**ACKNOWLEDGEMENT**

I want to thank **Manipal Institute of Technology** for encouraging me to take up internship as a part of college curriculum. Asserting importance of practical experience for an engineer it motivated me to find an internship and **Manipal Institute of Technology** invited a plethora of companies and organizations that offered internship opportunities in myriad of domains and helped me choose my research internship at **Mitacs** in the domain of Computer Networks.

I want to express my gratitude to **Mitacs** for facilitating contact with **Ontario Tech University** a special thanks to it as well for funding my research and providing with all the necessary resources to facilitate the research internship and to connect us with fabulous mentors who have the best knowledge about their respective domains and to give a glimpseabout their culture in online mode.

I am using this opportunity to express deepest gratitude and special thanks to **Professor Shahrem Heydari** for giving me an opportunity to conduct research under his mentorship and for guiding me along the course of this internship imparting theoretical and practical knowledge gained over an impressive career as a network engineer and professor which helped me in learning about a new technology which is revolutionizing networks.

**ABOUT THE ORGANIZATION**

**Mitacs** is a non-profit national research organization that, in partnerships with Canadian academia, private industry and government, operates research and training programs in fields related to industrial and social innovation. Mitacs was founded by Canadian mathematicians in 1999. The organization, whose name originally stood for "Mathematics of Information Technology and Complex Systems", worked in the field of mathematical sciences and associated disciplines but has since expanded. In 2004, the Mitacs Accelerate program was launched and has since supported over 10,000 internships nationally.

The Faculty of Business and Information Technology (FBIT) at **Ontario Tech University** is committed to your success and takes pride in knowing our graduates will make a strong impact in their communities. FBIT offers innovative undergraduate and graduate degree programs in business and information technology. By placing a strong emphasis on how technology can enhance business opportunities, you will be prepared to launch a successful career in business and industry.

The Faculty of Business and Information Technology (FBIT) boasts several world-class research laboratories with infrastructure dedicated to conducting a unique blend of undergraduate and graduate research in a variety of research areas. Our faculty members are committed to research that can be translated to reality and continue to demonstrate this commitment through their research outcomes.

**Abstract**

This report highlights my work as a Research Intern at Ontario Tech University organized by Mitacs. I contributed under the mentorship of Professor Shahrem Heydari on Programmable Software Defined Networks. Being a vast domain, my area of research was to study programmability in data plane using a language called **P4** which is abbreviation for **Protocol Independent Packet Processor**. With everything being incorporated in a programmable stack, aim of P4 is to bring programmability in networks as well. Using this technology our networks would become hardware agnostic thus giving us full control of packet processing to the service redeemer.

**NETWORKING IN VIRTUALIZATION:**

* Underlay is just the physical elements in the network for example router, switches and basically all the underlying hardware on which the virtual network is running.
* **Fabric** is the main thing that makes up the virtual environment for networking.
* **TEP or Tunneling End Point** is the physical point through which the virtual network is connected to the underlying system.
* The intuition of tunneling end point can be eloquently described in the form of Russian dolls meaning that it all works based on encapsulation.
* We all know that virtual environment uses the hardware of the host machine which also includes the networking hardware and so basically the packets from the virtual environment are first sent to the host machine through the tunneling point.
* After that the host machine performs the networking as if it is its own packet records the response and sends it back to the virtual machine.
* Cognitively speaking virtual network is the brains in networking these days and the physical layer existing in network is just dumb.
* Virtualization helps in optimizing the network with advantages like scalability.
* Now taking about the virtualization of network using virtual machines the main skeptical thing is how these virtual machines communicate with each other like we can do in physical network with ethernet switches.
* This roadblock can be solved using virtual switch which is nothing but a software program in the virtual machine which helps in inter-network as well as intra-network routing.
* It intelligently forwards the traffic in the network.
* Now communicating with the outside network, you need a physical adapter which can route your requests outside the virtual network.
* Each virtual machine are assigned ports in the a network through which they can communicate using the virtual search
* Uplink in virtual network jargon means the adapter through which they can communicate to the outside network or between the virtual machines on different hosts.
* Virtual networking is a boon in testing the various networks that can be build and tested before bringing it into market.

