OPTIMIZING NETWORKING APPROACHES USING P4 PROGRAMMING

By

Debopam Sanyal

Senior Thesis in Computer Engineering

University of Illinois at Urbana-Champaign

Advisor: Professor Sibin Mohan

May 2020

Abstract

Forwarding data packets between switches is the basis of propagation of information in computer networks of all types. Hence, in order to ensure optimal utilization of hardware and computing resources, it is imperative to make the forwarding process fast, efficient and resilient. Decentralizing intelligence in networks often makes packet transmission time-consuming as there is no separation between the forwarding and routing processes. In this project, I focused specifically on reducing this time by implementing the forwarding decisions of the switch in the data plane and leaving the control plane to handle the routing process. This is a well-known feature of software-defined networks (SDNs). In most cases, switches forward packets depending on their characteristics. Therefore, I focused on this conditional basis of forwarding at the switch level.

I implemented three different forwarding paradigms for two topologies: (a) utilizing unused, but optimal topology paths, (b) placing importance on the urgency of data flows and (c) guaranteeing connectivity. I used the P4 programming language over the OpenFlow protocol since the latter does not provide complete protocol independency. The evaluation process involved testing the programs on the mininet network emulator first and then on the SDNet and SUME simulations of the NetFPGA-SUME.

The goal of this project is to provide end-to-end deadline guarantees for real-time networks using an SDN by programming the switch data plane. This consists of developing and evaluating (i) deadline-driven (end-to-end delay guarantees) and (ii) budget-driven packet scheduling algorithms and (iii) providing resilience in case of link failures. The broader mission is to forward packets in an optimal fashion.

Subject Keywords: software-defined networking; forwarding; data plane; switch; P4; OpenFlow

Acknowledgments

I would like to express my deepest gratitude to my advisor Prof. Sibin Mohan for guiding me throughout my research. I appreciate his help in all the areas of the project. I would also like to extend my thanks to graduate students Ashish Kashinath, who provided me with multiple resources essential for conducting the research and Rakesh Kumar, who helped me in coming up with the conditional forwarding implementation ideas.

Contents

1. Introduction

Observed from a big picture perspective, SDNs have control plane centralization that calls for the switches to be non-intelligent forwarding devices whereby the control plane makes all the decisions. Fast failover, in essence, breaks the model above [1]. It makes a local switch-native decision, without calling out the controller in the event of a link going down. This is done so that the latency hit (i.e. going to the controller for an alternative route) can be avoided. The cost of using fast failover is pre-planned paths. Now, the switch does not only need to know the primary path, but also the failover paths. Such planning requires computation, which is mostly done at design time.

Conditional Forwarding, as presented in this thesis, extends the motivation of fast failover further. It is by the way, just a mechanism. It does not, in of itself, guarantee anything; just like the fast failover mechanism does not guarantee anything. A conditional forwarding enabled switch is capable of doing some more smart things. The switch in this case makes a forwarding decision for a flow based on the state of some other flow. The other flow likely has a different (more likely higher) priority level in terms of forwarding class. The state is mere presence in a given time window (has the switch seen a packet in say *t* seconds?) in experiment 1, while it is a comparator (is one host's packet generation rate greater than that of the other host?, is the link traffic greater than the link capacity?) in experiments 2 and 3. The design problem that is addressed here is whether such a mechanism can be constructed so that it can still provide guarantees on end-to-end delay and connectivity, even when failures occur.

1.1 Software-defined Networking

Switches are at the forefront of the majority of Ethernet/IP (industrial protocol) and TCP (transmission control protocol)/IP (internet protocol) networks including local area network (LAN) and wide area network (WAN) connections. They play a central role in the effective functioning of SDNs [2]. Previous research in this area has shown the benefits of separating packet forwarding at switch level from the control plane that mainly oversees the routing in a network [2,3,4]. Software-defined networking emerged due to the advent of cloud computing and the complexity of real-world IP networks, that mostly had vertical integration in them: a common control and data plane. Moreover, SDNs also promised innovation through network programmability [4]. The basis of the paradigm lies in the separation of network logic (control plane) and switches/forwarders (data plane). This results in the centralization of intelligence within the control plane/controller while empowering the user with the ability to dynamically program the network configuration [2]. Hence, switches act as mere packet

forwarding devices while the control plane is in charge of determining the overall behavior of the network [3]. Related work has shown that implementing SDN style networks allows more flexibility by improving network management and simplifying the introduction of new abstractions in networks [5]. However, a number of flaws mainly in security and scalability have been well documented in SDNs [6].

