

Towards accurate simulations of programmable dataplanes

Master's Thesis

Andreas Pantelopoulos
Department of Computer Science

Advisors: Edgar Costa Molero, Maria Apostolaki
Supervisor: Prof. Dr. Laurent Vanbever

August 14, 2017

Abstract

During the last decade computer networks have become more configurable and programmable than ever. The Software Defined Networks trend led this transformation, and now P4, which is a language for programming network forwarding elements. P4 programs are compiled and loaded on P4 targets, which can be implemented in software or hardware. The most popular P4 software target is the behavioral model (bmv2) switch, which is written in C++. Researchers have used bmv2 to implement complex P4 programs and test new ideas. Usually it is coupled with the Mininet framework in order for virtual networks to be created. Even though bmv2 can run any P4 program, it lacks in performance. Further, emulation of huge virtual topologies or usage of real networks is not feasible. Thus, researchers are forced to evaluate ideas in packet level simulators. However, these simulators do not yet support P4 programs and great effort is needed in implementing the P4 program's behavior. Furthermore, network simulators, like ns-3 and OMNET++, are large software projects which are difficult to use, understand, and extend.

In this thesis, we ported the behavioral model switch to ns-3 network simulator with the use of the Direct Code Execution (DCE) framework. DCE gives programmers the ability to run unmodified real-world applications within the context of ns-3 simulations. By using this capability, we managed to run the bmv2 application as a network node within ns-3 and implemented complex simulation scripts. We further proved that the port is sound by executing the same P4 program in Mininet and the ns-3 simulator, while observing the same trend in the results. We argue that our contribution is important mainly for two reasons. First, existing P4 programs can be used within ns-3 simulations with little effort. Second, results from recent research papers that use P4 can be easily reproduced and benchmarked against.

Acknowledgements

I would like to thank my supervisor, Prof. Laurent Vanbever, for giving me the opportunity to work on such an interesting and exciting topic. Further, I would like to thank my advisors, Edgar Costa Molero and Maria Apostolaki for the constant support throughout my thesis. Lastly, I would like to thank my colleague and friend Antonios Karkatsoulis for providing help with the DCE framework and also for our meaningful discussions during my thesis.

Contents

1 Introduction	1
1.1 Motivation	2
1.2 Approaching the task	3
1.3 Thesis contribution	3
1.4 Thesis structure	4
2 Related Work	5
2.1 Real world applications in ns3	5
2.1.1 Ported Applications	6
2.2 P4 research	7
3 System Description	9
3.1 The ns-3 discrete event network simulator	9
3.1.1 Overview	9
3.1.2 Important concepts	10
3.1.3 Extensions to the codebase	11
3.2 The Direct Code Execution framework	14
3.2.1 Benefits	14
3.2.2 How it works	14
3.2.3 Important components	16
3.3 The behavioral model and Mininet	17
3.3.1 The P4 language	17
3.3.2 bmv2 Implementation details	20
3.3.3 Mininet	22
4 Porting bmv2 to ns-3	23
4.1 Port challenges	23
4.1.1 Porting the thread library	23
4.1.2 Porting the libpcap library	24
4.1.3 Try-catch constructs in DCE	24
4.1.4 __cxa_thread_atexit_impl function	25

4.1.5	Interface monitoring	25
4.1.6	Problem with bmv2 execution	25
4.1.7	Populating Match-Action tables	25
4.1.8	Modifying CsmNetDevice	26
4.2	Unsolved issues and limitations	26
4.3	Simulated topologies	28
4.3.1	Sample Topologies	28
4.3.2	Fat-Tree	28
5	Experimental results	31
5.1	Mininet scalability issues	31
5.2	Port accuracy	34
6	Conclusions and Future work	37
A	Instructions for porting bmv2 to ns-3 for an Ubuntu VM	39
A.1	ns3 requirements	39
A.2	Install clang and clang++	40
A.3	Install cmake 3.4 from source	40
A.4	Download and build libcxx and libcxxabi from llvm	40
A.5	Download, patch and build libpcap from source	41
A.6	Download and build boost with clang++ and libc++	41
A.7	Configure LD_LIBRARY_PATH	42
A.8	Download and build thrift server	42
A.9	Download, patch and build behavioral model and custom thrift clients	43
A.10	ns3 build and install	44
A.11	Build latest Linux Kernel for use	44
A.12	Configure and build DCE	45
A.13	Download and build the ip program	45
A.14	Download and build the iperf program	46

List of Figures

3.1 Fat-Tree topology with $k = 4$	12
3.2 DCE architecture	15
3.3 P4 abstract model	18
3.4 P4 programming workflow.	19
3.5 Bmv2 workflow	20
3.6 Architecture of simple_switch target	21
4.1 Example topologies implemented in DCE.	29
5.1 Bandwidth results for linear topology.	32
5.2 Execution time for ns-3 simulator.	33
5.3 Flowlets vs Ecmp executed in Mininet.	35
5.4 Flowlets vs Ecmp executed in ns-3.	35

LIST OF FIGURES

Chapter 1

Introduction

Network operators nowadays experience unprecedented traffic demands, that increase every year. In order to keep up, they need to operate flexible networks, which are able to adapt to varying traffic and also support new services as fast as possible. They need a network that is easily configurable and programmable, one that can change its behavior if needed, and is not tied to specific protocols or technology. Unfortunately, networks usually consist of devices that are closed, proprietary black boxes, which provide limited interfaces and programmability.

This need of programmability and control over the network has been heavily addressed by the research community over the years. It can be traced back in time with early efforts, like active networks [37], and more recently with OpenFlow [29], which introduced a new way to build and manage networks, by separating the control and data plane. However, OpenFlow enabled devices were still tied to specific header formats and did not support arbitrary byte matching on incoming packets, which resulted in limited functionality, as well as an ever extending OpenFlow specification in order to support more protocols.

To address these issues, the P4 high level language was created [22], which enables the programming of protocol independent packet datapaths. P4 targets are forwarding elements, that are not tied to specific protocols and forwarding behavior, but are programmed by the operator with P4 programs. Furthermore, these programs are independent of the target, meaning that the same program can be compiled and loaded to different devices which implement the P4 specification. While there are no hardware targets widely available in the market, there exist several software ones. The most popular and widely used is the *behavioral model* [1], also known as *bmv2*, an open source

software switch written in C++, which implements the full P4 specification [30].

P4 adoption by network operators might provide solutions to the problems mentioned before. Furthermore, datapath programmability enabled researchers to implement new ideas, like universal monitoring [27] or tcp problem diagnosis [24]. Researchers usually prototype in bmv2 and then implement their ideas in a network simulator in order to test the performance of their proposed solutions. Re-implementing the datapath behavior in a simulator is potentially problematic; extra effort is needed to implement the P4 specification in the simulator and also the implementation might have bugs or be inaccurate. Ideally, researchers should prototype in bmv2 and experiment with the same P4 program in a network simulator.

1.1 Motivation

When we started working for our thesis, the scope was different. The original goal was to implement load balancing techniques with P4 programs for Fat-Tree [20] topologies. At start, we decided on the algorithm we wanted to try and we prototyped it in bmv2. Since there exists *Mininet* [26] support for bmv2 switches, we implemented the Fat-Tree topology on it and implemented in P4 our algorithm as well as the Equal Cost Multipath (ecmp) algorithm to use as a baseline.

However, it soon became clear that bmv2 and Mininet are not the right tools to emulate and evaluate load balancing techniques for a data center. The main problem was the overall scalability of the bmv2 application. It consists of several parallel threads, which lead to a huge system overhead when trying to implement large interconnected topologies. Furthermore, datacenter traffic consists of mostly small, short lived, latency dependent flows [21] which were impossible to be reproduced accurately in Mininet.

The solution to the above problems was to use a network simulator. After researching the simulator space we decided to implement our load balancing algorithms in the *ns-3* simulator, mainly because it is well documented and widely used within the research community. *Ns-3* has a big learning curve and also the implementation phase was strenuous and difficult. Surprisingly, there does not exist a straightforward manner in which someone can implement a switch in the simulator with custom forwarding behavior.

During the implementation process, it became clear how extremely useful would be to use the P4 program which was prototyped in bmv2 in the context of the simulator, without having to implement the exact forwarding behavior, or, at least, with minimal

effort. Ideally, we would like to run the original bmv2 application in the simulator as a network node. This way, not only we would not invest extra time to re-prototype our ideas, but also be certain that the implementation is bug free and adheres completely to the P4 specification. At this point, the scope of our thesis shifted *from* developing load balancing techniques for data centers, *to* porting the behavioral model application to the ns-3 simulator.

1.2 Approaching the task

There exist various ways of approaching the port of bmv2 to the ns-3 simulator. The most obvious way is to re-implement the bmv2 application inside the simulator. However, this is nearly impossible. This is due to the fact that bmv2 is very complex and refactoring the code in order to run in the simulator would take months and might not be functional in the end. Also, it would result in modifying the application code, which is undesirable, as mentioned in the previous section.

Another option would be to run the bmv2 application externally, and use inter-process communication to communicate with the simulator. This approach might work, however, the application would not run in the simulator context and thus would not perceive time as in the simulation, making time based forwarding decisions, like flowlets [38], impossible to implement.

For the above reasons, we decided to not follow these approaches. Instead, we opted to use the simulator's Direct Code Execution System (DCE) [36], which enables the usage of nearly unmodified applications and the Linux kernel code in the context of the ns-3 simulator. This way, we were able to fully port the bmv2 application, almost unmodified, in ns-3, and conduct meaningful experiments.

1.3 Thesis contribution

The contributions of the thesis are the following :

- A complete and fully functional port of the behavioral model to ns-3 with the use of DCE system. We demonstrate the port with simple examples and topologies as well as more complex ones. The validity of our implementation is proved by prototyping complex traffic engineering techniques in P4 programs for Mininet, executing the prototypes in ns-3, and observing the same trends in the results.

- A complete set of instructions that describe step-by-step the porting process, easy to follow by anyone willing to reproduce our experiments or use bmv2 in simulations.
- A set of libraries and scripts for ns-3 in C++, that implement Fat-Tree topologies, complex traffic patterns simulation and custom Layer-2 forwarding nodes.

1.4 Thesis structure

The thesis is organized as follows:

In Chapter 2 we will describe recent research papers that make use of P4 language to implement new ideas and also make use of a network simulator in their evaluation. Furthermore, we will review examples of other real-world applications that were ported successfully to the simulator. In Chapter 3 we will describe ns-3, DCE, Mininet Framework and bmv2 application. Chapter 4 will provide a detailed description of bmv2 port to ns3 and also the major problems we encountered during this process. A step-by-step guide of the port will also be provided in the Appendix. In Chapter 5 we will evaluate the port, compare it with the Mininet implementation, and also present results involving complex P4 programs in order to attest to it's correctness. Lastly, we will summarize our work and provide suggestions for future work.

Chapter 2

Related Work

In this section we will describe the background and related work of the thesis. We will first introduce the ns-3 simulator and the Direct Code Execution (DCE) framework and describe existing real world applications that have been ported to ns-3. Afterwards, we will briefly describe research papers that use P4, prototype on bmv2 and perform experimental evaluations on a network simulator. We will argue that such works can greatly benefit from our thesis contribution.

2.1 Real world applications in ns3

Ns-3 [10] is a discrete event network simulator which is widely used in the network research community. Each simulation consists of a series of events that affect the simulation state. Each event is executed until completion and may generate more events. Ns-3 is developed in C++ and has optional support for scripting in Python. More details about ns-3 concepts related to the thesis will be provided in Chapter 3. Ns-3 tries to model reality as accurately as possible, however, for certain use cases this is not always possible or requires considerable effort. Such a use case, for example, is the TCP layer of an endpoint, which is very involved due to the complexity of the TCP protocol.