**NETWORKING MODES IN VIRTUAL BOX:**

* **NAT**: This networking option basically masks the ip address of the virtual machine to that of the host device whenever it needs to contact the outside network. There is a dedicated NAT engine which performs this translation. In this mode host and any other external machine cannot access the guest os running on the virtual machine.
* **PORT FORWARDING:** In this method we set a port on the host machine to communicate with the guest running on virtual machine. We give instructions on the virtual box as to which port to on host machine is assigned to communicate with the guest machine. Then messages received on the host port will be forwarded to its corresponding on the guest.
* **NAT NETWORK:** This option enables the virtual box to create a NAT network in which all the guests can communicate with each other. So there is a common NAT gateway through which all the guests can communicate to the external network and for this the mechanism is same as the NAT protocol described earlier.
* **BRIDE NETWORKING:** In this type of communication option the virtual network itself sets up a port alongside the host port which filters the packets to virtual machine and host and directs them accordingly. Also, it handles the communication of the virtual machine with the external network as well. We can tell that there are 2 different machines are communicating with the network.
* **Internal Network:** In this network mode the virtual machine is unable to connect to the network of the host but multiple virtual machines or say guest OS can connect with each other. The only way to connect to the outside world is through host and this mode prevents the virtual machine to connect to the host or any other external host.
* **Host Only Network Mode:** This network mode is similar to that of the internal network but the only addition is that of host in the internal network.
* **Router Solicitation:** This packet is encountered when a host asks all the routers to show their presence in the LAN.
* **SSDP:** Simple packet discovery protocol is used by the devices in the network to know about the

services that are offered by the hosting machine.

* Incorporating any change in our networking capability according to our use case takes a lot of time.
* This time is ironically consumed in developing hardware rather than software.
* Current scenario of data plane is that the chip vendors are currently governing how our networks must behave by specifying how their hardware is processing packets.
* The aim of data plane programming is to enable us to define how packet processing should take place fulfilling our requirements.
* A thought of unfairness sparked the idea of P4 language as each domain specific implementation has its own programmable hardware and so why not the networking domain.
* P4 is on the path to revolutionize networking domain with its PISA architecture which is like fixed function chips but the only difference here is that everything is programmable.
* Explicit congestion notification (ECN) enables end-to-end congestion notification between two endpoints on TCP/IP based networks.
* ECN notifies networks about congestion with the goal of reducing packet loss and delay by making the sending device decrease the transmission rate until the congestion clears, without dropping packets.
* ECN-capable senders mark packets as ECN-capable. If a sender is not ECN-capable, it marks packets as not ECN-capable. If an ECN-capable
* of a switch, the switch marks the packet as experiencing congestion. When the packet reaches the ECN-capable receiver (destination endpoint), the receiver echoes the congestion indicator to the sender (source endpoint) by sending a packet marked to indicate congestion.
* **Load balancing** refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool.
* **The aim of load balancer is to monitor the traffic going to the server pool which is connected to it, and redirecting the requests to appropriate server which has less load as compared to other servers.**

**P4**

* P4 stands for **P**rogramming **P**rotocol-independent **P**acket **P**rocessors.
* P4 is a language for expressing how packets are processed by the data plane of a programmable forwarding element such as a hardware or software switch, network interface card, router, or network appliance.
* Sole aim of P4 is to program the data plane of a target and if the target has control plane application as well P4 is not concerned with it.
* In a traditional switch the data plane implementation or functionality is defined by the manufacturer and based on that we must develop application which is not a promising way in a dynamic industry, and it incurs huge costs for companies.
* A P4-programmable switch differs from a traditional switch in following 2 ways:

1. **Data plane functionality** is not fixed in advance rather it is defined by a P4 program which does the job of configuring switches which does not have any prior knowledge of network protocols.
2. **Control Plane** communicates with the data plane using the same channel as fixed function chips, but the set of tables and other objects in the data plane are no longer fixed as they are defined by P4 program.

* Compiling a set of P4 programs produces 2 artifacts:

1. A **data plane configuration** that implements the forwarding logic described in the input program.
2. An **API** for managing the state of the data plane objects from the control plane.

* The computational complexity of P4 program is linear in the total size of all headers.
* **Header** data type facilitated the definition of packet protocol headers. It has an implicit validity field indicating whether the header is a part of a packet. This can be checked using standard methods such as **setValid(), setInvalid(), isValid().**
* **emit()** method is used by the Deparser to include all the valid headers into the packet.
* A P4 program can easily manipulate the headers of a packet by setting and unsetting the validity bit of a header.
* Every type of header that is used by the packet must be a part of P4 program.
* Based on previously parsed header information any number of further headers can be extracted from the packet.
* Unlike other programming languages P4 does not allow the programmer to create loops.
* Match-action tables are defined within the control blocks and invokes actions depending on the header and metadata fields.
* Its corresponding table entries that are to be made to govern the switch are provided by the control plane.
* Match-action tables call appropriate actions that are mapped to by the key of the match-action table.
* P4 library has 3 standard match types: **exact, lpm, ternary.**
* Along with all the actions that are defined by the programmer there is also a default function that can be executed .
* Externs are used to provide additional functionality to P4 language. They need to instantiate before using their constructor method. Other methods can be called using the instantiated object.
* Externs can store additional state between packets. This state is accessible to either control plane or data plane or both.
* These externs are defined in vendor specific API and it can also manipulate the payload data meaning the data that is being carried in the packet from sender node to receiver node.
* P4 compiler translates the high level P4 programs to target specific binary configuration that can be executed on P4 targets.
* Basic workflow of the compiler is divided into 2 parts:

1. **Front End:** This part is common for any target which consists of syntax and target independent semantic analyzer which translates the code into intermediate representation.
2. **Back End:** This part takes the intermediate form produced in the previous step as input and translates it into target specific binary representation.