1.2 OpenFlow and P4

OpenFlow has led SDN related research since it rose to prominence among researchers in 2011. Among other advantages, it provides efficient ways to design the controller's interfaces with the applications and the data plane [7]. At its core, OpenFlow is no more than a protocol definition of an application programming interface (API) that both the controller and the switches should abide by. It merely defines a system of rules, syntax and semantics that allow multiple entities in a communication system to interact with each other by transmitting information. The execution works with a set of flow tables that decide how packets are handled by the switches and the controller. This design style enables network programmers to ascertain the path of data packets across all the switches in the network. Although well suited to SDN research, OpenFlow has a major drawback: it is protocol dependent. This means that it supports protocols for common packet types like TCP/IP and Ethernet/IP but doesn't allow the switch programmer to define their own type (headers and fields). As shown in chapter 4, many useful applications such as tunneling become impossible and others like load balancing become much harder to implement. Therefore, a relatively new networking language, the P4 programming language, was utilized in this research. Chapter 2 expands on the reasons behind the preference of P4 over OpenFlow for this research's purpose.

1.3 NetFPGA-SUME Simulations

The experiments in this project were conducted by building networking algorithms using P4 and applying them to forwarders. The NetFPGA-SUME board from Digilent, that has a Xilinx Virtex-7 FPGA, was used as the target hardware on which the networking systems were constructed. The board is the outcome of a collaborative effort between Digilent, the University of Cambridge and Stanford University. As one of its main missions, the board seeks to be a state-of-the-art platform for networking for students who want to learn the fundamentals [8]. It hopes to provide an accessible development environment that enables creation of novel designs [9]. Researchers have found the NetFPGA useful when building high-performance networking systems [10]. While other boards show similar promise in terms of performance, the NetFPGA board has an edge over the others: its FPGA logic can be successfully utilized to implement the data processing functions independently while the control

functions can be implemented by a host computer that runs programs. This makes the platform ideal for designing routers. In this project, the board was used as a stand-alone device, which is one of its many unique capabilities along with being open source. This enables greater collaboration and higher-quality research.

In Chapter 2, a critical review is presented of prior work published in the literature that is directly relevant to the problems addressed in this thesis. A brief explanation on how the work described in the thesis supersedes some aspects of the previous approaches is given while outlining the research context of the work. Chapters 3 and 4 describe the main body of the research work undertaken in this study. It contains the methodology and experimental techniques applied to the research problem. In addition to the three models of conditional forwarding, these chapters contain the testing procedures used on mininet [11] first and then on the NetFPGA-SUME's SDNet and SUME simulation test beds. Finally, chapters 5 and 6 summarize the main takeaways from the project, emphasizing on the usefulness of P4, advantages of SDN, the conditional forwarding implementations, a few limitations of this research and scope for future work.

2. Literature Review

This section discusses some of the major pieces of related work on programmable devices and SDNs. Relevant publications not only discuss the history and evolution of SDNs and OpenFlow, but also the motivation behind their development [2, 4, 6, 7, 12]. SDNs are constantly evolving, being driven by emerging trends of virtualized cloud computing, mobility and big data applications [12]. An SDN's two standout features: network programmability and separation of the control and data planes are highlighted in almost all papers covering them [2, 3, 6, 7]. Efficient network management can be achieved by creating modular applications, supported on SDN platforms, from independent modules that jointly manage network traffic [13]. Large scale adoption of OpenFlow is encouraged for its proven benefits in running experimental protocols in everyday networks [14]. However, because OpenFlow is protocol-dependent, its current version explicitly specifies protocol headers on which it operates [15]. The rise in these headers has increased complexity without providing the flexibility to add new headers. Hence, P4, a protocol-independent networking language was used in this research. The three main goals of the P4 community can be summarized as: (a) reconfigurability in the field, (b) protocol independence and (c) target independence. The NetFPGA-SUME board is covered extensively in recent work [9]. An interesting claim made by its developers is that the board, when acting as a stand-alone device, can be used to explore the datacenter interconnect, fulfilling Thacker's vision [16] and bringing Input/Output (I/O) and network research closer to the CPU-architecture community.