For this reason the Direct Code Execution (DCE) [36] framework was developed. DCE enables the use of real world applications and the Linux Kernel, almost unmodified, within ns-3 simulations. It works by virtualizing in user space the execution of each application and redirecting important system calls, like, for example, network and filesystem calls or memory allocation, to the DCE subsystem which is responsible for handling them.

Such capability provides researchers with a great simulation tool and also comes with many benefits. First, it enhances simulations with realism, since real world applications can be used. Second, it makes experiments easier to reproduce and validate, since the same applications can be utilized in different experiments and scenarios. Lastly, due to the fact that the simulation is executed under one single process it enables easier debugging of the application at hand. We will describe DCE in greater detail in Chapter 3.

Alternatives to DCE enabled simulations exist, the most popular being Mininet [26]. Mininet uses lightweight OS virtualization in order to create and interconnect network topologies. It is a flexible tool that enables rapid prototyping and uses applications and protocol stacks without modifications. However, as we already stated in Chapter 1 and as will experimentally demonstrate in Chapter 5, this approach does not scale and cannot provide accurate results in the context of P4 prototyping. While it is ideal for creating and validating prototypes, it does not scale with the size of the network topology and increasing traffic demand.

2.1.1 Ported Applications

In this part we will briefly mention and describe existing applications that are ported to ns-3. The existence of such proofs of concept greatly encouraged us during our work :

- **The Linux Kernel** : The network stack of Linux Kernel can be used in ns3. This alleviates the need of building custom network stacks for host endpoints and also enables more realistic simulations. The project responsible for this work is net-next-nuse [12].
- **Quagga** : Quagga [18] is a routing software suite that provides implementations for various routing protocols. It has been successfully ported to ns-3 and used in research papers.
- **Open vSwitch (OVS)** : OVS [14] is a software switch operating in user and kernel linux space. It has been ported to DCE and demonstrated under a software defined wireless network scenario [28].
- **Network Controllers** : Various Software Defined Network (SDN) controllers have been successfully ported and used in ns-3 simulations under DCE [32], [17], [28].
- **Common Applications** : A variety applications have been ported to ns3. Some of them are iperf, ping, httpd and CCNx, which implements a Content-Centric Networking architecture.

2.2 P4 research

P4 [22] is a high level language for programming the behavior and functionality of protocol independent network devices. Unlike the OpenFlow [29] protocol, it does not dictate the use of specific header fields when parsing and matching incoming packets, but rather supports arbitrary parsers and byte level matching. It also supports custom match-tables, specified by the device programmer, with custom pattern matching, rules and table pipelines. These capabilities enable engineers and operators to program networks in a way that best serves their needs and traffic.

P4 programs are high level constructs, compiled and loaded on P4 *targets*, which need to adhere and implement the P4 specification [30]. P4 targets exist in hardware [19] and software [1], [33], with the most popular and widely used being the *behavioral model* (bmv2), which is a software switch that fully implements the P4 specification. In Chapter 3 we will describe bmv2 in greater detail.

The advance of P4 software switches enabled network researchers to experiment with new ideas and implement complex P4 prototypes. Recent papers make use of P4 for monitoring the network [27], monitoring and diagnosing TCP performance [24] or monitoring network flows with minimal overhead [34]. Further, P4 implementations were used to load balance data centers with global congestion awareness [25] or for precise detection of large flows [35]. Most of P4 research papers use bmv2 to prototype the ideas involved, and then evaluate the performance of the proposed solution by simulating complex scenarios in a network simulator.

Using a network simulator for evaluating P4 solutions comes with two disadvantages. First, the P4 program's behavior needs to be re-implemented in the simulator at hand. As mentioned in Chapter 1, our own experience showed that this is a difficult and error prone procedure and also that extra care is needed in order for the P4 functionality to be captured accurately. Second, in most research papers that use P4 programs, the authors usually publish the P4 code that is used in the paper but not the simulator code that evaluated their solution. This makes it difficult for the results to be validated, reproduced or benchmarked against.

The contribution of this thesis solves both the aforementioned problems. By porting bmv2 to ns-3 and executing arbitrary P4 programs, we are completely certain that we capture correctly the functionality and behavior of a P4 program. Also, minimal effort is needed in order to create complex topologies and simulation scenarios involving P4 network switches. Lastly, in order to reproduce previous research ideas and benchmark against them, we only need the P4 program at hand, which is usually also published or can be easily found.

Chapter 3

System Description

In this Chapter we will describe all the relevant components that were used in our thesis and also introduce the theory that is needed in order for someone to understand our experiments and deliverables. We will start by presenting the ns-3 simulator, describing important concepts that appear in every simulation and our extensions to the code. We will then continue by introducing the DCE system and detailing how it works. Lastly, we will introduce the behavioral model switch, the Mininet framework, and shortly describe the P4 language.

3.1 The ns-3 discrete event network simulator

3.1.1 Overview

Ns-3 is a discrete event network simulator. Each simulation consists of the execution of a series of events, where the execution of each individual event might create and schedule more [11]. The simulation ends when all events have been executed or the simulation's total time has expired. Thus, the main task of the simulator is to execute all events until completion in the correct order. The execution of events takes place in a discrete manner, meaning, for example, that if two consecutive events are scheduled with a time difference, without any other events existing between them, upon completion of the first event the simulator will instantly jump to the second event execution, advancing also the simulation *virtual time*. A call to *Simulator::Run* starts the simulation, and is usually the last thing called by the user, after everything is set up. This call executes the first event that is placed in the simulator event queue.

Ns-3 is written in C++ and it also supports Python bindings. This means that the core components and modules are written in C++, while simulations can be written in both C++ and Python. Ns-3 is mainly used for educational and research purposes and is perfectly suited for studying network or protocol performance under a reproducible and scalable environment. We need to clarify here that by *scalable* we refer to the ability to create scenarios that contain a large number of network resources and not the execution time of the simulator. There exist a big variety of modules in ns-3 that implement functionality ranging from wireless and vehicular communications to data center modeling.

3.1.2 Important concepts

We will present here basic concepts and abstractions [9] that ns-3 uses in order to build and execute simulations. Understanding these components is crucial, because they appear in every ns-3 script. We will give emphasis on components that were modified, enhanced or used during the course of our thesis.

Nodes

Node is the basic computing abstraction of ns-3. Usually most simulations create topologies by interconnecting Node objects together. Node objects are very rudimentary, but can be enhanced by aggregating extra functionality and resources like NetDevice, Stack and Application objects. While this design requires extra effort from the developer when creating a simulation, it resulted in a modular Node model that is easily extensible.

Net Devices and Channels

The `NetDevice` and `Channel` abstractions facilitate communication between pairs of Nodes. A `NetDevice` object is usually created in pair with a `Channel` Object. In real world terms, we can think of `NetDevice` Objects as the hardware interface and also the software needed for controlling a Linux network interface, while `Channel` objects represent the communication medium on which data is transmitted. Throughout the thesis we use the `CsmaChannel` which is attached to `CsmaNetDevice`. This channel models a communication subnetwork that implements a carrier sense multiple access communication medium, which adds Ethernet like functionality in our simulations.

Each type of `Channel` pairs with the same type of `NetDevice` upon creation of both of

them. `NetDevice` and `Channel` Objects are installed on sets of `Nodes` and enable communication between them. Usually the developer does not use these objects directly, rather than a set of `Helper` objects that facilitate the instantiation of them. The same applies for most of ns-3's core components.

Protocol Stack

Like real world network devices, ns-3 supports protocol stacks, which are aggregated to `Node` Objects in order to enhance their functionality. Ns-3 provides implementations for the most popular protocols, like IPv4, ARP, UDP, TCP and also enables researchers to develop and test their own protocol stacks. *Callbacks* are used in the implementation in order to develop protocol stacks. Upon the receiving of a packet from a `NetDevice`, the appropriate callback is found, if any is registered, and is called with the received packet. Furthermore, it is possible to integrate existing protocol stacks into ns-3, like the Linux Network Stack, with the use of the DCE framework. We refer the reader to [6] in order to get a detailed understanding of how multiple components are chained together.

Applications

Applications are executed on `Node` Objects and drive simulations. Application is the abstraction that represents user programs that typically run on an operating system. Starting an application is a separate event that needs to be executed by the simulator. It is important to note here that ns-3 simulator does not model the concept of an operating system or different privilege levels.

3.1.3 Extensions to the codebase

Custom Switch

In this section we will detail how a developer can create a layer-2 switch as an ns-3 `Node` and use it in meaningful simulations. As briefly mentioned in Chapter 1, creating a switch in ns-3 is not straightforward. We need a way to aggregate multiple `NetDevice` Objects to a `Node` and specify our custom code to be executed upon receiving of a packet at any device. This code should be responsible for implementing any forwarding logic and ultimately forward the `Packet` to a `NetDevice` or drop it.

In order to achieve this we needed to modify the code of `BridgeNetDevice`. This device handles multiple `CsmaNetDevice` objects that are aggregated to the same Node and performs MAC address learning on them, essentially handling each `CsmaNetDevice` as a distinct switch port. By following the code of `BridgeNetDevice` we created the module `src/custom-switch`. This module defines the classes `CustomL2Switch` and `CustomL2SwitchHelper`.

By extending these two classes in a new module and also overriding the methods `Receive` and `AddSwitchPort` a Layer-2 switch with custom forwarding behavior can be created. The developer is free to add any extra state to the devices in order to support specific use cases. By following this methodology we were able to create switches that implement the Equal Cost Multi Path algorithm for Fat-Tree topologies and use them in our simulations. We refer the reader to module `src/test-l2-switch` for an example of a switch creation.

Fat-Tree Topology

We implemented scripts that simulate the Fat-Tree topology [20]. Fat-Tree topologies are multi-tiered tree topologies where each switch consists of k distinct ports. Figure 3.1 illustrates a Fat-Tree with $k=4$. As we see, it has three distinct layers, the *Edge* layer, where servers are connected, and the *Aggregation* and *Core* layer. A unique characteristic of Fat-Tree topologies is that all devices are commodity Ethernet switches and there is no need for expensive high-performance network devices, which reduces the total cost of the topology.

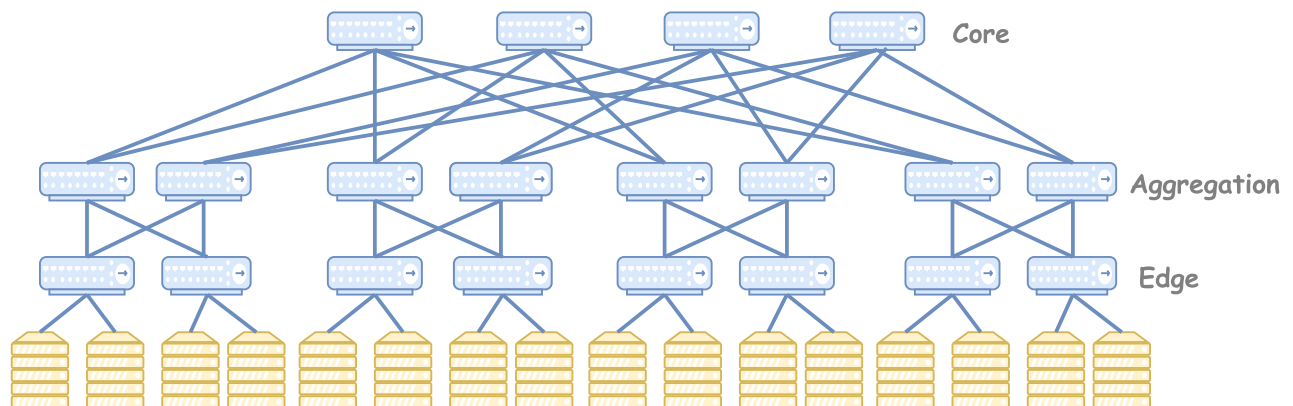


Figure 3.1: Fat-Tree topology with $k = 4$.