* The intermediate representation is in JSON format for P416.

**BENEFITS OF P4:**

* **Flexibility**: P4 makes forwarding policies expressible in contrast to fixed function chips which expose fixed function forwarding engines to their use.
* **Expressiveness:** P4 can express hardware agnostic sophisticated algorithms along with portability among the devices which has P4 compatibility and support
* **Resource mapping and management:** P4 programs describe storage resources abstractly (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources and manage low-level details such as allocation and scheduling.
* **Software engineering:** It provides benefits such as type checking, information hiding and software reuse.
* **Component Libraries:** The underlying hardware has its own indigenous architecture and compiler wrapped around hardware specific functions, to be used by high level P4 program.
* **Decoupling Hardware and Software Evolution:** Target manufactures can use abstract architecture to further decouple the low-level architectural details from high level processing.
* **Debugging:** Target manufactures can provide software environment where the designs can be tested before being deployed into the field.

**ARCHITECTURE:**

Architecture is an important aspect of P4 language because it acts as a contract between the program and the hardware.

The hardware provider provides compiler and the architecture on which the compiler is built.

From this we can say that P4 programs are not portable across different architectures.

The behavior of a P4 program can be fully described in terms of transformations that map vectors of bits to vectors of bits.

**P4 PROGRAM STRUCTURE:**

1. Headers
2. Parser
3. Check-Sum Verification
4. Ingress Processing
5. Egress Processing
6. Check-Sum update
7. Deparser
8. Switch

**EXPLAINING THE STRUCTURE:**

1. **HEADER:** This part describes the headers that should be present in the packet. It basically instructs the parser what headers are to be parsed and what information it should contain based on that the parser automaton can be constructed. The keyword in P4 to define a header is **header,** and it is a structure which is like a structure in C language. All such headers are to assemble into a single structure where only the type is to be declared and no need to instantiate it.
2. **Parser:** It parses the incoming header and depending on the parsing conditions it decides whether to drop the packet or to pass it further for ingress processing. It is same as an Automata system where the parser advances to a new state depending on the condition of the data whether it contains appropriate information required to advance to the next step. The parser method takes input the **packet**, along with **header outline declared as headers**, **metadata** and a special struct called **standard\_metadata\_t.** The states are defined using the keyword **state** followed by the name of the state and they are to be arranged in chronological order of their occurrence.
3. **Checksum Verification:** This part is important as per the fact that there are chances of the packet getting corrupted during transit because of the external noise which it encounters while passing through physical link. So after the packet is parsed and is accepted and before ingress processing a checksum verification is performed to verify that the packet is without any external attenuation.
4. **Ingress Processing:** In this phase all the packet processing that is to be needed like updating TTL value, changing the source and destination address and many more. This phase also contains the match action pipeline containing actions and tables which act as switch case as in C. Hence, it contains a key and based on that the actions are called. There are different kinds of tables which can be looked up in the documentation available on the official P4 language repository.
5. **Egress Processing:** This phase refers to the processing that is to be done before the packet goes out of the switch and into the external network.
6. **Checksum Computation:** As we all know that once the packet passes through ingress and egress processing pipeline, there will be change in data in the packet and it is manifested that the checksum will change and hence a new checksum is to be computed otherwise the other switches and routers in the network would understand that the packet is corrupted and will drop it as they are unaware of the fact that the packet may not be corrupted and there must be some change in data during ingress or egress processing at a previous switch or router.
7. **Deparser:** This phase builds the packet and then pushes it out into the external network.

**STATEFULNESS OF P4**

* Counters, meters, and registers maintain state for longer than one packet. Together they are called stateful memories.
* They are managed by the compilers.
* Stateful memories are organized into named array cells and cells are referenced by its array name and index.
* These stateful memory resources are global that is it can be referenced by any table.
* Normally, multiple table entries, whether they are in the same table, may refer to the same cell. This is called **indirect access.**
* P4 also allows **direct access** where the stateful memory resource is bound to one table and each entry in the table is allocated its own dedicated cell in that memory.
* A **bytes type counter** gets incremented by the packet length in bytes whenever the count action is executed either implicitly or explicitly.

**Counters:**

* P4 program can only update the counter values and not read them whereas it is readable for the control plane.
* **Direct counters** are counters that are associated for each P4 table.
* Counters are only intended to support packet counters and byte counters or a combination of both.

