UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

A state consistency framework for programmable network data planes

Hugo Alexandre Mendes Garcia

Mestrado em Engenharia Informática
Especialização em Arquitetura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Naercio David Pedro Magaia

2021

Resumo

A tecnologia de Rede Definida por *Software* (SDN – Software-Defined Networking) é um método de gestão de rede que permite um planeamento dinâmico e programaticamente eficiente da mesma, a fim de melhorar o seu desempenho e monitorização. SDN tenta centralizar o cérebro da rede num único dispositivo, desassociando o processo de encaminhamento de pacotes (*Data Plane*), do processo de roteamento (*Control Plane*). A constituição do *control plane* consiste em um ou mais controladores, nomeadamente, aplicações que atuem de forma estratégica, que são referenciadas como sendo o cérebro da rede SDN, onde toda a inteligência é agregada. Esta dissertação foca-se em redes programáveis, o que significa que o programador da rede, pode controlar a forma como um dado dispositivo controla o fluxo de pacotes, por meio de *software,* sendo executado de forma independentemente do *hardware* da rede.

Dado que a cada dia que passa as SDNs tornam-se cada vez mais importantes, permitindo aos programadores optar por diferentes métodos e formas para desenhar, implementar, e operacionalizar as redes com que diariamente se deparam. Assim, uma ferramenta que permita garantir uma comunicação fidedigna entre os dispositivos de rede, assume um papel cada vez mais relevante. Existe também o objetivo desejável de manter um estado global partilhado entre os dispositivos que constituem a rede. A crescente necessidade de garantir a segurança das redes, que cada vez mais são o veículo de transmissão da informação neste mundo globalizado e por conseguinte a proteção dos dados que nelas circulam, provocam de uma forma global um crescente interesse na atividade dos atacantes. Cada vez mais utilizam ferramentas e conceitos mais sofisticados tentando quebrar ou contornar as regras de segurança na tentativa de obter gratificação pessoal, ou porque, ao existirem melhorias numa nova área, é provável que com a sua introdução, estas insiram novas vulnerabilidades que possam ser exploradas por esses atacantes. Além disso, existe o fato que, no campo das SDNs, muitos propuseram mecanismos de comunicação entre os dispositivos do plano de dados, ou seja, *switches*, e mesmo entre estes.

No entanto, dada a inexistência de uma maneira globalizada para realizar esta comunicação, este trabalho visa propor uma estrutura de consistência de estado que aproveite a abstração de uma máquina de estados, usando atualizações de rede

consistentes, entre *switches* adjacentes, por meio da clonagem de pacotes, oferecida pela linguagem de programação P4 e por *piggybacking* de pacotes que já se encontravam a circulação na rede. Também é usada uma média movel exponencial ponderada, onde determinada condição é alcançada, e de seguida acionamos o recurso P4, anteriormente mencionado, i.e., acionamos a clonagem de pacotes, garantindo assim que todos os pacotes chegam ao seu destino. Além disso, usamos ferramentas para realizar as verificações bit a bit, considerando que a linguagem P4, ainda carece de uma forma para aceder a partes específicas do pacote que circulam na rede.

Também utilizamos uma estrutura capaz de armazenar qualquer tipo de informação, i.e., os registos, nos quais decidimos armazenar todos os dados por nós considerados relevantes e a serem utilizados durante a execução. Concretamente, guardamos toda a informação relativa, tanto aos endereços de origem e ao endereço de destino do pacote, bem como os portos de entrada e os portos de saída do *switch*.

Foi também ponderado o uso de outra linguagem de programação em vez do P4, porém esta outra linguagem, o Domino, não nos oferece a característica principal que é possibilitar a criação novos cabeçalhos, pois este era o objetivo inicial quando a linguagem foi procurada, acabando por ser posto de parte o Domino e a escolha recaiu sobre o P4, como a linguagem de programação para a implementação da ferramenta.

Tendo em conta as ferramentas de simulação e de emulação dos casos de teste, á frente apresentados, foram estudadas outras ferramentas, como o NS-3 e o Estinet.

Pelo conhecimento prévio e experiência na utilização do Mininet, para simular redes de casos-tipo e atendendo que uma das características chave do Mininet é similar á do NS-3 e ambas têm um mecanismo de virtualização leve, optamos pelo Mininet, pois o Mininet possibilita-nos o uso direto do OpenFlow, sem a necessidade de adicionar extras, ao contrário das outras duas ferramentas atrás referidas, que necessitam de adicionar esses extras, para suportar o uso do OpenFlow.