The aggregation and edge switches are split into k pods, where each layer in a pod consists of $k/2$ switches respectively. Each edge switch connects to $k/2$ servers and to $k/2$ aggregation layer switches. Each aggregation layer switch connects to $k/2$ core switches and $k/2$ edge switches. There exist $(k/2)^2$ core layer switches and $(k^2/2)$ aggregation and edge layer switches respectively. Fat-Tree topologies support a total of $(k^3)/4$ hosts, since each edge switch connects to $k/2$ hosts and the topology has k pods of $k/2$ edge switches. We will not explain Fat-Tree topologies in more detail since they were not a major part of our thesis. The code for Fat-Tree simulations can be found in *scratch/* directory of ns-3 simulator in our repository.

Bulk Send Application

As we mentioned before, ns-3 uses Applications in order to drive simulations. Applications generate traffic and produce results for analysis. Throughout our thesis we used a modified version of the `BulkSendApplication`. This application opens a TCP Socket with a server and sends a specified amount of data to it. We modified `BulkSendApplication` in order to record the *flow completion time (fct)* of each flow. We consider that the flow is *complete* when all the data have been transmitted and the receiver has acknowledged them. In order to achieve this, we monitor the TCP socket's buffer, where un-acknowledged data are stored and wait until this buffer is empty and the application has sent all the data to the receiver. This way, we do not take into account the teardown of the connection into the flow completion time, which was problematic, because upon heavy congestion *FIN* or *RST* packets can be lost and thus applications might never finish.

Another option for measuring flow completion times was to use the Flow Monitor [23] module, which operates by adding traces to all ns-3 devices, monitors all the packets traversing the network, and produces statistics for all the observed flows when the simulation finishes. However, this was potentially problematic, especially for large topologies with a great amount of simulated traffic, because the use of traces adds excess overhead to the simulation and would hinder ns-3's scalability. The code for `BulkSendApplication` can be found in *src/applications* under the ns-3 directory.

Traffic Simulator

In order to simulate complex traffic scenarios for Fat-Tree topologies we created the `Traffic Simulator` module which contains functions that set up traffic between ns-3 Nodes. The module supports various traffic scenarios, from stride traffic patterns to

patterns that simulate flows chosen from user-defined distributions. It is important to note here that our traffic simulator works only for Fat-Tree topologies. It's code can be found under *src/traffic-simulation* directory.

3.2 The Direct Code Execution framework

In this section we will describe the Direct Code Execution (DCE) framework for ns-3. We will start by giving a general overview of the system and then we will briefly describe how it works.

3.2.1 Benefits

The *Direct Code Execution* (DCE) [36] framework enables the use of unmodified applications and the Linux Kernel code in the context of ns-3 simulator. Such capability has numerous advantages. Firstly, it provides experimenters with advanced simulation realism since they use real world applications in the experiments. Secondly, it enables experimentation scalability, since the experiment is not bounded by machine or network resources in any way. Thirdly, since DCE operates under a single process, it can be used for easily debugging a real world application. Especially distributed applications running on multiple nodes in parallel should benefit from such capability. Fourthly, DCE enables easy and low cost replication of experiments. Since DCE uses unmodified applications someone would only need the code of the application at hand to reproduce an experiment and also benchmark against it. Lastly, DCE greatly increases the number of available applications and protocols that can be utilized in ns-3.

3.2.2 How it works

In Figure 3.2 we observe the architecture of DCE framework. We can see that there exist three distinct layers, the *Virtualization Core* layer, the *Linux Kernel* layer and the *POSIX* layer.

The Virtualization Core layer makes possible the execution of every simulated application within the same process. It is responsible for managing the heap and stack of each simulated process as well as provide memory for the kernel network stack. It uses a custom memory allocator internally to implement malloc and free functions. It also

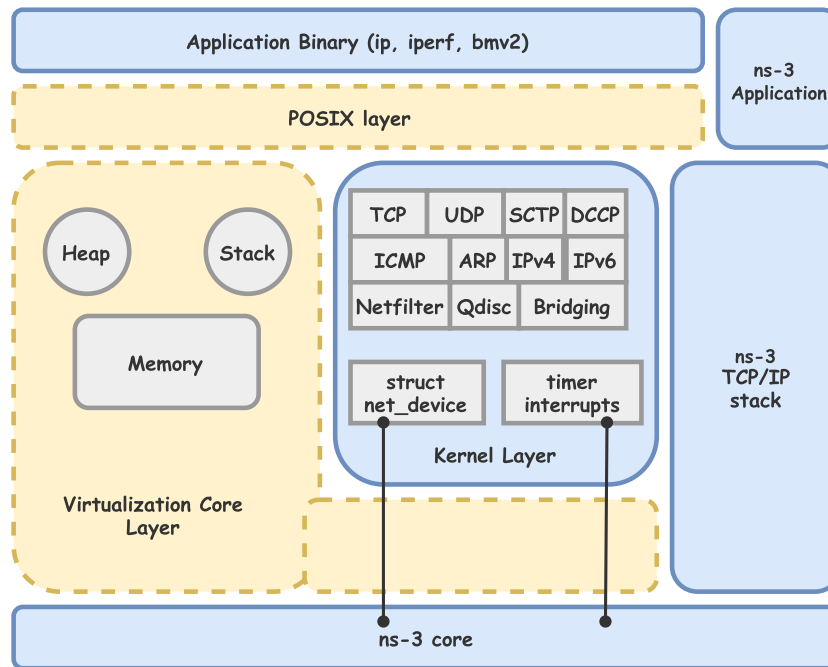


Figure 3.2: DCE architecture from [36].

uses its own Task scheduler in order to manage and schedule between threads created by simulated applications.

On top of the Virtualization Core layer sits the Kernel layer, which is responsible for implementing the functionality that is found in the Linux Kernel network stack. It uses the Virtualization Core Layer for resource allocation. At the bottom of this layer the Linux network stack is connected with the ns-3 simulator. Every interface in Linux is represented as *struct net_device*, thus this layer is responsible for creating such fake constructs and connecting them with the *NetDevice* Objects that exist in ns-3 simulations. Further, at the top of the Linux Kernel Layer, simulated applications communicate via sockets with the kernel level socket data structures.

On top of Virtualization Core and Kernel layer sits the POSIX Layer. This layer is responsible for intercepting and handling all the POSIX calls that a simulated application might invoke. Most of them can be redirected in the host operating system; calls like `strlen` or `atoi` do not need special care. However, calls that access system or kernel resources need to be intercepted and handled accordingly. The best example to illustrate such need is a call to `gettimeofday` function, which needs to return the simulation time and not the host operating system time. Further, socket related calls need to be connected

to the Kernel Layer and filesystem calls need to access a Node specific root filesystem implemented in the Virtualization Core Layer. We refer the reader to [4] for more details.

There exist some serious limitations when all the host operating system resource allocation mechanisms are bypassed, that a DCE developer needs to be aware of [7]. Firstly, the scheduler of the Virtualization Layer is not as advanced as that of Linux. An infinite loop that does not yield execution will result in DCE hanging. Secondly, not all calls of the POSIX API are currently covered, which means that porting a new application to DCE that uses such a system call will probably not function correctly. During the porting of bmv2 this issue posed a major problem, as we will describe in Chapter 4. Lastly, the Virtualization Layer has certain limitations when handling resources. The ones that we are aware of are that is a maximum number of files that can be opened during a simulation and also the stack allocated for each process might not be enough for complex executables.

3.2.3 Important components

In this section we will describe some important classes that every DCE simulation script uses.

DceManager

DCE simulated applications run on ns-3 Node Objects. An instance of DceManager needs to be instantiated on every Node that will run applications. The manager is responsible for creating virtual processes and manages their execution.

TaskManager

TaskManager is utilized by DceManager and manages the threads of virtualized processes running in DCE. It uses specialized methods like Stop, Wakeup, Sleep and Yield in order to schedule between tasks. All the POSIX calls that change the execution state are redirected to these calls via the TaskManager.

Loader

One of the most important components of DCE. It is responsible for loading the executable of the application in memory and isolating it from other executables, running on the same Node or otherwise. DCE offers two different loaders, CoojaLoader and a custom ElfLoader. The first one is more reliable but lacks performance, while the second is more efficient but is not supported in a lot of systems and is also somewhat unreliable. We refer the reader to [4] for more details, where we also found the information about

the DCE's important classes.

In order for an application to run within the DCE context it needs to be compiled and linked in a certain way. For compilation the flags `"-fPIC -U_FORTIFY_SOURCE"` are needed and for linking we need the `"-pie -rdynamic"` flags.

3.3 The behavioral model and Mininet

In this section we will describe the target application that we ported to ns-3. We will start by introducing the P4 language that is used to program behavioral model switches and then continue to describe the behavioral model itself. Lastly, we will briefly introduce the Mininet framework.

3.3.1 The P4 language

The abstract model

P4 is a high level declarative language for describing how packets are processed in datapaths, where as datapath we refer to any network element implementing forwarding functionality, like a switch, router, network function or NIC. P4 adheres to the abstract forwarding model that we observe in Figure 3.3 which describes how a packet should be processed when it is received by a datapath. All the components that the model consists of are configurable and can be changed by the P4 program. In this model, the *Parser* identifies the headers that are present in each incoming packet, which are forwarded to *Match/Action* tables that match on specific header fields and perform user defined actions.

As we see , the abstract P4 model consists of the following elements :

- **Parser** : Defines the parse graph, which is essentially the permitted byte sequences of an incoming packet. Each incoming packet is parsed and its headers are extracted.
- **Header Fields** : After the parser extracts the header of each packet it breaks it into header fields which are forwarded to further stages for matching. P4 does not impose any restrictions on the type of header fields and lets the developer define them. Matching is done only on the packet headers and not the packet payload.

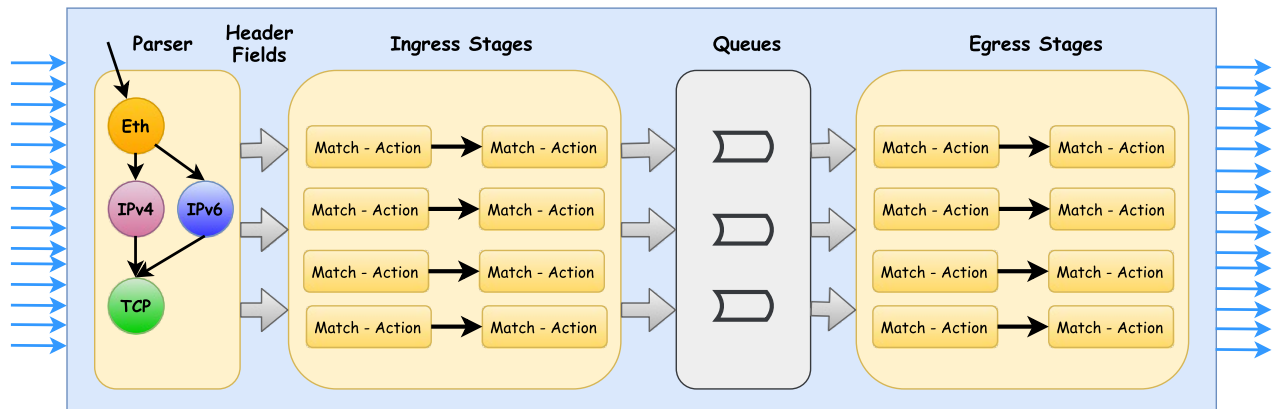


Figure 3.3: P4 abstract model from [15].