**LIMITATIONS OF P4:**

* P4 as a general-purpose programming language is very limited.
* P4 is not a Turing-complete language; it is narrowly defined for performing data-path packet processing.
* Surprisingly, there are even many packet-processing tasks that cannot be expressed in P4.
* P416 supports extern functions or methods; these are computational functions that are implemented outside of P4 and can be invoked from P4 programs. There is currently an effort to standardize a set of such methods; however, each P4 target platform can provide additional extern methods, e.g., to model hardware accelerators. Invoking extern methods is one way that P4 programs can perform otherwise impossible tasks.
* There is no iteration construct in P4. Loops can only be created by the parser state machine. There is no support for recursive functions. In consequence, the work performed by a P4 program depends linearly only on the header sizes.
* There is no dynamic memory allocation in P4. Resource consumption can be statically estimated (at compile-time).
* There are no pointers or references.
* There is no support for multicast or broadcast. These must be achieved by means external to P4. The typical way a P4 program performs multicast is by setting a special intrinsic metadata field to a “broadcast group”. This triggers a mechanism that is outside of P4, which performs the required packet replication.
* P4 has no built-in support for queueing, scheduling or multiplexing.
* P4 is unsuitable for deep-packet inspection. In general, due to the absence of loops, P4 programs cannot do any interesting processing of the packet payload.
* P4 offers no support for processing packet trailers.
* All the state in a P4 program is created when a packet is received and destroyed when the processing is complete. To maintain state across different packets (e.g., per-flow counters) P4 programs must use extern methods. We expect the standard library to contain support for such persistent arrays (counters, registers, meters). Even given support for registers, one cannot iterate over all counters to compute statistics.
* There is no standard way to communicate between the data plane and the control plane; this is usually offered using extern methods (e.g., to implement “learning”).
* There is no support for performing packet fragmentation or reassembly; thus protocols such as TCP cannot be described in P4.
* There is no support for generating new packets (e.g., an ICMP reply), only for processing existing ones.
* **Despite these limitations, P4 is a remarkably useful language.**
* **We also expect that future evolution will expand its capabilities.**

**BEHAVIORAL MODEL VERSION 2:**

Behavioral model is at its roots a software switch compiled by the industry experts to emulate the environment of a programmable switch.

It basically provides a target for execution of p4 program.

It ingests a JSON data file which consists of table entries for the target switch and the pipeline that is to be emulated is also present in the JSON data that is generated by the p4 compiler.

Main aim of behavioral model is to prove that network programming can be made hardware agnostic.

Bmv2 is currently supporting v1 model architecture which has the following pipeline:

Diagram

Description automatically generated

The V1Model consists of six P4 programmable components:

* Parser
* Checksum verification control block
* Ingress Match-Action processing control block
* Egress Match-Action processing control block
* Checksum update control block
* Deparser

The switches provided by behavioral model are chronologically ordered based on their recommended use.

It is a c++ user-space software switch to emulate p4 data plane.

**WORK FLOW OF BMV2:**

**Diagram

Description automatically generated**

Innovation in the field of networking is algorithms being shifted from hardcoded into the hardware to a more software approach where the algorithms can be sent as programs to the routers or be executed at a remote location and the results being sent back to the requesting router.

Bmv2 comes with simple\_switch target, which implements the P4 v1.0 / v1.1 abstract device model using the bmv2 library.

For running p4 programs on different targets provided by Bmv2 and those can be specified in the compilation command.

After that we need separate commands to run bmv2 switch independently and then run the mininet environment inside the bmv2 and check the working of the switch.

A file called runtime\_CLI is used to populate the tables in the switch and there are appropriate commands for it which can be found in the runtime\_CLI readme file.

A commands.txt file can be made which can be fed to the runtime\_CLI file and will populate the tables accordingly and the switch will forward the packets accordingly.

The above approach is for the configuration that is not provided as runtime json file.

**For proper functioning of your p4 program:**

**topology.json:** This file specifies the topology that is to be used for the p4 program. It consists of all the switches that are to be present along with the address of their runtime json file. Hosts and their mac address and ip address and interface information along with the commands that is used to initialize the networking information. This should be present in a folder named topo.

**Screen Shot of a topology.json file:**

**Text

Description automatically generated**

**Si-runtime.json:** This file contains basic build information one of them being the compiled P4 program json file route and other necessary information. Along with this it consists of table entries which are the most important part of our network implementation. In its skeleton, table entries consist of table name, action name depending on the packet headers which is also a part of this json file, action name this points to the action that is to be executed when an entry in the table matches a particular expression mentioned in the key section of the table.

**Screen Shot of a Runtime JSON File:**

**Text

Description automatically generated**

**Makefile:** This file helps in the compilation process which is basically a collection of compilation commands. It cuts unnecessary slack of writing multiple commands to run our p4 logic on bmv2 switch.

**<Filename>.p4:** The heart of our network program file which guides the switch on how to implement data plane.

Well, we know that it is a software emulated switch with no actual physical ethernet ports, so at the roots it is implemented just as any other process and the ethernet ports are virtual interfaces to which the operating system might instruct them to act as ports and to process packet via the P4 program.

As we are treating it as a real switch we can actually monitor the packets going to and from these virtual ethernet ports using **tcpdump or tshark commands.**

**BMV2 Mininet Environment:**

This is generated using Mininet python API. All the necessary JSON files are provided as command line inputs and are used in initializing the environment.