Para a implementação da ferramenta inicialmente, consideramos a utilização um método de pacotes de tipo *probe,* e depois atentamos uma metodologia baseada em um novo cabeçalho, para implementar a ferramenta ao invés da utilizada, i.e., o *piggybacking*.

Primeiramente, atentamos utilizar pacotes do tipo *probe* que seriam gerados pelos *switches*, e conforme acontecessem alterações na rede, e.g., aumentar ou diminuir a quantidade de dispositivos de rede, envisionamos que os pacotes seriam enviados pelo

switch e eles continham informação de modo aos *switches* restantes, poderem ter conhecimento dos eventos.

A outra metodologia que atentamos consistia em criar um novo cabeçalho no qual colocaríamos toda a informação que iria ser necessária para o correto funcionamento da ferramenta proposta. Contudo, e após exaustivas tentativas de testes para concretizar a sua implementação, esta apresentou demasiados problemas. Deparamo-nos com o problema fulcral que era saber onde colocar nova informação dentro dos componentes que formam a pilha de cabeçalhos da internet, uma vez que esta, é de uso globalizado e quebrar ou tentar alterar a sua estrutura, significaria que o pacote seria puramente descartado sem a possibilidade de atentar qualquer intervenção nele.

Para avaliar a solução proposta, coletamos uma ampla quantidade de informações através de um programa corrido nos emissores e bem como nos recetores dos pacotes e para isso usamos dois cenários de teste distintos.

No primeiro cenário, usando uma rede com um pequeno número de dispositivos, mais concretamente, 4 *switches* e 3 *hosts*, coletamos o número de pacotes recebidos no destino e o número de pacotes retransmitidos na origem.

No segundo cenário, usamos uma rede com um número maior de dispositivos em comparação com o primeiro cenário, ou seja, desta vez a rede seria composta por 8 *switches* e 4 *hosts*, e nela coletamos o mesmo tipo de dados, isto é, a quantidade de pacotes recebidos no destino, após o *host* ter sido reajustado de modo a fazer uso desta nova rede. Também coletamos o número de pacotes retransmitidos na origem.

Após uma análise extensa dos resultados obtidos, foi possível verificar que apesar de falhas nos links superiores a 70%, ainda assim conseguimos garantir que mais de 95% dos pacotes chegam com sucesso ao seu destino.

Apresentando os resultados com mais detalhe, no primeiro cenário, relativamente ao número de pacotes recebidos no destino, em 75% das execuções realizadas, a nossa solução, assegurou que perto de 100 pacotes chegassem ao seu destino em cada execução. Em referência ao segundo cenário, em 79% das execuções, a nossa solução conseguiu entregar perto de 99 pacotes por execução e que estes alcançaram o seu destino.

No que diz respeito ao número de pacotes retransmitidos, nestes encontramos três tipos de retransmissões. As retransmissões normais, que como o nome sugere, são aquelas em que nada de extraordinário acontece, ou seja, os pacotes são simplesmente clonados e são novamente enviados por uma rota alternativa para seu destino.

Identificamos também, retransmissões com pacotes espúrios, estas retransmissões são verdadeiramente aleatórias e repetem-se em 1 ou 2 pacotes em todos os enviados.

Por último, existem as retransmissões que ocorreram e que foram afetadas pelo problema de congestionamento de pacotes ou ciclos de CPU do PC quando este se encontrava mais congestionado ou ocupado. Nestas circunstâncias os pacotes foram simplesmente marcados, para existir a possibilidade de distingui-los dos restantes.

Em referência ao número de pacotes retransmitidos, no primeiro cenário a nossa solução obteve em 75% das execuções, perto de 98 pacotes retransmitidos por cada uma dessas execuções.

No segundo cenário, houve uma consistência nos tipos de retransmissões, pois podemos constatar que se mantiveram os mesmos tipos, todavia foi-nos também possível verificar uma diminuição considerável dos tipos de retransmissão com pacotes espúrios e nas retransmissões afetadas pelo problema de congestionamento de pacotes ou ciclos de CPU do PC quando este se encontrava mais congestionado ou ocupado. Pelo que neste segundo senário foi notável um aumento do número de pacotes retransmitidos.

Para concluir, este trabalho procurou resolver o problema de inconsistência de informação presente nos dispositivos de rede ou mesmo entre eles, i.e, *switches*, para isso propomos uma ferramenta que garanta consistência de estado para *data planes* programáveis e que aproveite a clonagem de pacotes através da aplicação de uma condição ligada á média movel exponencial ponderada e *piggybacking* usando P4.

