use this to configure ARP cache, default gateway, and routing information in the hosts. These entries are important for the hosts to understand the network and be able to communicate within the network.



“Activate\_cli” method: After configuring hosts and the switches, we activate the mininet cli console to navigate and make use of the environment for testing and debugging. Mininet console gives us the flexibility of navigating within hosts using “xterminal” and configure services in the hosts.

>>>>>>>>>>>>>>>

Topology creation: To further reduce the complexity of a user using the simulator, we introduced a terminal CLI where the user can select the type of topology he/ she intends to build to test the P4 code. Once the user selects the topology, the Makefile takes care of building the environment

**Comparison between traditional and proposed method of simulation**

As discussed above, in order to simulate and verify the desired P4 code, one has to install the BMv2 switch and the P4 compiler in the virtual machine. Having installed the complier, one has to create virtual interfaces and make sure no other service but P4 uses the virtual interfaces.

This can be a complex method for one who has less experience in configuring virtual interfaces. Moreover, this method isn’t scalable at all. If a developer wants to test his/ her P4 code for a datacentre network? Traditional method with one switch would not be helpful enough.

The proposed simulator solves both the issues regarding complexity and scalability. With the combination of *topology.json* file and integration of mininet, it is easy to simulate a network of P4 switches and hosts.

# References

|  |  |
| --- | --- |
| [1] | [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. |
| [2] | [Online]. Available: https://p4.org/assets/P4\_tutorial\_01\_basics.gslide.pdf. |
| [3] | [Online]. Available: https://p4.org/assets/p4-ws-2017-p4-compiler.pdf. |
| [4] | [Online]. Available: https://p4.org/assets/p4-ws-2017-p4-compiler.pdf. |
| [5] | [Online]. Available: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet#what. |
| [6] | [Online]. |