**Analysis of P4\_mininet.py file:**

This file describes code for P4 hosts and P4 switch.

**P4 Host** is a class in this file which inherits the Host class of mininet python API and is used to provide an outline of the host type that we are going to use. It is called while defining the type of host that we are going to use in the topology while initializing the **Mininet class**. It has commands running to disable the ipv6 using the **cmd()** command.

It also has a method called **describe()** which prints out the interface information along with MAC address and IP address.

**defaultIntf().name 🡺** prints the name of the interface which is renamed to eth0 using the **rename** method of Host class.

**defaultIntf().IP() 🡺** prints the IP address of the host default interface.

**defaultIntf().MAC() 🡺** prints the MAC address of the host interface.

**P4 Switch** is class which gives outline of the P4 switch which we are going to employ for our P4 programs to run. It has **init () method** with following parameters:

1. **Name:** Name of the switch to be identified.
2. **Sw\_path:** Path to the executable binary file of Bmv2 target switch.
3. **Json\_path:** Path to the compiled P4 program in json format.
4. **Thrift\_port:** Port number through which the switch will accept configuration commands.
5. **Pcap\_dump:** Path to the directory where the packet sniffing files will be dumped.
6. **Log\_console:** If this flag is true then it prints the log messages to the console.
7. **Log\_file:** Path to the log directory where all the log message files will be stored.
8. **Device\_id:** device on which the switch will be executed.
9. **Enable\_Debugger:** enables debugger if true.
10. **\*\*Kwargs:** Any extra keyword arguments that are passed apart from the above mentioned are stored in this parameter.

All the above parameters are stored in the class’s own instance variable for further use with all the error checks necessary along with appropriate error messages.

**Check\_switch\_started(self, pid):** process id is as the switch is executed as an independent. While the process is running (pid exists), we check if the thrift server has been started. If the thrift server is ready, we assume that the switch was started successfully. This is only reliable if the thrift server is started at the end of the init process.

**start(self, controllers):** starting a p4 switch with the following args:

1. **-i:** This tag followed by port number followed by ‘@’ concatenated with interface name initializes the interface of the switch.
2. **-pcap:** This tag followed by the path to the pcap dump folder instructs the switch where to dump the packet sniffing files.
3. **–thrift-port:** This tag followed by port number will initialize the thrift port for this switch.
4. **–nanolog:** No idea about this tag.
5. **–device-id:** device id for the switch.
6. **–debugger:** This tag confirms that debugger should be kept on.
7. **–log-console:** This tag affirms that the messages can be displayed onto the console.

After this the P4 switch is started using this command:

elf.cmd(' '.join(args) + ' >' + self.log\_file + ' 2>&1 & echo $! >> ' + f.name)

This will start the P4 switch and will return a pid of the process where this switch is started as a new process.

**stop():** This method terminates the P4 switch by killing the process which is running the switch using the **cmd()** method of mininet python API.

**Analysis of 1swdemo.py:**

It is a single switch topology mininet environment with different mininet API used taking the configuration from the command prompt in the form of various tags:

**1)behavioral-exe(--behavioral-exe):** This tag gives the address to the binary executable of the target bmv2 switch.

**2)thrift port (--thrift-port):** Used for specifying thrift server port for table updates.

**3) number of hosts (--num-hosts):** number of hosts to be connected to switch

**4) mode (--mode):** not sure about what this means

**5) Json file path (--json):** specifies the json path which is to be executed on the switch

**6) pcap dump (--pcap-dump):** dump packets on interfaces to pcap files

Using the above arguments in terminal we can configure our environment.

**P4RuntimeSwitch.py Script Analysis:**

This is a BMv2 switch with gRPC support. The parameters are the same as the P4 switch class explained above. The only addition is grpc\_port parameter which takes in the grpc port number.

The command line arguments are also the same except there is one more argument called:

**--grpc-server-addr 0.0.0.0** concatenated with the grpc port number.

**Analysis of the utils directory programs from Tutorials Repository:**

**1)run\_exercise.py:** This program helps build the bmv2 mininet environment with all the files provided as arguments in the command line. Command line arguments are:

1. **–quite:** helps suppress the log messages.
2. **–t:** Path to the topology file
3. **-l:** mentioning the log directory where all the logging files will be stored.
4. **-p:** mention the pcap directory where all the packet sniffing files will be stored.
5. **-j:** specify the json file of the compiled P4 program that is to be executed on the switch.
6. **-b:** Path to the behavioral executable file

**Get args():** This function extracts all the arguments along with tags and passes it onto the exercise runner class which initializes the complete p4 mininet environment.

**ConfigureP4Switch Method:**

This class configures a P4 switch according to the switch executable type mentioned in the argument. The purpose is to ensure each switch’s thrift port is unique.

It has 2 different subclasses of switches one is of type which employs grpc for communication and the other one employs a simple P4 switch.