É notável que mesmo com falhas nos links superiores a 70%, a nossa solução mesmo assim, consegue garantir que 95% dos pacotes que foram enviados, chegam aos seus destinos.

Focando-nos no trabalho futuro, poderia ser utilizado o conceito de *tuplos* para tornar o código mais simples e tornar o acesso a certas partes do pacote ainda mais simples.

Para finalizar esta dissertação, contribui uma nova ferramenta que garante consistência de estado para data planes programáveis que nos garante que os pacotes que circulam dentro de uma rede não são perdidos utilizando a clonagem de pacotes através da utilização de uma média movel, também possibilita a utilização de *piggybacking* através da criação de espaço no pacote que circulam na rede.

Por fim, baseado nesta dissertação foi submetido um artigo para uma conferência internacional.

Palavras-chave: SDN, Consistência de estado, Framework, Redes Programáveis

Abstract

The Software-Defined Networking (SDN) technology is a method of network management that allows dynamic, programmatically efficient network planning to improve its performance and monitoring. This dissertation focuses on programmable networks, which means that the network programmer can control how the network devices control packet flows via software that runs independently from network hardware.

Given that SDN at each passing day becomes more and more prominent, a framework that can ensure reliable communication and a global state among devices become more and more important. There is also the desirable goal of being able to maintain a global shared state among all network devices. Also, there is the fact that in the field of SDNs, many have proposed communication mechanisms among data plane devices, i.e., switches, and between the latter and the controller. However, given the inexistence of a widespread manner to do so, this work aims at proposing a state consistency framework that leverages on a state machine abstraction using consistent network updates among adjacent switches through packet cloning offered by the P4 programming language and packet piggybacking. We also use a moving average that when a condition is met, such P4 feature is triggered, hence ensuring that all packets arrive at their destination. Also, we use tools to perform bitwise checks, considering that the P4 language lacks ways to access specific parts of a packet.

We also use a structure that is capable of storing any type of information, i.e., registers, in which we decide to store the data to be used.

To evaluate the proposed solution, we collected a broad amount of information using two scenarios.

In the first scenario using a network with a small number of devices, we collect the number of packets received at the destination and the number of packets retransmitted at the sender.

In the second scenario this time, using a network with a higher number of devices compared to the first scenario, we collect the same type of data, that is, the number of packets received at the destination after it has been readjusted to mirror this new network structure. We also collect the number of packets retransmitted at the sender.

After an extensive analysis of the results obtained, it was possible to verify that despite link failures higher than 70%, we still managed to have more than 95% of packets arriving successfully.

Keywords: SDN, State consistency, Programmable networks, Framework

List of Figures

Figure 1: An OpenFlow Architecture Example (Extracted From:[32].) 19
Figure 2: Implementation Example Using TCP in FAST Data Plane (Extracted From: [6]) 25
Figure 3: A P4 Pipeline Example (Extracted from: [23].) 28
Figure 4: A Topology Example Using a GUI for Mininet (MiniEdit) (Extracted From: [24].) 33
Figure 5 - State Machine Example Using our Parser 36
Figure 6 - A Timestamp Example (Extracted From: [27]) 37
Figure 7 - A SACK Example (Extracted From: [27]). 38
Figure 8 - V1 Metadata that is Used 39
Figure 9 – Example of a Cloned Packet 39
Figure 10 – Example of a Normal Packet 40
Figure 11 - An alternative path for packets to go from H1 to H2 41
Figure 12 - An example scenario 43
Figure 13 - Example of TCP from a pcap file in Wireshark 45
Figure 14 - Flow from H1 to H2 – F1 46
Figure 15 - Flow from H3 to H1 – F2 46
Figure 16 - Distribution of received packets for the first scenario. 57
Figure 17- Average number of received packets to H2 for the first scenario 58
Figure 18 - Distribution of retransmitted packets for the first scenario. 60
Figure 19 - Average number of retransmitted packets to H2 for the first scenario 61
Figure 20 - Second Test Scenario Network 62
Figure 21 - Second Scenario Network with Flows 63
Figure 22 - Distribution of received packets for the second scenario. 64
Figure 23 - Average number of received packets at H4 for the second scenario.. 65
Figure 24 - Distribution of Type of Retransmissions in the Second Scenario 66
Figure 25 - Average number of received packets at H4 for the second scenario.. 67

List of Tables

Table 1 - Table for the Number of Retransmitted packets from the Switch S1 during the first set of results 70