Previous research in closely related areas has focused on network resiliency. For example, the concepts of link failure recovery or congestion notification systems have been around since the beginning of the internet. Chu et al. [17] show a way to guarantee 100% traffic reachability in hybrid SDN networks (traditional IP routers and SDN switches coexisting) by redirecting traffic from the failed link to SDN switches and then using these switches to explore possible backup paths for the post-recovery network. Behesti and Zhang [18] propose algorithms to improve network resiliency by maximizing the possibility of fast failover (doesn't contact the controller and allows switch to make local decisions). But this mechanism needs a large storage of primary and failover paths. Wang and Shi [19] consider budget and deadline constraints on scheduling algorithms. To the best of my knowledge, my work is the first to propose models for *optimizing and providing guarantees on an SDN by the means of only programming the switch data plane*.

3. Implementation

3.1 The use of the Programming Language P4

This section covers some aspects of P4 and what attributes of it was used in the work. In traditional networks, when building a system, we first purchase a packet processing chip that defines how the rest of the system will be built as well as how the control plane interacts with the data plane. Changing a feature/protocol in this case is extremely difficult. P4 provides a programmable data plane that reverses this design philosophy by defining packet processing rules at a high-level via a P4 program. This program can then be compiled on numerous P4 programmable devices [15]. Figure 1 shows the workflow steps. This way designers get to select the features they want in their network without having to let third-party chip vendors make those decisions for them. P4 itself is based on a high-speed packet processing device called an architecture description; the V1 Model Architecture was used in this research [15]. This contains a parser that the user uses to define the headers they would like their network to recognize and their order within an incoming packet. It is in the Match Acton Pipeline (MAP) that the user defines the tables and packet processing algorithms that they would like to use in their switches. Finally, the deparser is responsible for putting the final headers back together in the user's desirable order to form the final outgoing packet. Figure 2 depicts a typical architecture description. The parser, MAP and the deparser are all programmable. The vendor of a particular architecture also provides extern libraries that are not defined in P4 but can be invoked by P4 programs like it invokes its core libraries.

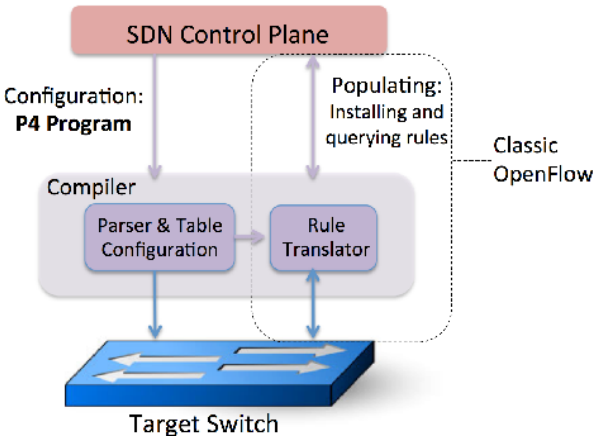

Figure 1 Using P4 in SDNs

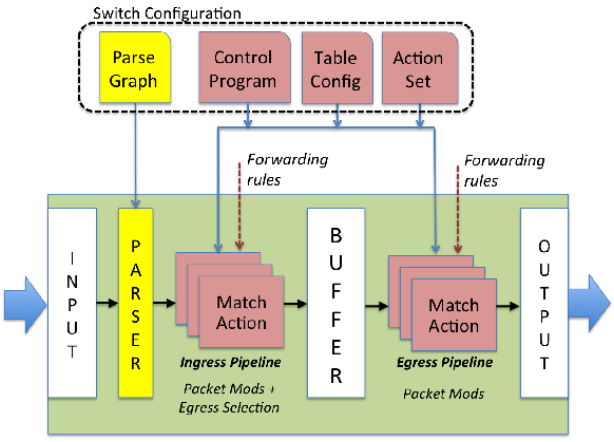

Figure 2 An architecture description

In the V1 architecture, the switch has ingress and egress ports. In P4, *ingress_port* is the port on which the packet arrived, *egress_spec* is the port to which the packet should be sent to and *egress_port* is the port from which the packet is departing (read only in the pipeline). A typical program can be divided into