**Exercise Topo Class:**

This is the mininet topology class for the P4 tutorial exercises which is configured using the topology.json file provided.

**\_\_init\_\_():** This method takes input the path to executable switch file along with dictionaries of switch and hosts which is parsed from json file and stored in dictionary.

Later in the code it is looped through to find the number of hosts and links are established between the hosts and switches as stored in the dictionary. Mininet python API method **addLink()** is used to establish virtual links in the mininet environment between the hosts and the switches present. Same is used for establishing links between the switches. Also to initialize hosts we use **addHost ()** method and for switches we use **addSwitch ()** method for simple switches or custom made switches written in python can be used our application per se.

There is a helper **parseLink ()** function which is used to identify the linking using string functions to parse the link represented in json array format.

**Class Exercise Runner:**

This class is the essence of how bmv2 and mininet are executed hand in hand. It takes input the topology in json format along with compiled p4 program in the form of json as well. Furthermore, it also take the path of log file and pcap dump file.

The **init()** method parses the json file and initializes appropriate dictionaries for further use by the topology generator method.

**run\_exercise()** method is used to actually run the mininet bmv2 environment creating the network, initializing the hosts and running commands that will populate the tables in hosts and switches which will be used for packet forwarding. **start()** is a sub method which is a mininet API method which actually starts the topology. As mentioned earlier that hosts are to be programmed using the commands that are mentioned in the json file. Same goes with the json file of switches.

**Parse\_Links()** method takes input the description of the form [node1, node2, latency, bandwidth] where latency and bandwidth being optional, parses these descriptions into dictionaries and store them as an array of dictionaries with the above mentioned fields.

**create\_network():** This method creates the mininet network object and store it as self.net. Here mininet topology instance stored as self.topo. Mininet instance stored as self.net

**program\_switch\_p4runtime(self, sw\_name, sw\_dict):**

The parameters are the object of the class, the name of the switch that is to be configured along with all the commands that are being parsed as dictionary from the topology json file. Then the json file is passed along with configuration details such as address of the switch along with port number, device\_id, json config file, working directory, path to the proto\_dump file.

This function is for those switches that has a json file giving all the important configuration details.

**program\_switch\_cli(self, sw\_name, sw\_dict):**

This method will start up the CLI and use the contents of the command files as input.

The switch here will run as a subprocess with a thrift port for command communication.

**program\_switches(self):**

This method will program each switch using the BMv2 CLI and or P4runtime, depending if any command or runtime JSON files were provided for the switches. It will search through the switches dictionary for appropriate fields and decide on the type of switch function to be called.

**program\_hosts(self):**

execute any command provided in the topology.json file on each mininet host.

We know that the json file is type of a collection of dictionaries and in that each host is sub dictionary with key as the host name and command array as the value of the dictionary.

We will loop through this host collection which is stored in a variable called hosts. We will get the host object from the mininet topology and then run through its commands executing it using the mininet python API function **cmd()** which takes the command that is to be executed as input.

**do\_net\_cli(self):**

Starts up the mininet CLI and prints some helpful output. It assumes a mininet instance is stored as self.net and self.net.start() has been called which is a method that will start the mininet topology.

**get\_args():**

This method parses all the command line arguments that are present when this python script is called. The tags are as follows:

1. -q: This tag asserts that the log messages are to suppressed.
2. -t: The file path followed by this tag points to the json file that meticulously elicits the topology that is to be initialized by the mininet.
3. -l: Specifies the path to the log directory where all the log messages are to be stored for further reference.
4. -p: Specifies path to the dump directory where all the sniffing files are to be stored.
5. -j: Specifies path to the compiled p4 program which is also in the form of json.
6. -b: Specifies the binary executable file path of the BMv2 target switch.

First the get\_args() function is called which will extract all the necessary information from the command line. Following it will be the instantiation of Exercise Runner class which will help build the topology and initialize the BMv2 switch with tables and actions that are to be performed when a packet is encountered which will be governed by the P4 program. Following which will be the call to run\_exercise() method which do all the magic of mininet environment and will establish a test environment in mininet.

**FINAL PROJECT**

**Domain:** Programmable software defined Networks

**Project Aim:** Investigate what kind of local network information on the switch can be obtained using P4.

**PROJECT IMPLEMENTATION IDEA FOR COUNTER**

* Counter is a stateful object of P4 which essentially counts the number of packets that the switch encounters.
* At first it was suggested to use UDP server to receive the network information, but the road hit a rock when writing a python client inside a BmV2 switch was a dead end.
* Instead, a workaround python script is devised which will send a packet like **probe packet** which contains the required headers in which this information will be stored by the switch on encountering this packet.
* For the switch to perform this function we will write a P4 code with required parser for our probe packet, a match-action pipeline, counter, bloom-filter or register array.
* Match-action pipeline will contain a single function called **tally** which will increment the counter associated to the match key which is **ingress port.**
* Simultaneously, we will compute a hash on **ingress port** using **hash** function provided by v1model library. This hash will serve as key to bloom-filter which will store the value of the number of packets encountered by that port by incrementing the value stored in bloom-filter on every table hit **count\_table** which manages **tally** function.
* After incrementing the count, the result is read from the filter and is put into header called **n\_packets** which has a 16-bit field called **num\_packets** where the value will be stored and will be directed towards the same port through which the probe packet came initially.
* As elicited earlier that a python script will be sending the packet, that same python script will receive the packet sent by the switch after putting in the required data and will be displayed.