Table 2 - Totals for Retransmitted Packets during the first set of results 71

Table 3 – Table for the Number of Received Packets at the end host H2 72

Table 4 - Totals for Received Packets during the first set of results..................... 73

Table 5 . Table for the Number of Retransmitted packets from the Switch S1 during the second set of results 74

Table 6 - Totals for Rertransmitted Packets during the second set of results 75

Table 7 - Table for the Number of Received Packets at the end host H4 76

Table 8 - Totals for Received Packets during the second set of results 77

Content

Chapter 1 Introduction 13

1.1 Motivation 13
1.2 Problem 14
1.3 Objectives 15
1.4 Contributions 15
1.5 Document structure 16

Chapter 2 Background and Related Work 17

2.1 OpenFlow 18
2.2 Consistency 20
2.3 State Machine Abstraction 23
2.3.1 Flow-level State Transitions (FAST) 23
2.3.2 OpenState 25
2.3.3 FlowBlaze 26
2.3.4 Discussion 27
2.4 Programming Languages 27
2.4.1 P4 27
2.4.2 Domino 29
2.4.1 Discussion 30
2.5 Simulation and Emulation Tools 31
2.5.1 NS-3 31
2.5.2 Mininet 32
2.5.3 EstiNet 33
2.5.4 Discussion 34

Chapter 3 The proposed solution 35

3.1 Solution overview 35

3.2 Implementation 42
3.2.1 Network Configuration 42
3.2.2 Registers and Metadata 46
3.2.3 Actions 48
3.2.4 Tables 49
3.2.5 The workflow of the program 51
3.2.6 Difficulties 55

Chapter 4 Evaluation of the Proposed Solution 56

4.1 First Test Scenario 56
4.1.1 The number of received packets 57
4.1.2 The number of retransmitted packets 59
4.2 Second Test Scenario 61
4.2.1 The number of received packets 64
4.2.2 The number of retransmitted packets 66

Chapter 5 Conclusion and Future Work 68

5.1 Conclusion 68
5.2 Future Work 68
5.2.1 Registers 68
5.2.2 Offloading and Separation 68

Appendix 70

Bibliography 78

Chapter 1

Introduction

1.1 Motivation

When the Internet was still in development, hence, closed from the general public, the only ones that had access to it were its developers. In these restricted networks, researchers would conduct experiments on new protocols and if they were considered interesting by the scientific community, they would get funding and present them to the Internet Engineering Task Force (IETF) to be standardized. However, this process ultimately frustrated many researchers since it was very slow.

In response to this problem, some researchers tried a different approach and proposed a new paradigm called *Active Networking* [1]. This, at the time, was a very radical approach as it went against many principles behind networking that advocated the simplicity of networking. At its core, *Active Networking* envisioned a network API where the network resources would be exposed on very simple entities and support new custom functionalities for packet processing. There were two programming models: (i) the capsule model, where they envisioned new data-plane functionalities in which packets carried the code to be executed and they also used caching to improve the efficiency of code distribution, and (ii) the programmable switch/router model, where it was planned to give decisions to the network operator.

As this was in the primordial phases of the Internet, the foundations for future work had to be done and they were the following: (i) the introduction of programmable functions in the network, (ii) enabling the virtualization of the network, and (iii) the ability to demultiplex, and last but not least, (iv) the idea to unify the architecture for middleboxes.

All of these would eventually lead to the separation of the planes, i.e., the control and data planes, which enabled the idea of Software Defined Networking (SDN) [2] to appear. The control plane relates to all the functions and processes that determine which path to use. These are more known as routing protocols, some of them are spanning tree protocol [3], Consensus routing [4], which will be covered later on in this document

meanwhile, the data plane denotes all the functions and processes that forward packets/frames from one interface to another. These can be numerous devices such as switches, routers, firewalls amongst others. Concretely, it would be the increase of the Internet traffic given that the latter was now open to the general public, and the fact that anyone having access to it could use it. As a result, the volume of data increased and there was a need to guarantee performance and reliability. However, conventional switches and routers were conceived with both planes together in mind, and, therefore, unnecessarily complicating several tasks such as debugging problems or controlling routing behavior.

Consequently, there was a need to improve and innovate to tackle new trends whether they were in usage or in demand. This was also the rationale underneath the appearance of SDN. That is, the existence of an open interface between the control and data planes would allow logically centralized control of the network.