- **Match-Action Tables** : Match and Action tables contain rules that match on the header fields that were extracted from the Parser and execute user-defined actions. This model builds upon OpenFlow’s match and action abstraction, however, it also enables programmers to define both the match and action components. Actions can modify the header fields, create new headers, choose output ports or drop the packet. The configuration of the tables is completely defined by the P4 program, including the form of each table, the number of the tables and the chaining between different tables. Match-Action tables are empty when the target is booted and are populated by a target specific API at runtime. The P4 specification does not specify the API populating the tables and it is left on each network vendor to define it.

We should note that Match-Action tables exist in both *Ingress* and *Egress* Stages. Ingress Stage ultimately determines through Actions the set of output ports and number of packets for each port and then the packets are forwarded to Egress Stage where its header fields are further matched to Match-Action tables. There the packet’s headers can further be modified. Finally, the packet is forwarded to the chosen physical port(s) or dropped.

- **Queues** : Before forwarding the packet(s) to the Egress Pipeline they are placed to Queues, which handle over subscription of the output ports, however, they are not imposed by the P4 specification.

P4 language closely resembles other declarative languages, like C, but is very rudimentary and lacks many features typically found in them. For example, P4 does not yet support for loops or arbitrary function calls. We refer the reader to the P4 specification document for a complete description of the P4 high level language [30].

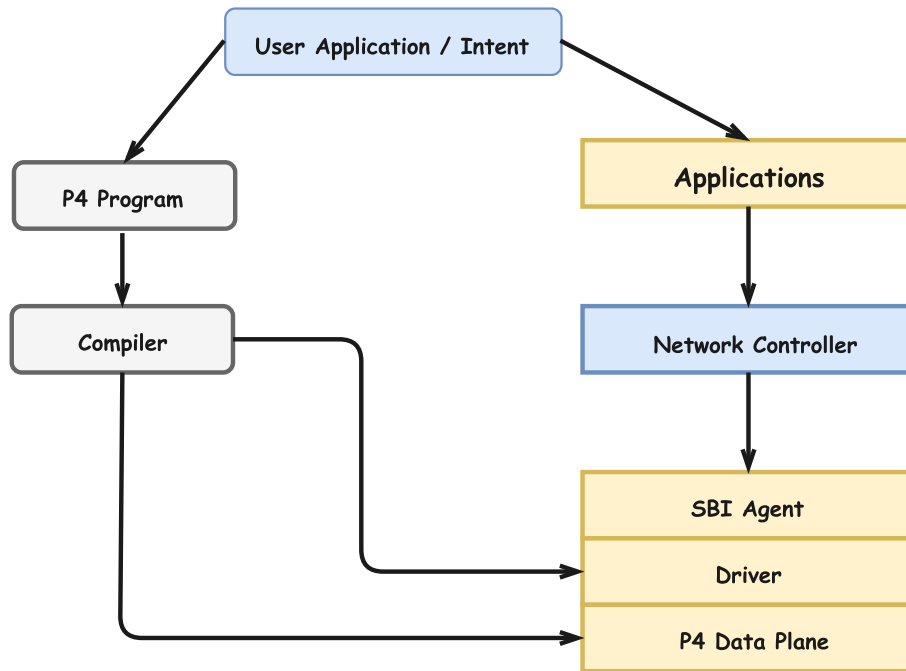


Figure 3.4: P4 programming workflow from [16].

P4 Workflow

In Figure 3.4 we observe the typical workflow that programs a P4 device. Such devices are typically called *targets*. We will use this term in order to refer to any P4 enabled device in the rest of the text.

As we see in the figure, the P4 program is first compiled to a representation suitable for the specific target and loaded into it. Here lies one of the greatest P4 advantages; the same P4 program should program different targets, as long as each target comes with a suitable compiler for it. In the behavioral model target this representation is a *JSON* file, which is loaded into the application. The compiler also usually auto-generates an API for programming the target, although this is not part of the language’s specification. Lastly, the target can be programmed by any program that understands and supports the target’s API. In the figure this program is a network controller. Specifically, for the behavioral model exists support for the ONOS operating system [13], although we did not use it in our thesis.

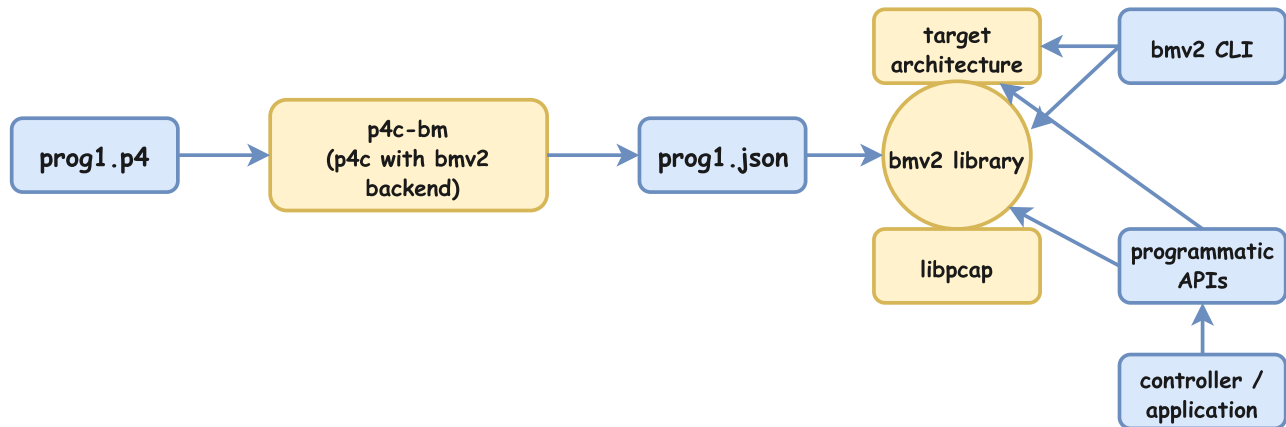


Figure 3.5: Bmv2 workflow from [2].

3.3.2 bmv2 Implementation details

In this section we will describe the *behavioral model* (bmv2) application which is the target application that we ported to ns-3. Bmv2 is a software switch developed in C++ that operates in user-space and emulates a P4 datapath. It aims on implementing the full P4 specification and being architectural independent. This means that potential P4 vendors can use bmv2's building blocks in order to implement their own software switch emulator.

Bmv2 implements three different targets: *simple_router*, *l2_switch* and *simple_switch*. The first two serve as examples of different targets that can be built with bmv2 while *simple_switch* is the standard P4 target that implements the full P4 specification and typically is used to test and showcase all P4 features. This is the target that we ported to ns-3 simulator.

Figure 3.5 illustrates how bmv2 operates. The important elements here are the following :

- **p4c-bm** : This is the compiler that translates the P4 program (prog1.p4) to a suitable representation that can be fed to the target. In the case of the bmv2 this representation is a JSON file (prog1.json). This file is specified when the target is started. Bmv2 also supports swapping of P4 programs while the target is running.
- **bmv2 library** : These are the core components of bmv2 that are used to build targets. The library includes objects that implement, for example, the parser, the headers, the match-tables, the actions and any other functionality defined in the

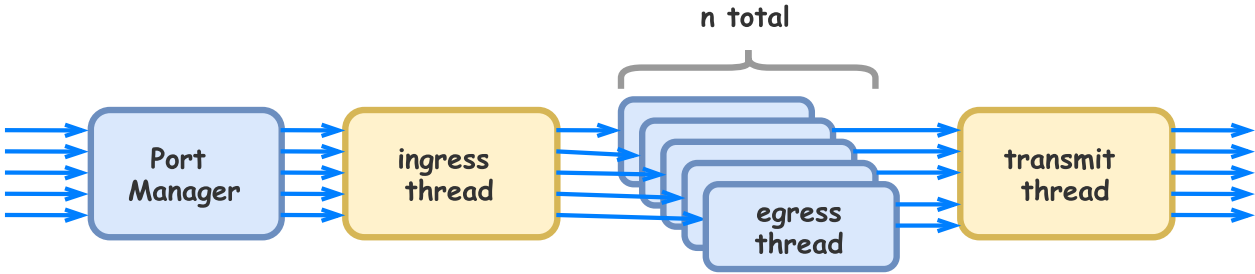


Figure 3.6: Architecture of simple_switch target

P4 specification.

- **target-architecture** : This is the target that was built with the core bmv2 library. In our case, this is the simple_switch target.
- **libpcap** : Bmv2 uses the libpcap library in order to bind on interfaces, listen for packets and transmit packets to them. Each interface specified is a port of the switch. Libpcap operates in promiscuous mode since the interfaces need to act as switch ports. We should not here that targets do not interact with libpcap calls directly rather than use the BMI interface that provides wrappers around them.
- **bmv2 CLI** : This is the client used to populate the Match-Action tables of the switch at runtime. Bmv2 runs a Thrift Server that listens at a specific port and installs rules at the switch tables. The developers provide a Python Client (*runtime_CLI.py*) for this server that also supports an interactive CLI for examining the state of the switch.
- **controller application / programmatic APIs** : Auto generated APIs that can be used by a controller application in order to populate Match-Action tables. We will not go into detail since we have not used them for our thesis.

The simple_switch target consists of different threads responsible for handling incoming packets, implementing the specified P4 ingress and egress pipelines and transmitting them through chosen output interfaces. Figure 3.6 illustrates the architecture of simple_switch. As we see, the *ingress_thread* is responsible for receiving packets from the switch ports and implementing the specified Ingress pipeline. Afterwards, it forwards the packets to one of the multiple egress threads which are responsible for implementing the Egress pipelines. Finally, packets are handled to transmit thread and forwarded to the chosen output port, if any. All thread communication is done with blocking queues.

3.3.3 Mininet

Mininet is a framework written in Python that creates virtual networks in a single host. It uses OS-level virtualization techniques, like processes and Linux namespaces in order to isolate hosts in the same topology and virtual Ethernet pairs in order to create functional links between Mininet's switches or routers. It exposes a flexible Python API that allows developers to create complex topologies and run experiments. Mininet hosts run isolated on separate namespaces and use unmodified host machine applications and the protocol stack.

Hosts are usually interconnected by OpenFlow enabled switches running in user or kernel space. In our thesis we used bmv2 switches in order to interconnect hosts, which we populated with custom P4 rules. Each node in the Mininet topologies we created is the `simple_switch` target that we described in this section.

Chapter 4

Porting bmv2 to ns-3

In this Chapter we will describe the port of bmv2 to ns-3 simulator. Firstly, we will describe in detail all the challenges that we encountered and how we managed to solve them. Secondly, we will discuss any limitations of the port and unsolved issues. Lastly, we will explain the simulation scripts that we created and use bmv2.

4.1 Port challenges

Porting bmv2 to ns-3 was a slow and strenuous process, mainly because bmv2 is a modern C++ application which uses features that DCE developers did not take into account when building the framework. In this section we will describe in detail all the challenges we encountered during this process and how we solved them. In Appendix A we provide detailed instructions for reproducing the port.

4.1.1 Porting the thread library

In Chapter 3 we described the `simple_switch` target architecture. We observed that it consists of multiple threads that interact with each other via queuing mechanisms. The `simple_switch` target uses the `std::thread` library in order to create and manage threads, which usually builds on POSIX Threads in Linux. Since DCE supports POSIX Threads, the library should work out of the box with no modifications.

However, when we first started bmv2 on DCE the application crashed unexpectedly with

SIGSEGV. After investigating the issue, we discovered that the call to `pthread_create` was not properly intercepted by the DCE framework. Since, all other calls of the Pthread library were intercepted by DCE, this resulted in the access of uninitialized resources and thus the application crashed. This problem was observed when building the application with `libstdc++`, which is the standard c++ library.