the following blocks: headers, parser, checksum verification, ingress processing, egress processing, checksum update, deparser and the switch (an instance of the V1 model). For evaluation purposes, table entries were used as the main algorithmic components. A runtime control API is required so that the user can insert, remove or modify table entries after compiling the P4 program and loading it into the data plane. The runtime control API used for the switches was the P4 Runtime because it is both target-independent and protocol-independent. To generate and receive packets, python scripts were used: *send.py* that can produce packets and alter the packet generation rate at the source host and *receive.py* that is used to sniff data packets at the end host. Mininet is a *network emulator* which creates a network of virtual hosts, switches, controllers, and links [11]. A mininet emulation is shown in figure 3.

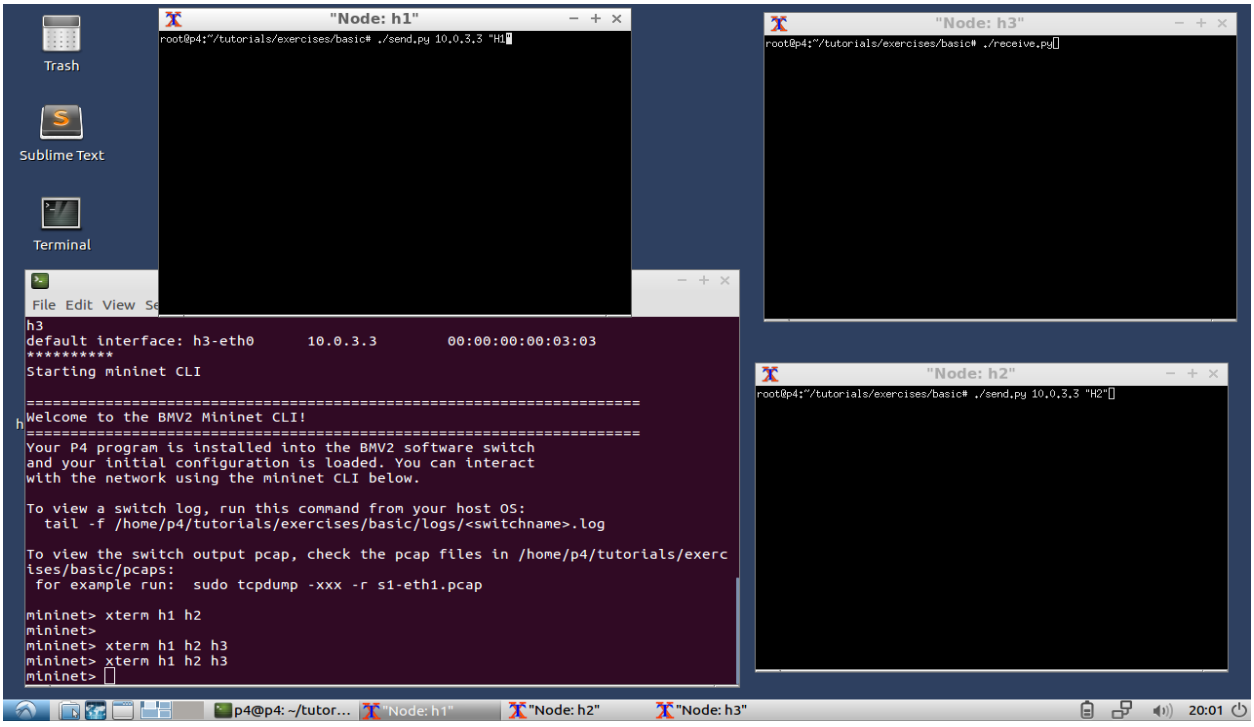

Figure 3 A sample mininet emulation with three hosts: h1, h2 and h3

3.2 Testing the Models

All the programs were compiled using the P4 compiler and their functionalities tested on mininet. Correctness of each experiment was determined by checking the auto-generated log files of all the switches in the network. An example log file is shown in figure 4. Testing the models on the NetFPGA SUME simulations and board proved to be an arduous task. A lot of technical difficulties had to be overcome, such as incorrect hardware specifications, Vivado [20] and SDNet [21] licensing issues along