1.2 Problem

The problem we are mainly trying to address here is the fact that there can be inconsistencies in the existing information in the data plane network devices, i.e., switches, or even among themselves. Specifically, the real challenge is: (i) when and how to update the switches?, (ii) with what messages, that is, by simply creating more overhead in the packets, or by sending more packets to the switches? By taking into consideration that the time it takes to process a packet in hardware is negligible [5], the addition of more packets into the network might be a good option. However, there is still the issue of questions (i) and (ii).

Let us take a concrete example to give a better understanding of the problem at hand. Let us say that there is an attacker that can both inject new packets into the network and take original packets that were sent by trustworthy sources and alter them for their gain. And what could be the gain? In this case, it could be to perform a Denial of Service (DoS) attack, hence making an operational switch seem to the network that it is not [6]. There are several ways to do this, but one way would be to send a packet that was malformed to the point that when it reached the switch, any action that would be taken, would cause the device to crash.

Another exacerbating factor of this type of attack is the fact that between adjacent switches there is no form of authentication. Thus, leading back to the original problem,

that is, "how to avoid the original problems?", and when trying to prevent it, "which information to send to the switches?", "who should have a consistent view of the network?", so that the switches can make appropriate choices.

1.3 Objectives

The main objective of this dissertation is to develop a software-based state consistency framework for data plane networking. Specifically, we aim at porting novel state machine abstraction to P4 programming language and using key features such as packet cloning to ensure that all packets arrive at their destination and packet piggybacking to guarantee a consistent network view.

1.4 Contributions

In this report, a state consistency framework leveraging packet cloning and piggybacking for programmable network data planes is proposed. It was implemented using the novel Programming Protocol independent Packet Processor (P4) [7] programming language that runs on switches. We use a moving average that triggers packet cloning, and every packet that passes through the switch gets added necessary information for the framework to work. The proposed framework ensures that packets arrive at their destination and guarantees that a consistent state is achieved via packet piggybacking. To the best of our knowledge, this is the first work implementing a network state consistency framework that ensures a high packet delivery rate in the presence of high link error rates in P4.

The contributions of this report are summarized as follows:

- A state consistency framework for programmable network data planes;
- Packets are salvaged through packet cloning. A full copy of the packet is done, except for the metadata that is inerrant to that specific execution. The clones are sent via another route;
- Packet piggybacking utilizing the Transport Control Protocol (TCP) headers. It is mainly used to store the information exchanged between switches.
    - A state machine abstraction for P4;

An article based upon this work was submitted to an international conference.

1.5 Document structure

The organization of the documentation is as follows:

Firstly, in chapter 2, the SDN topic is approached in depth. We begin by defining the term Consistency which is commonly used in areas such as Distributed Systems and/or Computer Networking. Afterward, we present the types of programming languages used in this field. Lastly, we present simulation and/or emulation tools. Please notice that at the end of each section we present a small discussion, hence motivating our choices.

In chapter 3, we present the design and implementation of the proposed solution. To do so, here we begin by explaining why we choose which model to use, the V1 model [8], and only then do we a deep explanation of the structure of the solution itself. Here we approach the most important parts of the solution such as actions, tables, registers, and metadata.

In chapter 4, the topic of evaluating our proposed solution is handled. For that, we give two different scenarios and thoroughly analyze them. In each scenario, we use different network topologies to simulate different environments for the solution to work on.

To check how well it performs, we gather several types of data, the two that could better show the solution's performance and that were used are:

- The number of retransmitted packets;
- The number of received packets at the receiving host.

In chapter 5, we give our conclusion for this report, based upon the results that were shown in chapter 4, and then present some future work that needs to be done to make sure the solution can adapt to new situations.

Chapter 2

Background and Related Work

In the beginning, the major focus of SDN was: (i) the open interface linking the data and control planes, and (ii) distributed state management.

Regarding the first, at the time there were some proposals among researchers, and the most notorious one was ForCES [9], which proposed an open interface to the data plane that enabled innovation in the control plane. Some routers used this new protocol (e.g., ForCES) to let a separate controller establish new forwarding-table entries in the data plane and thus enabling the removal of control from routers.

Regarding the second, and more importantly, the focus of this dissertation, it was logically centralized on controllers and had lots of problems maintaining the distributed state. Among the many challenges was the fact that the controller had to be replicated to handle its failures. However, the replication itself could introduce inconsistencies throughout the replicas [1].