In order to solve this, we built and linked the application with `libc++` and `libc++abi`, which are alternative implementations to the standard c++ library provided by the llvm project. This way the `thread` library worked without breaking. As a result, all other projects linked with `bmv2` need to be built with `libc++ / libc++abi`. This also forced us in using `clang++` compiler for building `bmv2` and not `gcc++`.

4.1.2 Porting the libpcap library

In Chapter 3 we explained that `simple_switch` uses the `libpcap` library in order to work with Linux interfaces. Modern versions of `libpcap` use a ring-buffer that is shared with the Linux Kernel in order to avoid extra expensive copies. Mapping kernel memory to user space did not work in the context of DCE, and thus we had to disable this feature for `libpcap` to function correctly.

4.1.3 Try-catch constructs in DCE

When running the `bmv2` application on DCE we observed that it was crashing unexpectedly. After investigation, we discovered that this happened when an exception was thrown and the `catch` statement was executed in the code. The problem here was that the DCE developers have not provided coverage for the call `posix_memalign`. This call allocates aligned memory and should be intercepted by DCE framework and redirected to the Virtualization Layer.

When a `throw` statement appears in a C++ application, the compiler translates it to code that, among other things, allocates the appropriate memory for that exception. This is done with a call to `posix_memalign`. This call was not intercepted by DCE and thus memory was ultimately allocated in the host operating system rather than the Virtualization Layer of DCE. When the exception was caught, the memory allocated for it was freed with the use of `free`, a call which is intercepted and handled by DCE. This resulted in instructing the DCE Virtualization Layer to free memory that was never allocated and thus the system crashed. The solution to this was to add the `posix_memalign` call to DCE. More details about how to add a system call to DCE can be found here [3].

4.1.4 `__cxa_thread_atexit_impl` function

This problem stems from the use of `thread_local` scope which was recently introduced in C++. At the time of DCE development, this feature did not exist, and thus the developers have not provided support for it. The problem here lies at a call to `__cxa_thread_atexit_impl` which happened when a thread was exiting. Specifically, the linker could not find where this function was defined in order to link it to the binary. The solution for this was to patch `libc++abi` in a minor way, in order for this function not to be called, without of course breaking the overall functionality of the library.

4.1.5 Interface monitoring

`Simple_switch` target constantly monitors the interfaces it binds on, in order to determine if they are fully functional. This is achieved by reading system configuration files that describe these interfaces. In DCE, nodes do not have such files and thus the simulated `bmv2` can not read them. The solution to this problem was to remove monitoring of the switch ports. For the context of an `ns-3` simulation this is not a problem, because interfaces can not fail and remain constantly active.

4.1.6 Problem with `bmv2` execution

While running `bmv2` in DCE simulations the DCE execution was hanging with no apparent reason. After investigation, we discovered that this was happening when the `thread_local` storage duration specifier was used in the application. A workaround this problem was to convert all `thread_local` variables to normal variables in `bmv2` code. This solved the above problem and allowed `bmv2` to execute properly.

4.1.7 Populating Match-Action tables

In order for `bmv2` to be useful it needs to be populated with rules. As mentioned in Chapter 3, Match-Action tables are populated with the `runtime_CLI.py` tool, which is written in Python. This script connects to the Thrift server running along `bmv2` and populates the tables. Unfortunately, DCE only supports applications written in C and C++, thus, this script could not be ported to `ns-3`. A workaround for this problem was to create special `bmv2` clients written in C++, using Thrift's C++ auto-generated library,

that populate bmv2 switches with custom rules. These clients then can be ported to ns-3 and run as separate applications on the Nodes that run bmv2, once they are compiled with the correct options. For each P4 program that we run in ns-3 we needed to create the appropriate client.

4.1.8 Modifying CsmaNetDevice

We needed to modify the source code of CsmaNetDevice in ns-3 in order to simulate the behavior of switch ports. This was needed for two reasons :

- Every CsmaNetDevice in ns-3 has its own MAC address. This resulted in frames that were forwarded through this device to have as a source MAC address that of the device. This limitation would make P4 programs that use the source MAC address for matching in Match-Action tables, like l2 learning, impossible to simulate. We fixed this issue by modifying the code of CsmaNetDevice in order not to change the source address when transmitting the packet. In order for our changes to be compatible with the rest of the codebase the modifications take place only when the device is attached to a Node that runs the bmv2 application.
- When we run simulations with bmv2 we observed that packets were increasing in size when they were forwarded from Nodes functioning as bmv2 switches. It turned out that this happened when a packet was received by a CsmaNetDevice and was subsequently forwarded towards a Linux interface of a Node running the Linux Kernel. The extra bytes were added as an Ethernet trailer at the end of the frame. We solved this by modifying the CsmaNetDevice in such way that these bytes are removed upon transmission. Again, these changes are backwards compatible with the rest of the ns-3 codebase.

4.2 Unsolved issues and limitations

Although we managed to successfully port bmv2 as an ns-3 node and run accurate simulations, there exist some limitations. In the following list we summarize them and also sketch possible solutions for them.

- The behavior of a P4 target is not tied to any specific header or protocol. It can be used with any type of header, as long as it is represented in the parse graph. How-

ever, ns-3 NetDevice objects work with specific header types. For example, CsmaNetDevice expects to receive Ethernet frames and PointToPointNetDevice expects to receive point-to-point frames. Thus, using custom layer-2 headers in simulations would break the functionality of these devices, because they would not be able understand them. For this reason, simulations using bmv2 in ns-3 are restricted to packet formats that ns-3 can parse and understand. A possible solution for this is to build a NetDevice that would accept and transmit packets without parsing and decoding their headers. The device would only utilize a Queue for storing the packets and a Channel to transmit them.

- `thread_local` storage duration specifier is used for variables, for which every thread that uses them maintains a copy. This specifier is widely used in bmv2, and as we mentioned in the previous section, it does not work within the DCE framework. Unfortunately, we could not find out why the specifier does not work, thus, we resorted into removing it from wherever it was used in the source code. Fortunately, the specifier was used only for performance reasons by bmv2, thus, removing it did not alter the functionality of the application. However, since bmv2 is a work in progress, the specifier might be further used in the future. A better solution would be to discover why `thread_local` does not work with DCE and issue a fix for it.
- We mentioned before that for each P4 program we used in our simulations we developed a C++ client to program bmv2. This was needed because `runtime_CLI.py` could not be ported in DCE. These clients are also simulated by DCE and must be started in the simulation after we have installed the `simple_switch` target on ns-3 network nodes. We noticed that these clients need to be started with a small time difference in DCE, otherwise DCE execution hangs. This must be a possible DCE bug, but, for our simulations starting bmv2 clients with a time offset solves the issue. A better approach would be to find the bug and fix it.
- Another limitation when using bmv2 in ns-3 is its scalability. This problem stems from the DCE system and not our porting. As we mentioned before, DCE virtualizes all execution under a single process. Usually, in DCE simulations most of the nodes are native ns-3 nodes while simulated applications run only on endhosts. In our case, all the nodes of the simulation run virtualized applications, endhosts run iperf servers or clients, while nodes acting as switches run the bmv2 application. This results in a large number of different threads and processes managed by the Virtualization Layer, which in turn increases the total simulation time. Unfortunately, there does not exist an easy solution for this problem since the single process model is inherent to the DCE framework. However, for bmv2 specifically, we can configure the number of egress threads from Figure 3.6 to be equal to one, since having multiple parallel threads in the single process model is useless.

All the changes described in this section have been packeted together into two *.patch* files residing in the directory *patches* of our repository.

4.3 Simulated topologies

In this section we will briefly describe the sample simulations we created with *bmv2*. For each one, we will provide the topology graph and briefly describe the logic of the P4 program. These examples demonstrate the correctness of porting *bmv2* to *ns-3* simulator and should also serve as a base for anyone that wants to develop custom simulation scripts. Furthermore, we are going to describe in more detail the Fat-Tree simulation with *bmv2*, which was also used in our experiments.

4.3.1 Sample Topologies

We implemented various toy topologies in *ns-3* simulator in order to test our port, which are illustrated in Figure 4.1. All these topologies run the *bmv2* application at network Nodes. End hosts are configurable and can run both the Linux Kernel Stack or the native *ns-3* stack. In all scenarios multiple flows are started between the hosts. In the case of hosts running on top of the Linux Kernel network stack we used the *iperf* tool, while we use the *BulkSendApplication* when hosts run the *ns-3* stack.

In the top topology, the linear, *bmv2* nodes run a very simple P4 program; all traffic received by one port is forwarded to the other. The number of nodes between hosts is configurable by the script parameters. In the bottom left topology the single *bmv2* node runs a P4 program that matches on the destination IPv4 address of packets and forwards traffic to the port connected with the appropriate server. In the bottom right topology we demonstrate a P4 program that implements the ECMP algorithm, where *bmv2* nodes on the edge, the ones where hosts are connected, hash traffic to one of the two possible paths. The P4 programs that are used in these topologies can be found in *p4code/dce/* directory of our repository and the client programs reside in *bmv2_clients/* directory.

4.3.2 Fat-Tree

In order to test the accuracy of our port we simulated the Fat-Tree topology, like the one we see in Figure 3.1, in *ns-3* using *bmv2* nodes and implemented two different load

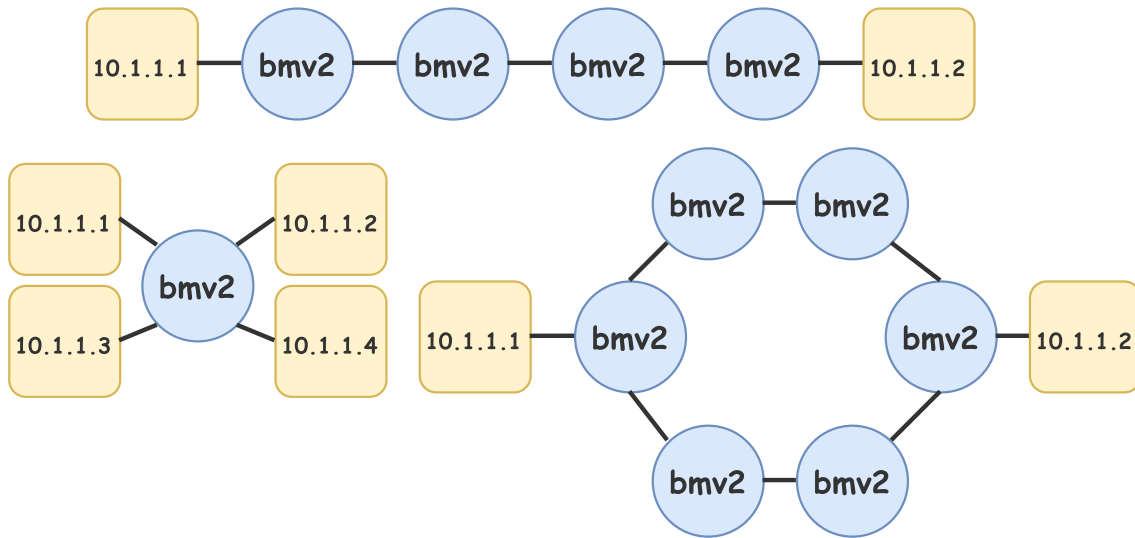


Figure 4.1: Example topologies implemented in DCE.

balancing techniques in P4 :

- *Equal Cost Multipath (ECMP)* : In ECMP [5], packets towards a destination can be delivered through multiple paths, thus, there exist multiple next hops per destination. Devices that implement ECMP hash the tuple containing the following header fields (*Protocol number, Source Ip, Source Port, Destination IP, Destination Port*) for each packet, in order to choose the output port. This ensures that packets belonging to the same flow will follow the same path in the network and no out of order packet delivery will happen at the receiver side. The P4 programs for ECMP can be found in `p4code/ecmp/` directory.
- *Flowlets*: Flowlets [38] are traffic bursts that belong to the same flow followed by an idle time interval. If this interval is large enough, then flowlets can be routed across different paths in the network without the receiver experiencing out of order packet delivery. Devices that implement flowlets keep a set of next hops per destination and a timer per flow. They measure the inter-arrival time interval of packets for each flow, and if they observe an interval greater than the flowlet gap they rehash the flow to the set of possible next hops. This technique splits flows into multiple paths and thus it achieves greater utilization of the network. The P4 programs for flowlets can be found in `p4code/flowlets/` directory.