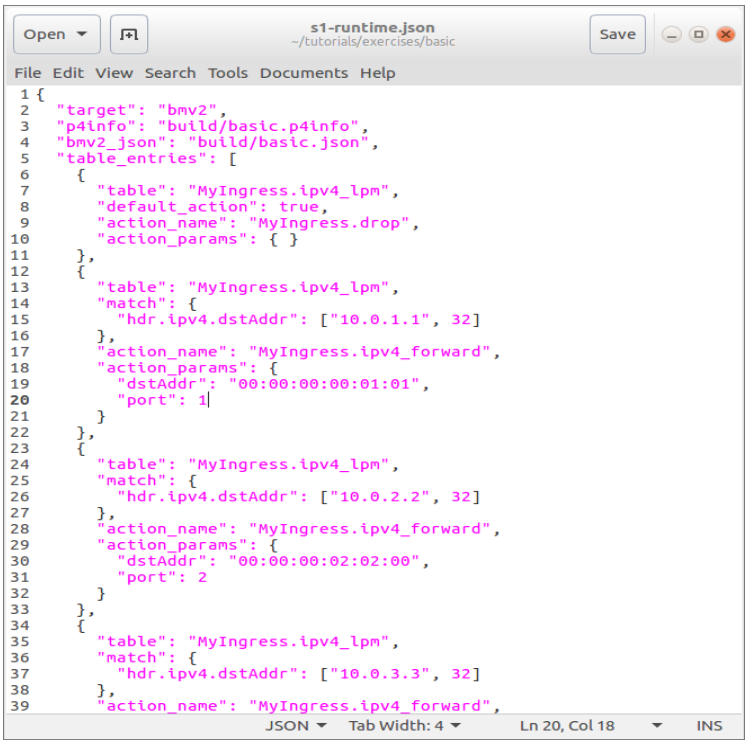

Figure 4 A sample log file of a switch in basic forwarding

with incompatible versions of software. The first step was to run the acceptance test [22] on the board in order to determine if it had any faults. After installing the correct version of Vivado and getting the SDNet license, the P4 programs were tested using the Xilinx P4-SDNet toolchain and the NetFPGA-SUME reference switch design. To verify correctness of each program, *gen_testdata.py* files, a python script that generates test data to be used in simulations, were written. On running the P4-SDNet compiler, a resulting hardware description language (HDL) was generated and a simulation framework initiated. Next the SDNet simulation was executed. The SUME simulation was run similarly with *run.py* files, that read the test packets generated by *gen_testdata.py*. A useful debugging tool was the HDL waveforms on the Vivado graphical user interface (GUI). After some careful debugging, successful SDNet and SUME simulations of all the three designs were obtained. The block diagram in figure 5 represents the procedure followed for testing.

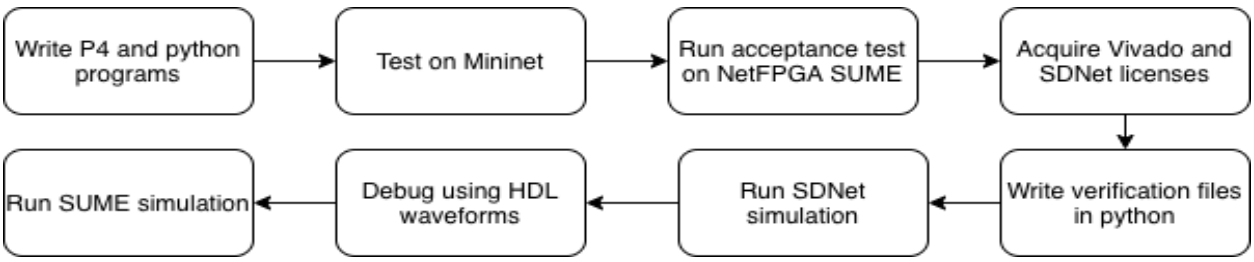

Figure 5 Block diagram to show the testing steps

4. Experiments and Results

4.1 Conditional Forwarding

Conditional forwarding can be implemented in simple programs such as basic forwarding, tunneling and routing [23]. Since tunneling involves unwrapping layers of headers, in OpenFlow, any header outside the network protocol specifications of OpenFlow crashes the network. Load balancing in a WAN also becomes tedious due to different types of protocol headers in traffic from multiple WANs. In all of the experiments, the switches broadly had the following functions: (1) update destination address with the address of the next hop, (2) update source address with the address of the switch, (3) decrement the time-to-live (TTL) of the data packet, (4) set correct egress port for the next hop and (5) update the state variable based on the latest data packet. The switch had logic only to decide whether to forward, drop or leave a packet alone based on its header (by looking up table entries). For experiments 1 and 2, the topology depicted in figures 6 was utilized.