**PROJECT IMPLEMENTATION IDEA FOR QUEUE DEPTH**

* Queue depth is pivotal in case of congestion control as the number it is holding determines the packet sending rate of the sender to avoid network congestion.
* This value can be accessed through P4 using struct **standard\_metadata\_t** which stores this value in the field **enq\_depth.**
* Same idea as counter was thought to be implemented but the issue faced was the number of bits allotted to **enq\_depth** is 19 which is an odd number and so it is non-viable for header accepted by BMV2 switch.
* So, it became difficult to implement the idea. Hence, it could not be incorporated into the header.

**PROJECT IMPLEMENTATION IDEA FOR METER**

* Meters are stateful objects that measure the data rate, either in packets or bytes per second, and output the result as one of three colours: red, yellow, or green, which are encoded as a 2-bit-wide field.
* For our implementation we will modify the probe packet header which we used for counter and add a new field called **meter\_val** and a new table **m\_read** in P4 code.
* The key to m\_read table is source mac address as we will mark the sender with the above-mentioned colour coding using the **extern meter object** provided by v1model.p4 library which keeps track of number of bytes each sender is sending using the meter and marks sender as **0(green), 1(yellow), 2(red).**
* Each table entry will be associated with independent meters just like counters.
* Whenever a hit occurs the meter value of the source is read and is stored in meta data with variable name **meter\_tag.** This value will subsequently be stored in **meter\_val** of the header **n\_packets.**
* We will run the same python script with modified header and read the value **meter\_val** along with **num\_packets** and display both values.
* Also, we have instructed switch to drop packets of sources whose meter value is 1 or 2 meaning those who have moderate and high traffic rate respectively.