Chapter 5

Experimental results

In this section we will present experimental results when using the behavioral model switch in the context of the ns-3 simulator. All the experiments were conducted in a virtual machine utilizing 22 CPUs in total and 90 GB of RAM.

5.1 Mininet scalability issues

The purpose of this experiment is to prove that bmv2 in Mininet does not scale as we increase the number of nodes. As we explained in Chapter 1, this was our main motivation for porting bmv2 to ns3. For this experiment we will utilize the linear topology that we described in Figure 4.1 while we vary the total number of nodes. We connect the nodes and hosts with 1 GBit/sec capacity links.

We will start a single flow, originating from host with IP address *10.1.1.1* towards the host with IP address *10.1.1.2*. We will use the *iperf* program both in Mininet and ns-3, since it is already ported with the use of DCE. We also make sure to use large TCP windows. We will plot the throughput that the iperf client reported for each different total number of nodes in the linear topology . We tried to simulate the exact same setup, both in ns-3 and Mininet, in order for the results to be comparable.

As we illustrated in Chapter 3, bmv2 consists of multiple parallel threads that implement the switch pipelines. These threads are a great overhead for the Mininet implementation, especially as the bmv2 nodes and bandwidth of the links increase, thus, we expect that throughput will collapse as we increase the number of total nodes of the topology.

In Figure 5.1 we observe the results. For the Mininet implementation we plot the average and standard deviation of the reported bandwidth across three repetitions of the experiment, while for ns-3 we report a single value because the results are deterministic across different runs.

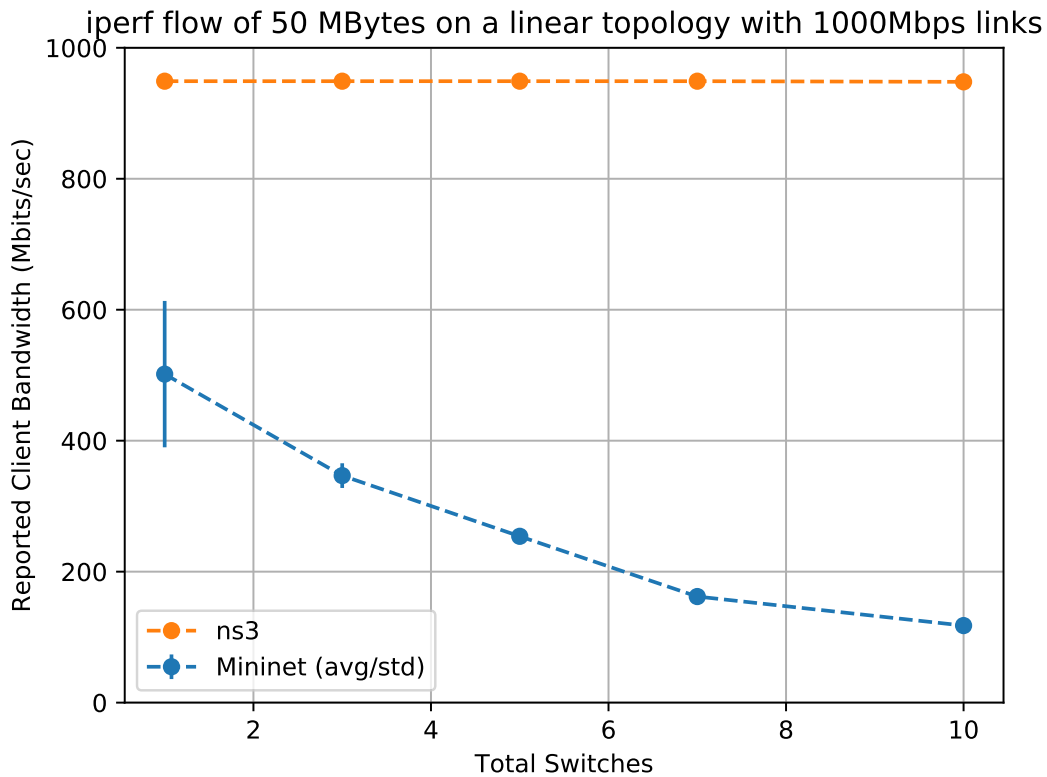


Figure 5.1: Bandwidth results for linear topology.

As we expected, the bandwidth of the Mininet implementation decreases as we add more nodes to the topology. Further, we observe that for a linear topology of 1 node, the bandwidth greatly varies between different executions of the experiment. We argue that there exist two reasons for preferring the ported bmv2 in ns-3 instead of the Mininet implementation. First, ns-3 scales and produces accurate results as we increase the nodes of the topology. Second, ns-3 based simulations produce the exact same results across runs for identical configurations. This alleviates the need of executing multiple repeats of the same experiment in order to produce statistically sound results.

In Figure 5.2 we observe the running time of the linear topology experiment in ns-3.

Unfortunately, even though the implementation scales in terms of accuracy, it does not scale in terms of execution time. As we have already mentioned in Chapter 3, this problem stems from the fact that the DCE framework virtualizes every simulated application under one process, which increases the total simulation time.

Simulation time for a 50 MB flow on a linear topology with 1000Mbps links

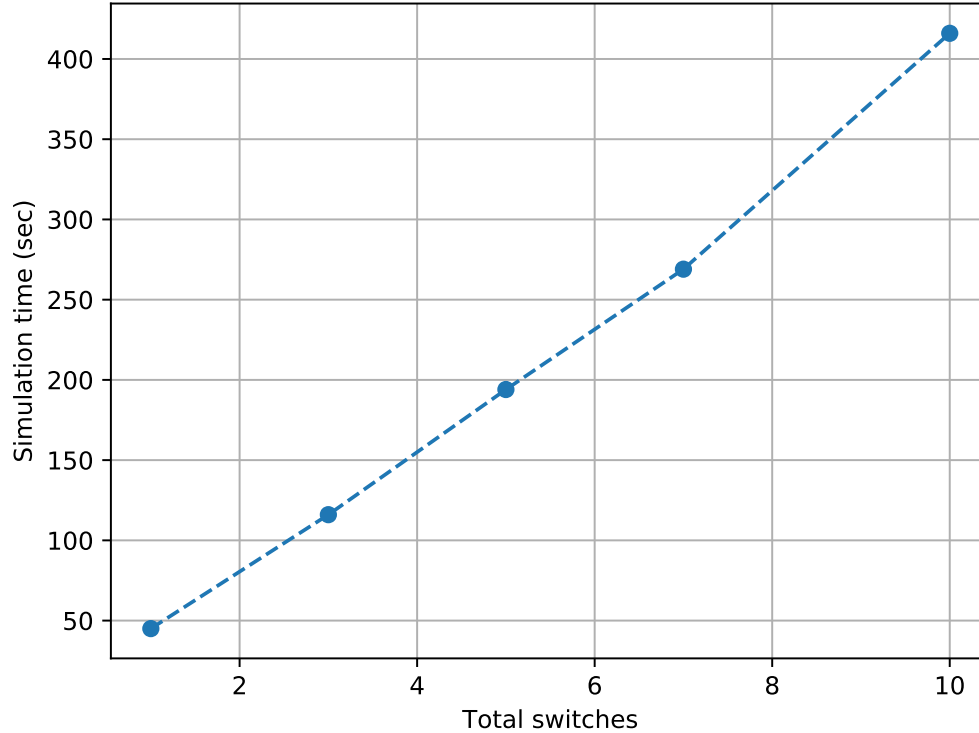


Figure 5.2: Execution time for ns-3 simulator.

5.2 Port accuracy

The purpose of the experiments described in this section is to prove that the bmv2 port works correctly in the ns-3 simulator. In order to achieve this, we executed the same P4 program in Mininet and ns-3, and observed the trend in the results.

Specifically, we implemented the Equal Cost Multipath and Flowlets techniques for Fat-Tree topologies with $k=4$. The load balancing techniques were described in Chapter 4, while the Fat-Tree topology was described in Chapter 3. In a Fat-Tree with $k=4$ there exist 16 hosts in total and each switch in the Edge layer is connected with two hosts. In our experiments, we implemented a stride traffic pattern, where each host with index i , where $i < 8$, sends a single flow to host with index $i + 8$. The links of the topology have 10 Mbit/sec capacity and the flows are of size 100 MBit, meaning that the ideal flow completion time, without any protocol overhead, should be 10 seconds. For the Mininet implementation we used the *iperf* program in order to start flows, while for the ns-3 implementation we used the *BulkSendApplication*, which was described in Chapter 3. Unfortunately, we could not use the *iperf* application for this experiment due to a bug in the ported Linux Kernel in DCE.

We need to clarify here that the purpose of this experiment is not to demonstrate that Flowlets perform better than ECMP; this is already well established, but rather to construct an artificial scenario where we expect the results to follow a certain trend and observe that in our experiments. This way, we will be sure that bmv2 was ported correctly to the simulator. In the scenario described above, we expect that ECMP will hash some flows that originate from the same Edge switch to the same uplink port, and thus congestion will occur. Since ECMP is stateless, it will not change the port for any of the flows that experience collisions, while the Flowlets technique will detect the congestion and reroute the flows. Once the flows are hashed to a path that is uncongested, the Flowlets technique will not reroute them. The flowlet gap is set to the theoretical calculated RTT in the ns-3 experiment, while in the Mininet experiment is set to a value slightly smaller than the theoretical RTT. This is done because Mininet does not emulate propagation and transmission delay as accurately as the ns-3 simulator.

In Figure 5.3 we observe the results for Mininet and in Figure 5.4 the results for the ns-3 simulator.

Flowlet implementation with gap slightly smaller than the theoretical RTT

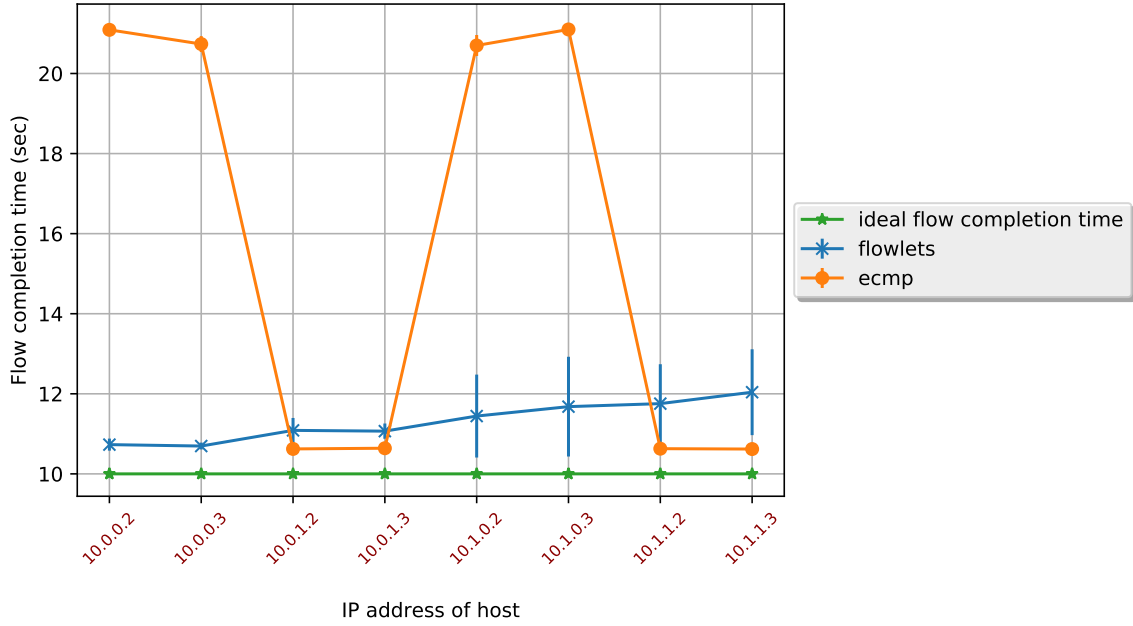


Figure 5.3: Flowlets vs Ecmp executed in Mininet.