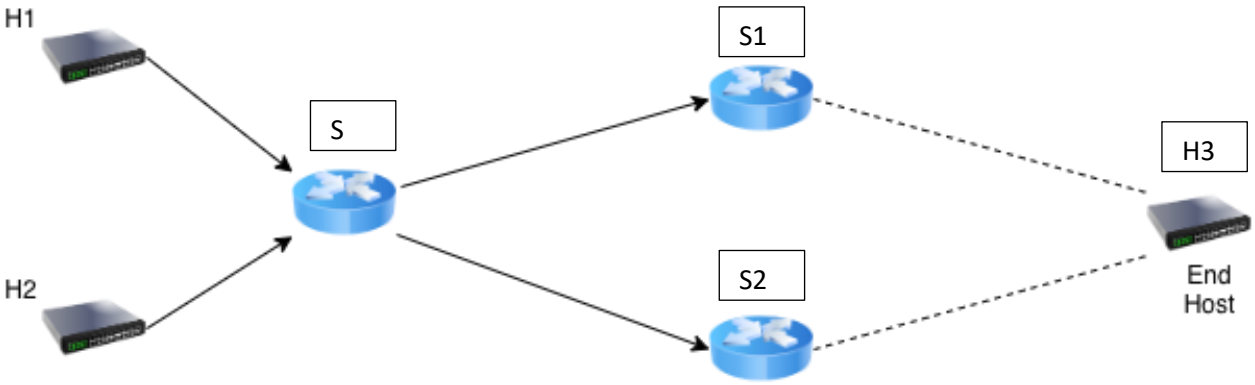

Figure 6 Topology for Experiments 1 and 2

4.2 Experiment 1

The switch *S* is allowed to maintain a state variable that is a representation of a time window and check for the header type of every entering packet. Header types have different sources and priorities, but the same destination. Each header type has a pre-defined path and the switch *S* is in all the corresponding paths. The switch *S* makes the forwarding decisions based on the state of the variable. The variable represents a time window in seconds. Example: *H1*-> higher priority, *H2*-> lower priority. If the time elapsed since the last seen *H1* data packet at *S* is greater than or equal to a predetermined time *t* in seconds, the switch *S* chooses to forward *H2* data packets using *H1*'s path (to *S1*) until the switch encounters another *H1* data packet, which is when packets of *H1* and *H2* are again forwarded in their original paths by the switch. This implementation's premise is based on the assumption that *H1*'s path is

more optimal (takes less time) than *H2*'s path. Different paths in the topology merely translate to different egress ports of the switch. Checking and updating is done on a per-packet basis at *S* and it executes all of the five functions outlined in section 4.1 once it receives a packet. This implementation optimizes the network as more packets take the optimal path to the end host. For this experiment, the following values were used: t = 5 sec and t = 10 sec.

4.3 Experiment 2

The second implementation is built on the same system as described in experiment 1 but the forwarding condition changes, in that it involves giving precedence to a flow with higher packet generation rate (more critical). In this implementation, when the rate of *H2* exceeds that of *H1* (even though *H1* has a higher priority than *H2*), the switch *S* starts forwarding *H2* packets to *H1*'s original path (to *S1*) and *H1* packets to *H2*'s original path (to *S2*; *H1*'s original path is more optimal). This continues until *H2*'s rate becomes less than or equal to *H1*'s rate, which is when packets of *H1* and *H2* are again forwarded in their original paths by *S*. The state variable here is a tuple of the packet generation rates of *H1* and *H2* which are directly obtained from the *send.py* file. The rates can be changed using this python script. Here, priority does not matter as long as the flow being given preference has a higher packet flow rate (more urgent). Similar to the previous implementation, checking and updating is done on a per-packet basis. This implementation seeks to provide deadline guarantees for the more critical flow, i.e., send its packets to the end host faster (optimally).