**P4 CODE**

|  |
| --- |
| /\* -\*- P4\_16 -\*- \*/  #include <core.p4>  #include <v1model.p4>  #define BLOOM\_FILTER\_ENTRIES 4096  #define BLOOM\_FILTER\_BIT\_WIDTH 16  const bit<16> TYPE\_IPV4 = 0x800;  const bit<16> TYPE\_S = 0x1234;  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* H E A D E R S \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  typedef bit<9> egressSpec\_t;  typedef bit<48> macAddr\_t;  typedef bit<32> ip4Addr\_t;  header ethernet\_t {  macAddr\_t dstAddr;  macAddr\_t srcAddr;  bit<16> etherType;  }  header ipv4\_t {  bit<4> version;  bit<4> ihl;  bit<8> diffserv;  bit<16> totalLen;  bit<16> identification;  bit<3> flags;  bit<13> fragOffset;  bit<8> ttl;  bit<8> protocol;  bit<16> hdrChecksum;  ip4Addr\_t srcAddr;  ip4Addr\_t dstAddr;  }  header pack\_tot{  bit<16> num\_packets;  bit<32> meter\_val;  }  struct metadata {  /\* empty \*/  bit<32> meter\_tag;  }  struct headers {  ethernet\_t ethernet;  pack\_tot pkts;  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  parser MyParser(packet\_in packet,  out headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  state start {  transition parse\_ethernet;  }  state parse\_ethernet {  packet.extract(hdr.ethernet);  transition select(hdr.ethernet.etherType) {  TYPE\_S: n\_pack;  default: accept;  }  }  state n\_pack{  packet.extract(hdr.pkts);  transition accept;  }  }  \*\*\*\*\*\*\*\*\*\*\*\* C H E C K S U M V E R I F I C A T I O N \*\*\*\*\*\*\*\*\*\*\*\*\*  control MyVerifyChecksum(inout headers hdr, inout metadata meta) {  apply { }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\* I N G R E S S P R O C E S S I N G \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyIngress(inout headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {    direct\_counter(CounterType.packets) direct\_port\_counter;  direct\_meter<bit<32>>(MeterType.packets) my\_meter;  register<bit<BLOOM\_FILTER\_BIT\_WIDTH>>(BLOOM\_FILTER\_ENTRIES) bloom\_filter\_1;  bit<32> position;  bit<16> result;  bit<19> res\_queue\_len;  action compute\_hashes(){  //Get register position  hash(position, HashAlgorithm.crc16, (bit<32>)0, {standard\_metadata.ingress\_port},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  }  action compute\_hashes\_custom(bit<9> port){  //Get register position  hash(position, HashAlgorithm.crc32, (bit<32>)0, {port},  (bit<32>)BLOOM\_FILTER\_ENTRIES);  //bloom\_filter\_1.write(position, 0);  }  action m\_action(){  my\_meter.read(meta.meter\_tag);  }    action drop() {  mark\_to\_drop(standard\_metadata);  }  action send\_back(){  bit<48> temp = hdr.ethernet.srcAddr;  bit<48> srcAddr = hdr.ethernet.dstAddr;  hdr.ethernet.dstAddr = temp;  standard\_metadata.egress\_spec = standard\_metadata.ingress\_port;  }    action tally(){  direct\_port\_counter.count();  }  table m\_read {  key = {  hdr.ethernet.srcAddr: exact;  }  actions = {  m\_action;  NoAction;  }  default\_action = NoAction;  meters = my\_meter;  size = 16384;  }  table count\_table {  key = {  standard\_metadata.ingress\_port: exact;  }  actions = {  tally;  NoAction;  }  default\_action = NoAction;  counters = direct\_port\_counter;  size = 512;  }    apply {  if(count\_table.apply().hit){  compute\_hashes\_custom(standard\_metadata.ingress\_port);  bloom\_filter\_1.read(result, position);  if(result >= 0){  bloom\_filter\_1.write(position, (result + 1));  }else{  bloom\_filter\_1.write(position, 1);  }  bloom\_filter\_1.read(result, position);  hdr.pkts.num\_packets = result;  if(m\_read.apply().hit){  hdr.pkts.meter\_val = meta.meter\_tag;  }else{  hdr.pkts.meter\_val = 4;  }  send\_back();  }  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* E G R E S S P R O C E S S I N G \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyEgress(inout headers hdr,  inout metadata meta,  inout standard\_metadata\_t standard\_metadata) {  apply {  //hdr.pkts.queue\_len = standard\_metadata.enq\_qdepth;  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\* C H E C K S U M C O M P U T A T I O N \*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyComputeChecksum(inout headers hdr, inout metadata meta) {  apply {  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* D E P A R S E R \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  control MyDeparser(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.pkts);  }  }  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* S W I T C H \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  V1Switch(  MyParser(),  MyVerifyChecksum(),  MyIngress(),  MyEgress(),  MyComputeChecksum(),  MyDeparser()  ) main;  The python script that is sending packets to the switch  #!/usr/bin/env python  import argparse  import sys  import socket  import random  import struct  from scapy.all import sendp, send, get\_if\_list, get\_if\_hwaddr,srp1  from scapy.all import Packet  from scapy.all import Ether, IP, UDP, TCP  from scapy.fields import IntField  from scapy.packet import bind\_layers  class Num\_Packet(Packet):  name = "Num\_Packet"  fields\_desc = [IntField("num\_packets",0), IntField("meter\_val", 0)]  bind\_layers(Ether, Num\_Packet, type=0x1234)  def get\_if():  ifs=get\_if\_list()  iface=None # "h1-eth0"  for i in get\_if\_list():  if "eth0" in i:  iface=i  break  if not iface:  print "Cannot find eth0 interface"  exit(1)  return iface  def main():  #addr = socket.gethostbyname(sys.argv[1])  iface = get\_if()  #print "sending on interface %s to %s" % (iface, str(addr))  pkt = Ether(src=get\_if\_hwaddr(iface), dst='00:04:00:00:00:00')  pkt = pkt /Num\_Packet(num\_packets = 0, meter\_val = 0)  pkt = pkt/' '  pkt.show2()  resp = srp1(pkt, iface=iface, timeout = 1, verbose=False)  resp.show2()  if resp:  num = resp[Num\_Packet]  if num:  print "number of packets till now",num.num\_packets>>16  print "the meter value is:", num.meter\_val>>32  else:  print "cannot find Num\_Packet header in the packet"  else:  print "Didn't receive response"  if \_\_name\_\_ == '\_\_main\_\_':  main() |

**OUTPUT IMAGES:**

1. Python script running on Host1 connected to port1 will send the packet and get the counter and information for port 1.

Text

Description automatically generated

**To Verify we will run the BMV2 runtime client to extract the value of counte**Text

Description automatically generated**r**

1. We will be running similar script on host 2 to get data for port 2

Text

Description automatically generated

**To verify our result we will look for counter value of port 2 in runtime-Client**

**Text

Description automatically generated**

**The above image is that of runtime Client**

1. We will be running similar script on host 3 to get data for port 3

Text

Description automatically generated

**To verify our results we will check counter value for port 3 in runtime-client**

Text

Description automatically generated

1. We will run a similar script on host 4 to get information for port 4

Text

Description automatically generated

**To verify our results we will check counter value for port 3 in runtime-client**Text

Description automatically generated

**COMMANDS USED TO RUN THIS PROJECT**

* **make run:** This command will initialize our topology and mininet environment.
* **./send.py:** This command is used to run our python script which will send and receive our custom packet to retrieve counter and meter values.