In particular, the authors of FAST [10] found three tools that helped the controller by reducing its involvement with dynamic applications, concretely DevoFlow [11], which is a variation of the OpenFlow model that severs the coupling between control and global visibility and maintains a useful amount of visibility without imposing unnecessary costs. Regarding what it introduces to help the controller, it reduces the controller overhead by introducing rule cloning and measurement triggers. They also investigated OpenFlow version 1.3 and found that it supported rate-limiting by allowing switches to track flow rates and tag/drop excess traffic without the controller's involvement.

Finally, they investigated Open vSwitch [12], a multilayer virtual switch. It was designed to allow massive network automation, while still encouraging standard management interfaces and protocols. In concrete, what they added towards helping the controller was the fact that it adopts the learn action for software switches that can install new rules when traffic matches an old rule.

2.1 OpenFlow

The OpenFlow protocol was the first attempt to unify a network and how they operate. At the time of the release of OpenFlow version 1.0 [11], there were a lot of different types of switches and routers with their specific language, and as such inter-network communication was very difficult, and this was where OpenFlow came in and contributed in a big way.

First of all, OpenFlow is a protocol that could be used in any switch or router and the OpenFlow developers also released their switches and OpenFlow gave an API where the network devices could be remotely controlled through it. OpenFlow exploited the fact that at the time most OpenFlow-enabled switches contained flow tables that were built at line rate and were used for various situations, such as Network Address Translation (NAT), Firewalls, Quality of Service (QoS), etc. [11].

However, these flow tables need to be managed somehow, and because most of them used different protocols, OpenFlow gave an open protocol that could program the different network devices (e.g., switches and routers). At the time the flow table had three primary fields: (i) the packet header that defined the flow, (ii) an action that defined how the packets should be processed, and finally (iii) the statistics field, keeping a record of the number of packets and bytes for each flow, as well as the time since the last packet, arrive from the flow.

When OpenFlow was released to the public, their dedicated switches had three basic actions when a packet was received, namely, the switch need to be able to: (i) forward the packet to a given port and hence allowing packets to be rerouted throughout the network; (ii) to encapsulate and forward the packet to the controller, this was mostly used for the first packet of a new flow, so the controller can decide whether or not to add a new entry to the flow table; (iii) to drop the packet but the most prominent one being to drop packets that are being sent to cause a Denial of Service (DoS); and finally, (iv) to forward packets through the switch's normal processing pipeline [11].

Initially, OpenFlow version 1.0 only supported a total of four protocols that were Virtual Local Area Network (VLAN), Ethernet, Internet Protocol (IP), and Transmission Control Protocol (TCP) [11]. To better illustrate the structure of OpenFlow version 1.0 we introduce figure 1.

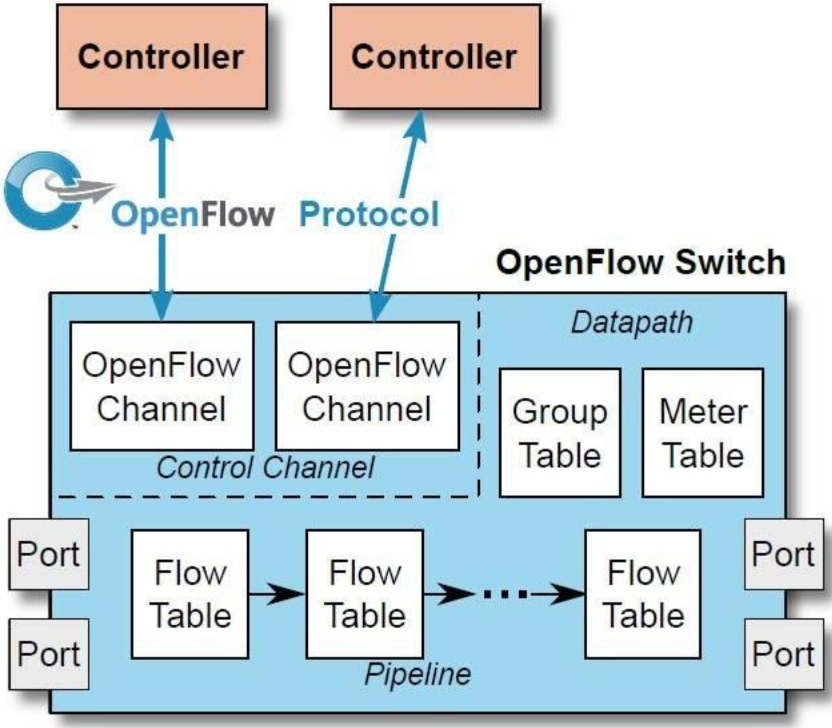


Figure 1: An OpenFlow Architecture Example (Extracted From:[32].)