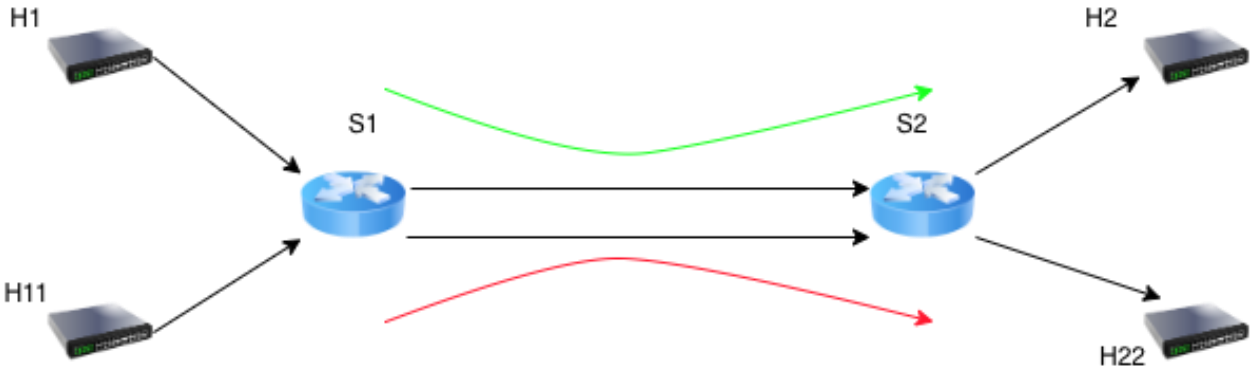

Figure 7 Topology used for Experiment 3

4.4 Experiment 3

This experiment is based on the topology shown in figure 7. Here *H1* uses the link between *S1* and *S2* to send traffic to *H2* and *H11* uses it to send traffic to *H22*. The link can handle only up to a maximum rate *R*. Once the traffic in the link exceeds this rate, the link suffers from congestion and might break; the switches are in a great position to detect congestion and reduce traffic before the link breaks. In this

model *S2* has a bit that becomes high if the traffic exceeds a threshold rate *r*, where *r* is significantly less than *R*. As soon as the bit changes to high, *S1* keeps reducing the entering traffic at its ingress ports at the rate of *k* until the bit in *S2* becomes low, i.e., traffic rate in the link becomes less than or equal to *r*. This process is often called explicit congestion notification. Again, the evaluation of the traffic rate at *S2* is done on a per-packet basis. This implementation ensures network connectivity by averting link failures due to heavy traffic. For this experiment, the following values were used: r = 10 packets/sec, R = 12 packets/sec and k = 0.5 packets/sec^2.

The results of the three experiments are summarized in table 1.

Table 1 Results of the three experiments

Experiment #	Components	Parameters	Benefits	Result
1	H1, H2, H3, S, S1, S2	t = 5 sec, 10 sec	Optimizes packet transmission in the network	Success
2	H1, H2, H3, S, S1, S2	No parameters	Provides end-to-end deadline guarantees	Success
3	H1, H11, H2, H22, S1, S2	R = 12 packets/sec, r = 10 packets/sec, k = 0.5 packets/sec^2	Ensures network connectivity	Success

5. Limitations and Future Work

There are a few limitations to the work presented here. First, only the programmability aspect of SDN was explored. The research did not concentrate on its security and scalability aspects. Thus, the results represent a very one-dimensional perspective on SDN. A holistic survey is required to measure its viability in computer networks. Second, P4 was used for only some applications of SDN, which paints a favorable picture for P4. It is certainly the case that in many areas of networking, OpenFlow trumps P4. Thirdly, the experiments haven't yet been tested on the NetFPGA hardware. Hence future work involves trying to run them successfully on the hardware. Some rules in the third experiment could also be changed in order to fasten up the data packet transmission in the network: diverting *H11*'s packets to a new link between *S1* and *S2* in case of heavy traffic instead of reducing the ingress packet rate at *S1*.