Flowlet implementation with gap equal to the theoretical RTT

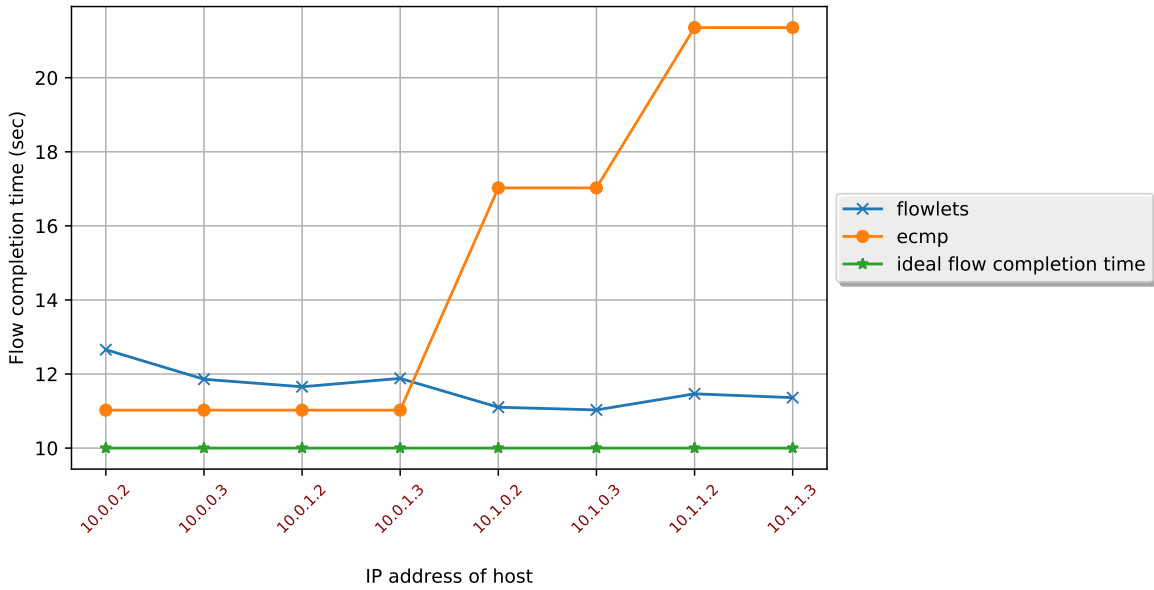


Figure 5.4: Flowlets vs Ecmp executed in ns-3.

We can see that flowlets achieve almost the ideal flow completion time for all the flows in both implementations. Further, we observe that in both graphs, in the case of ECMP, flows with an increased flow completion time originate from hosts that are connected to the same Edge switch. As we explained earlier, these flows are hashed to the same uplink port and thus they experience congestion.

However, we observe that in the Mininet implementation ECMP experiences collisions in different Edge switches than the ns-3 implementation. This is logical, because, in both implementations we can not choose the source port of the connection, which is chosen by the system from an available port range, and thus a different hash is produced. Further, any minor differences of the results should be attributed to the different network stack of the implementations. In Mininet, hosts use the native linux network stack, while in ns-3, hosts use the ns-3 native stack.

Chapter 6

Conclusions and Future work

In this thesis we successfully ported the behavioral model switch, which implements the full P4 specification, to ns-3 simulator with the use of the DCE framework.

Specifically, we first identified why we needed to port bmv2 to ns-3, in Chapter 5 we conducted experiments that verified our claims. Second, we described existing real world applications that are ported to ns-3, as well as research papers that implement ideas with P4, prototype in bmv2, and evaluate their prototypes with a network simulator. We argued that such works could greatly benefit from our contribution and ultimately use the ported bmv2 directly in ns-3. Afterwards, we described all the important system components that we used during our thesis and our extensions to the ns-3 codebase. Then, we detailed the full port procedure of bmv2 and the main challenges we faced. Lastly, we concluded with experiments that demonstrated the need of such port and also attested to the correctness of it.

Even though bmv2 was ported and executed successfully in ns-3, the port has some limitations that were briefly described in Chapter 4, which could be addressed in future improvements of this thesis. Firstly, we could develop a generic bmv2 client in C++ , just like the one provided in Python, that would read custom rules from a text file and program the switch accordingly. This client would run as a separate DCE application. In this way we would not need to create a new client for each P4 program we use, as we did in our thesis. Secondly, we could create a special NetDevice object in ns-3 that would not be tied to any specific header format or protocol usage. This would enable us to execute P4 programs that parse and understand custom Layer-2 protocols and are not tied to the Ethernet or Point-to-Point format. Lastly, we could address all the bugs that we discovered in DCE during the porting process.

Another interesting continuation for our thesis would be to gather P4 programs that were used in previous research papers [24] [27] [25] [35] [34] and run them in ns-3. This way we would evaluate our port with more complex scenarios and P4 programs, demonstrate how easy it is to run simulations with a P4 datapath, once you acquire the P4 program, and reproduce the scenarios described by the authors and compare the results with the ones reported in the papers.

Lastly, in response to the low scalability of the simulations in terms of execution time, we could try and utilize the Message Passing Interface (MPI) mode [31] of the simulator. This way we would partition the topology in different logical processing units, which could potentially reduce the total running time and enable us to simulate larger topologies with higher traffic demands.

Appendix A

Instructions for porting bmv2 to ns-3 for an Ubuntu VM

These instructions guide an ns-3 developer to port the behavioral model to the ns-3 simulator and run meaningful simulations with it. The instructions are quite involved, due to the different projects that need to be built and linked together.

All the clones in this instruction set will take place in the folder `~/git`. Our repository is called `P4_LB`. All the patches are placed in directory `patches/` of our repository. Also, we will be building everything from source, which means that we will download everything needed and build it without the use of `bake` or other tools. This will make the instructions longer but more clear.

Start the instructions with booting up a clean Ubuntu 14.05 image. In our vm the user is called `user` and the password is `user`. We recommend you to do the same in order for the instructions to have some compatibility.

A.1 ns3 requirements

Go to <https://www.nsnam.org/wiki/Installation#Ubuntu.2FDebian> and install all ns3 requirements. For ease of use you will find a script called `to_install.sh` in the repository.

Assuming you cloned our repository execute :

```
$ cd P4_LB
$ sudo bash to_install.sh
```

A.2 Install clang and clang++

```
$ sudo apt-get install clang-3.9 clang++-3.9
$ sudo ln -s /usr/bin/clang++-3.9 /usr/bin/clang++ && sudo ln -s /usr
↪ /bin/clang-3.9 /usr/bin/clang
```

A.3 Install cmake 3.4 from source

```
$ git clone https://github.com/Kitware/CMake.git
$ cd CMake && ./configure && make && cd bin && sudo rm /usr/bin/cmake
↪ && sudo ln -s /home/user/git/CMake/bin/cmake /usr/bin/cmake
```

A.4 Download and build libcxx and libcxxabi from llvm

We will be building the bmv2 with the libc++ and libc++abi and linking against them. This is needed in order for the std::thread calls to function properly in DCE.

Follow the instructions here <http://libcxx.llvm.org/docs/BuildingLibcxx.html> in order to build libc++ and libc++abi. Clone the repos and do not build. We are going to apply a patch first.

Navigate to the libcxxabi source code and apply the libcxxxapi.patch :

```
$ cd ~/git/llvm/projects/libcxxabi/
$ patch src/cxa_thread_atexit.cpp libcxxabi.patch
```

Now build the libraries following the commands from the llvm site. Assuming we created the folder `~/git/llvm/build/` the commands are :

```

$ cd ~/git/llvm/build/
$ CC=clang CXX=clang++ cmake -G "Unix_Makefiles" ~/git/llvm
$ make cxx
$ export LIBCXX_PATH=~/git/llvm/build/lib
$ export CXX_INCLUDES_PATH=~/git/llvm/build/include/c++/v1
$ export LD_LIBRARY_PATH=$LIBCXX_PATH:$LD_LIBRARY_PATH

```

Add the export commands to `~/.bashrc` also. These variables are used by the compiler and the linker.

A.5 Download, patch and build libpcap from source

Execute the following :

```

$ git clone https://github.com/the-tcpdump-group/libpcap.git
$ cp ~/git/P4_LB/patches/libpcap.patch .
$ git checkout 0e37306a5c80ea5256c96402ac7bdcc9954b06cc
$ patch pcap-linux.c libpcap.patch
$ CFLAGS="-fPIC -U_FORTIFY_SOURCE" ./configure && make
export LIBPCAP_PATH=~/git/libpcap
$ sudo make install
$ export LD_LIBRARY_PATH=$LIBPCAP_PATH:$LD_LIBRARY_PATH

```

Add the export command to `~/.bashrc` also.

A.6 Download and build boost with clang++ and libc++

```

$ wget https://dl.bintray.com/boostorg/release/1.64.0/source/
  ↪ boost_1_64_0.tar.gz git clone https://github.com/boostorg/boost.
  ↪ git
$ tar -xvf boost_1_64_0.tar.gz
$ cd boost_1_64_0/
$ ./bootstrap.sh --prefix=/usr/local/
$ sudo ./b2 toolset=clang cxxflags="-std=c++11 -stdlib=libc++ -
  ↪ I$CXX_INCLUDES_PATH" linkflags="-stdlib=libc++ -L$LIBCXX_PATH"
  ↪ link=shared install

```

```
$ export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
```

A.7 Configure LD_LIBRARY_PATH

Execute the command :

```
$ export LD_LIBRARY_PATH=/usr/local/lib:/usr/lib/x86_64-linux-gnu/:/  
↪ lib/x86_64-linux-gnu:/lib64:$LD_LIBRARY_PATH
```

A.8 Download and build thrift server

bmV2 needs Apache Thrift to work. We need to clone it and build it from source while linking with libc++ / libc++abi, instead of libstdc++.

```
$ git clone https://github.com/apache/thrift.git  
$ cd thrift/  
$ ./bootstrap.sh  
$ CC=clang CXX=clang++ CFLAGS="-fPIC -U_FORTIFY_SOURCE" CXXFLAGS="-  
↪ fPIC -U_FORTIFY_SOURCE -std=c++11 -stdlib=libc++ -  
↪ I$CXX_INCLUDES_PATH" LDFLAGS="-pie -rdynamic -L$LIBCXX_PATH" ./  
↪ configure --disable-tests  
$ make  
$ export LIBTHRIFT_PATH=~/.git/thrift/lib/cpp/.libs  
$ export LIBTHRIFT_INCLUDES_PATH=~/.git/thrift/lib/cpp/src  
$ export LD_LIBRARY_PATH=$LIBTHRIFT_PATH:$LD_LIBRARY_PATH  
$ sudo env LD_LIBRARY_PATH=$LD_LIBRARY_PATH make install
```

Add the following comments to `~/.bashrc` also. For every new project that we will build we will add its library directories to `LD_LIBRARY_PATH` variable, in order for the linker to find them.