OpenFlow is the basis of SDN as a whole and therefore it is necessary to properly introduce it as a whole. However, here we mainly cover some of the major contributions the newer versions gave, and get even more in-depth with the newer version of OpenFlow 1.5.0 [13].

In OpenFlow 1.0, the option for groups and multiple tables was added. In version 1.1, support for an extensible match and for the controller to be able to change the roles was added.

Later on, in version 1.2, support to enable refactoring capabilities in a negotiation that allowed for more flexible table miss support was increased, and it was also included per-flow meters, auxiliary connections, and tunnel-ID metadata, among many others.

Further, in version 1.3, support for more extensible wire protocols was added, similarly, to flow monitoring, synchronized tables, a vacancy of events and bundles.

Lastly, in version 1.5.0, lots of new features such as egress tables, packet-aware pipelines, extensible flow entry statistics, and meter actions were added [13].

In version 1.5.0, the addition of tables for both ingress and egress enabled the possibility for match action to occur on both of the ends of the pipeline allowing more complex functions to be done. The packet pipelines are simply a pipeline that knows what

type of packets are in it and any other type of packet will not be able to go through it. This is done via header, specifically the *ns* type. Flow duration, flow count, packet count, and byte count statistics were added to the flow statistics. And, the possibility for the programmer to use them in action (if packet count = a number, do an action) was added [13].

2.2 Consistency

With consistency being the main goal of this dissertation, as expected it could not go without being approached. With that in mind, a brief explanation of what consistency is, whether it is in distributed systems or the SDN paradigm is given. Firstly, regarding the former, *consistency* means that when a system, let us call it A, sets a parameter, let us call it X, to the desired value, and later on another system, let us call it B, gets the value, and if no other changes are made to it in the meantime, it will return the value set by A. If a distributed system can ensure that value is still X under most circumstances it can be said that the system is consistent and that it offers a consistent service.

However, regarding the SDN paradigm, consistency is similar in the sense that a system must offer a consistent service but at their core, they must ensure very different things, such as when a switch requests information related to a flow or where a specific packet must go, the controller must always know where to send it. The main focus is precisely this, the controller must maintain the state of all the flows, rather than maintaining track of the most important flow, or even the heavy hitter flows. The latter is rather difficult because there can be inconsistencies between the information present in the controller and the real situation on the network.

Regarding those inconsistencies, in the past, there have been attempts to standardize the way to handle them. More specifically, the authors of [14] presented a *notion of correctness*, that has to do with the idea mentioned in the above paragraphs. And, to do so, while they were trying to demonstrate possible database models, they found potential studies related to the subject at hand.

However as is intrinsic to distributed systems regardless of their type, they all have to have into account the CAP (Consistency, Availability, and Network Partitioning) theorem [15]. Such theorem refers to the impossibility of being able to offer all three properties if the system is under some form of stress. Specifically, let us consider that there is a switch that goes down, “what should be done?” “stop everything in the system

and guarantee consistency?", or should the system continue running, and when the switch reboots it tries to update itself. Fortunately, the second approach is the favored one and has had a lot of work put into it. There is also the case when the network is split (i.e., partitioning happens) into two parts, let us say they are uneven to facilitate the decision (because if they were to be even there would be no right decision, meaning that since both parts contain the same number of nodes, both contain a majority of the nodes, hence both can commit their results and overlap the other one's results and they would be stuck here forever).

"What should be done?" Stop responding or continue handling requests? The answer is not straightforward and might diverge from system to system. For instance, there is the possibility of stopping the half that has the least amount of request handlers and when the partition ends, they simply redo the requests done by the other half that had been running. Or the other less used (only used in critical systems, such as banks and government-related systems), to stop the system altogether and when the partition ends, recommence regular operations.

Also in [14] they also proposed many ways to handle consistency such as logging the transactions, or utilizing consensus to decide which operation to do, so that every part of the system would have the same state. But above all, *atomic commitment* is nowadays one of the most important properties. Specifically, if a transaction starts, it either finishes and every part of the system agrees that it finished, or in case of a single failure to agree, the transaction is aborted and tried again at a later date, also known as the *all or nothing*.

Moving from a more general consistency to a particular approach, the authors of [16] talk a bit about how to guarantee consistency inside a network and provide some ways to do so, such as per packet and per-flow consistency.