6. Conclusion

As the exciting field of SDNs develops, and data plane programming becomes increasingly popular for optimizing networks, it is imperative to properly understand each step of the process and the influence programming language and target device choices have on network systems. The experiments in this thesis offer insight into the large differences that can arise from such design decisions. They show that programming the data plane gives the user more flexibility and allows for innovative conditional forwarding at switch level. The usefulness of P4 in terms of target and protocol independency are revealed and some of its advantages over OpenFlow are listed. Some conditional forwarding ideas that can optimize networks, provide end-to-end delay (deadline) guarantees and ensure network connectivity in the event of high traffic are presented. These ideas were implemented in P4 and tested on mininet first and then on the SDNet and SUME simulations provided by the NetFPGA-SUME board.

References

[1] Sahri, N. M., and Koji Okamura. "Fast failover mechanism for software defined networking: Openflow based." *Proceedings of The Ninth International Conference on Future Internet Technologies*. 2014.

[2] Kreutz, Diego, et al. "Software-defined networking: A comprehensive survey." *Proceedings of the IEEE* 103.1 (2014): 14-76.

[3] Kim, Hyojoon, and Nick Feamster. "Improving network management with software defined networking." *IEEE Communications Magazine* 51.2 (2013): 114-119.

[4] Nunes, Bruno Astuto A., et al. "A survey of software-defined networking: Past, present, and future of programmable networks." *IEEE Communications Surveys & Tutorials* 16.3 (2014): 1617-1634.

[5] Casado, Martin, Nate Foster, and Arjun Guha. "Abstractions for software-defined networks." *Communications of the ACM* 57.10 (2014): 86-95.

[6] Benzekki, Kamal, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. "Software-defined networking (SDN): a survey." *Security and communication networks* 9.18 (2016): 5803-5833.

[7] Farhady, Hamid, HyunYong Lee, and Akihiro Nakao. "Software-defined networking: A survey." *Computer Networks* 81 (2015): 79-95.

[8] NetFPGA-SUME Virtex-7 FPGA Development Board, Digilent Inc., 2020. Available at: https://store.digilentinc.com/netfpga-sume-virtex-7-fpga-development-board/. Accessed April 2020.

[9] Zilberman, Noa, et al. "NetFPGA SUME: Toward 100 Gbps as research commodity." *IEEE Micro* 34.5 (2014): 32-41.

[10] Lockwood, John W., et al. "NetFPGA--an open platform for gigabit-rate network switching and routing." *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. IEEE, 2007.

[11] Kaur, Karamjeet, Japinder Singh, and Navtej Singh Ghumman. "Mininet as software defined networking testing platform." *International Conference on Communication, Computing & Systems (ICCCS)*. 2014.

[12] Tourrilhes, Jean, et al. "The evolution of SDN and OpenFlow: a standards perspective." *IEEE Computer Society* 47.11 (2014): 22-29.

[13] Monsanto, Christopher, et al. "Composing software defined networks." *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013.

[14] McKeown, Nick, et al. "OpenFlow: enabling innovation in campus networks." *ACM SIGCOMM Computer Communication Review* 38.2 (2008): 69-74.

[15] Bosshart, Pat, et al. "P4: Programming protocol-independent packet processors." *ACM SIGCOMM Computer Communication Review* 44.3 (2014): 87-95.

[16] Thacker, Charles P. "Improving the future by examining the past: ACM Turing award lecture." *ACM SIGARCH Computer Architecture News*. Vol. 38. No. 3. ACM, 2010.

[17] Chu, Cing-Yu, et al. "Congestion-aware single link failure recovery in hybrid SDN networks." *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2015.

[18] Beheshti, Neda, and Ying Zhang. "Fast failover for control traffic in software-defined networks." *2012 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2012.

[19] Wang, Yang, and Wei Shi. "Budget-driven scheduling algorithms for batches of MapReduce jobs in heterogeneous clouds." *IEEE Transactions on Cloud Computing* 2.3 (2014): 306-319.

[20] Feist, Tom. "Vivado design suite." *White Paper* 5 (2012): 30.

[21] Yazdinejad, Abbas, Ali Bohlooli, and Kamal Jamshidi. "P4 to sdnet: Automatic generation of an efficient protocol-independent packet parser on reconfigurable hardware." *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2018.

[22] Acceptance Test Project, NetFPGA, 2020. Available at: https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/Acceptance-Test-Project. Accessed April 2020.

[23] Tutorials, P4 Language, 2020. Available at: https://github.com/p4lang/tutorials. Accessed April 2020.