A.9 Download, patch and build behavioral model and custom thrift clients

The behavioral model is the target application that we want to run in DCE. Each switch node in DCE is actually a linux-kernel enabled DCE node, because bmv2 is an application that runs on a linux box. In order for this to run correctly it needs some minor patches.

All the problems have been described in detail in Chapter 4.

Execute the following :

```
$ git clone https://github.com/p4lang/behavioral-model.git
$ cd behavioral-model/
$ git checkout eed6954af191174405b7345e6ad7571bc30fca39
$ cp ~/git/P4_LB/patches/bmv2.patch .
$ patch -s -p0 < bmv2.patch
```

Now configure and build bmv2, from bmv2 directory execute :

```
$ ./autogen.sh
$ CXX=clang++ C=clang LDFLAGS="-L$LIBCXX_PATH -pie -rdynamic -lthrift"
  ↪ CXXFLAGS="-fPIC -U_FORTIFY_SOURCE -stdlib=libc++ -
  ↪ I$CXX_INCLUDES_PATH" CFLAGS="-fPIC -U_FORTIFY_SOURCE" ./
  ↪ configure --without-nanomsg --without-targets --without-stress-
  ↪ tests --disable-logging-macros --disable-elogger
$ make && cd targets/simple_switch && make
$ export LD_LIBRARY_PATH=/home/user/git/behavioral-model/thrift_src/.
  ↪ libs:./git/behavioral-model/targets/simple_switch/.libs/:
  ↪ $LD_LIBRARY_PATH
$ export BMV2_PATH=~/.git/behavioral-model
```

Add the export command in the `~/.bashrc`.

As we explained in Chapter 4 the *runtime_CLI* is written in Python, thus we can not run it in DCE. For all the example scripts we have created we also created the appropriate clients written in C++, which are compiled and run in DCE.

We need to copy the client programs from our repository into bmv2 directory and build them. Execute the following :

```
$ cd ~/git/behavioral-model/thrift_src/
$ cp ~/git/P4_LB/bmv2_clients/* .
$ bash make_clients.sh
$ export DCE_PATH=$BMV2_PATH/thrift_src/:$DCE_PATH
```

Also add the export command in `~/.bashrc`

A.10 ns3 build and install

Assuming you cloned our repository, execute :

```
$ cd P4_LB
$ cd ns-3.26/
$ ./waf configure
$ ./waf build
$ sudo ./waf install
$ export NS3_DIR=~/git/P4_LB/ns-3.26
$ export NS3_DIR=$NS3DIR
```

Add the export commands to `~/.bashrc` also. The previous commands will build and install ns-3 to `/usr/local/lib`.

A.11 Build latest Linux Kernel for use

We will be running a new version of linux kernel with DCE. This is needed in order for the libpcap and the thrift server that is part of the bmv2 to work correctly. Execute the following instructions, which were found at <http://direct-code-execution.github.io/net-next-sim/> :

```
$ git clone https://github.com/libos-nuse/net-next-nuse
$ cd net-next-nuse/
$ make defconfig ARCH=lib
$ make library ARCH=lib
$ export $KERNEL_PATH=/home/user/git/net-next-nuse/arch/lib/tools
$ export $KERNEL_PATH=/home/user/git/net-next-nuse/arch/
```

Add the export command to `~/.bashrc` also.

After these commands you should have the file `libsim-linux.so` in the folder `~/net-next-nuse/arch/lib/tools`. This is the binary that DCE must be able to find at all times in order to boot the linux kernel.

A.12 Configure and build DCE

Now we are going to build ns-3-dce with our version of ns3 and use the kernel we just built before. We will use a clean dce repository which we will patch. Execute :

```
$ git clone https://github.com/direct-code-execution/ns-3-dce.git
$ cd ns-3-dce
$ cp ~/git/P4_LB/dce_* .
$ patch -s -p0 < dce_model.patch
$ patch -s -p0 < dce_myscripts.patch
$ ./waf configure --with-ns3=/usr/local/ --enable-kernel-stack=
  ↪ $KERNEL_ARCH_PATH
$ ./waf build
```

The p4 examples that use `simple_switch` read the json files from root directory `/`. Thus, you will need to copy the compiled p4 .json files into `ns-3-dce/files-*/` in order for dce to find it. The compiled programs reside in `P4_LB/`.

Now we will configure the `$DCE_PATH` variable that enables DCE to find executables .

```
$ export DCE_PATH=$KERNEL_PATH:~/git/behavioral-model/targets/
  ↪ simple_switch/.libs
```

Add the export command to `~/.bashrc` also.

A.13 Download and build the ip program

We need to build ip and iperf programs since they are widely used in DCE simulation scripts.

First, download and build appropriately the ip utility program.

Execute :

```
$ git clone https://github.com/shemminger/iproute2.git
$ cd iproute2
$ ./configure
```

Edit the Makefile and change lines 30,31 to :

```
CC = gcc -fPIC -U_FORTIFY_SOURCE
HOSTCC = gcc -fPIC -U_FORTIFY_SOURCE
```

And line 39 of the Makefile to :

```
LDLIBS += $(LIBNETLINK) -pie -rdynamic
```

Then execute :

```
$ make
$ export DCE_PATH=~/.git/iproute2/ip:$DCE_PATH
```

Add the export commands to `~/.bashrc` also.

A.14 Download and build the iperf program

```
$ git clone https://github.com/daynix/iperf2
$ cd iperf-2.0.5-stable
$ /configure CFLAGS="-O2 -fPIC -U_FORTIFY_SOURCE" CXXFLAGS="-O2 -fPIC -
↳ -rdynamic" LDFLAGS="-pie -rdynamic"
```

Add `sleep(1)` at line 412 of `compat/Thread.c` in order for iperf to work correctly in the DCE context.

Continue with the commands :

```
$ make
$ export DCE_PATH=~/.git/iperf-2.0.5-stable/src:$DCE_PATH
```

Add the export command also to `~/.bashrc`.

Up to here, we should have everything we need to run the examples. The DCE knows where to find iperf, ip, simple_switch and the linux kernel that we just build from source.

In all examples, hosts initiate flows, either with the iperf program when running linux kernels, or with a modified version of the bulk-send application with the ns-3 network stack.

In order to execute the bmv2 switches the switch needs to know where to find the appropriate json configuration that describes the p4 program. The filesystem of DCE nodes resides in *files-**, where *** is the node id in DCE, thus, we are going to copy the json files in there.

Execute the following :

```
$ cd ~/git/ns-3-dce/  
$ cp ~/git/P4_LB/p4_code/dce/copy_p4.sh .  
$ bash copy_p4.sh
```

Now we have copied all the compiled p4 programs into the correct directories for DCE to find them and we can run the examples.

Execute :

```
$ ./waf --run dce-bmv2  
$ ./waf --run dce-bmv2-server-fwd  
$ ./waf --run dce-ecmp
```

If the execution hangs, then possibly you have not removed the `thread_local` variables from the bmv2 switch. If you use a newer version of bmv2 switch that introduces more variables of that type, make sure to convert them to normal variables. However, also make sure that the functionality is not altered.

Bibliography

- [1] Behavioral model repository. <https://github.com/p4lang/behavioral-model>. Accessed: 2017-07-14.
- [2] Bmv2 tutorial slides. https://sched.ws/hosted_files/2016p4workshop/9f/Barefoot%2C%20Antonin%2C%20P4%20workshop%202016.pdf. Accessed: 2017-07-14.
- [3] Dce: How to add a system call. <https://www.nsnam.org/docs/dce/manual/html/dce-user-syscalls.html>. Accessed: 2017-07-14.
- [4] Dce tutorial. <https://www.nsnam.org/docs/dce/manual/html/how-it-works.html>. Accessed: 2017-07-14.
- [5] ECMP Load Balancing. Technical report, Cisco.
- [6] <https://www.nsnam.org/docs/models/html/internet-stack.html>. <https://github.com/nsnam/ns-3-dev-git>. Accessed: 2017-07-14.
- [7] *ns-3 Direct Code Execution (DCE) Manual*. Accessed: 2017-07-14.
- [8] *ns-3 Direct Code Execution (DCE) Quagga Manual*. Accessed: 2017-07-14.
- [9] Ns-3 overview page. <https://www.nsnam.org/docs/tutorial/html/conceptual-overview.html>. Accessed: 2017-07-14.
- [10] Ns-3 repository. <https://github.com/nsnam/ns-3-dev-git>. Accessed: 2017-07-14.
- [11] Ns-3 scheduler page. <https://www.nsnam.org/docs/manual/html/events.html>. Accessed: 2017-07-14.
- [12] Nuse repository. <https://github.com/libos-nuse/net-next-nuse>. Accessed: 2017-07-14.

- [13] Onos operating system. <http://onosproject.org/>. Accessed: 2017-07-14.
- [14] Open vswitch project page. <http://openvswitch.org/>. Accessed: 2017-07-14.
- [15] P4 language evolution. <http://p4.org/p4/p4-language-evolution/>. Accessed: 2017-07-14.
- [16] P4 tutorial slides. https://github.com/p4lang/tutorials/blob/master/SIGCOMM_2016/p4-tutorial-slides.pdf. Accessed: 2017-07-14.
- [17] Pox and ryu integration repository. <https://github.com/jaredivey/dce-python-sdn>. Accessed: 2017-07-14.
- [18] Quagga project page. <http://www.nongnu.org/quagga/>. Accessed: 2017-07-14.
- [19] Tofino, product page. <https://barefootnetworks.com/technology/>. Accessed: 2017-07-14.
- [20] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08*, pages 63–74, New York, NY, USA, 2008. ACM.
- [21] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC '10*, pages 267–280, New York, NY, USA, 2010. ACM.
- [22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
- [23] G. Carneiro, P. Fortuna, and M. Ricardo. Flowmonitor: A network monitoring framework for the network simulator 3 (ns-3). In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS '09*, pages 1:1–1:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
- [24] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research, SOSR '17*, pages 61–74, New York, NY, USA, 2017. ACM.
- [25] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research, SOSR '16*, pages 10:1–10:12, New York, NY, USA, 2016. ACM.

-
- [26] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX*, pages 19:1–19:6, New York, NY, USA, 2010. ACM.
- [27] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16*, pages 101–114, New York, NY, USA, 2016. ACM.
- [28] E. Mancini, H. Soni, T. Turetti, W. Dabbous, and H. Tazaki. Demo abstract: Realistic Evaluation of Kernel protocols and Software Defined Wireless Networks with DCE/ns-3, 2014. Demo Abstract in Proceedings of ACM MSWiM, Montreal, Canada, September 21-26 2014.
- [29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.
- [30] The P4 Language Consortium. *The P4 Language Specification*. Accessed: 2017-07-14.
- [31] J. Pelkey and G. Riley. Distributed simulation with mpi in ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques, SIMU-Tools '11*, pages 410–414, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
- [32] L. R. Prete, A. A. Shinoda, C. M. Schweitzer, and R. L. S. de Oliveira. Simulation in an sdn network scenario using the pox controller. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6, June 2014.
- [33] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16*, pages 525–538, New York, NY, USA, 2016. ACM.
- [34] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research, SOSR '17*, pages 164–176, New York, NY, USA, 2017. ACM.
- [35] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research, SOSR '17*, pages 164–176, New York, NY, USA, 2017. ACM.

- [36] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabous. Direct code execution: Revisiting library os architecture for reproducible network experiments. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 217–228, New York, NY, USA, 2013. ACM.
- [37] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, Oct. 2007.
- [38] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 407–420, Boston, MA, 2017. USENIX Association.