One reason that network updates are difficult to get right is that they are a form of concurrent programming. And therefore, doing them as per-packet consistent updates reduces the number of settings a programmer must consider to just two. For every packet, the programmer either considers the packet to have passed through the whole network before the update occurs or not. And consequently, it is easy to think that per-packet consistent updates as "atomic updates", but they are not the same. In other words, per-packet consistency points out that for a given packet, the traces generated during an update come from the old or the new configuration, but not a mixture of the two [16].

More formally, the authors introduce a new definition called the *relation of packets*. They state that every trace generated during the update has to be equivalent to a trace made by either the initial or final configuration. A per-packet mechanism may perform internal accounting by stamping version tags without violating our technical requirements on the correctness of the mechanism.

While the above might be simple and effective, is not always nearly enough because some applications have special needs, and to handle those needs per-flow consistency is necessary. In other words, per-flow consistency can be utilized as an abstraction called *per-flow abstraction*. To see its real need, we can consider a network that has several servers and only has a single switch. The issue here lies with the IP addresses. Initially, let us first consider only two servers A and B and their IP addresses would start with 0 and 1, respectively. However, if other servers are added a rebalance would be needed, and as such new IP addresses would also be needed.

Putting more simply, all packets in existing flows must go to the same server, where a flow is a series of packets with related header fields, entering the network at the same port, and not separated by more than n seconds, where n depends on the application at hand [16].

It guarantees that all packets in the same flow are treated all the same. Formally, the *per-flow abstraction* preserves all path properties, as well as all properties that can be expressed in terms of the paths traversed by sets of packets belonging to the same flow [16]. While the above speak mostly of how to handle consistency in a network, *consensus routing* [4] is a consistency-first approach that separates safety and liveness using two logically distinct modes of packet delivery. The stable mode where a route is adopted only after all dependent routers have agreed upon it. And the transient mode that heuristically forwards the small fraction of packets that encounter failed links.

In more detail, the stable mode does so by giving each router/switch a log that will update each epoch, for the entire system a distributed snapshot exists. Since the network is not static, this means that new routers/switches can join the network, not only that but they can also leave it. That is, the mode also has to view change [17], which is very important, to be able to maintain a consistent view of the system at any given epoch.

The transient mode is more focused on handling the failures of the network itself. It does so by simply *deflecting* the packet to a neighbor router/switch, so that it may try a new route to its destination. This neighbor is selected using a pre-computed backup route

that can be done by Resilient Border Gateway Protocol (R-BGP) [18], which will allow them to announce their backup routes to each other.

To finish, in practice this is how the protocol works. First, a distributed coordination algorithm runs to ensure that a route is accepted by every part of the system only after all dependent routers have agreed upon a globally consistent view of the global state.

Afterward, packets are forwarded using one of two logically distinct modes. The stable mode is only used if the computed routes using the coordination algorithm are consistent, and or the transient mode when a stable route is not available. Note for a route to be considered consistent, if router A accepts a route to a destination via another router B, then B adopts the corresponding suffix as its route to the destination.

2.3 State Machine Abstraction

In this section, we survey papers about state machine abstractions. A state machine abstraction is a state machine that runs on a set of states that can be arbitrary data structures. These structures must have at least one member in them.

2.3.1 Flow-level State Transitions (FAST)

In FAST [10], there are two types of flow rules: the proactive and reactive approaches. The former being linked to the controller where it populates the rules in the switches ahead of time. However, it requires a priori knowledge of the events in the switches. And the latter being an approach that supports dynamic applications. However, it offers poor performance and as such, it was not used in the FAST primitive. Nevertheless, the authors state that this approach should be used as it supports a far bigger number of applications [10].

FAST is divided into three parts: (i) the abstraction that allows programmers to program the state machines; (ii) the FAST controller; (iii) the data plane. Regarding the abstraction, FAST implements a regular state machine that features the regular states with storing counters that represent the many states. If needed, the controller can store the state names and their variables to a bit string for easier representation of the values. The transitions and actions can only be triggered if their guard condition is verified. The latter is equal to OpenFlow version 1.3. Though in FAST, it is not allowed for a packet to be processed through more than one state machine. It also features a filter for the programmer to search through the state machine more easily. Finally, the instance

mapping is done via task definition, where it must be specified by mapping each packet to an instance to its current state.

Regarding the control plane, FAST is divided into 3 parts: (i) the FAST compiler; (ii) the Switch agent; and (iii) the state machine [10]. The compiler simply translates the state machine to regular code so that it can run on switches. Although to do so, it requires information to make them switch specific.

The switch agents are agents that can communicate with the state machine during its execution. It has three responsibilities: (i) must know the features of the switch, how to support the FAST abstraction in the data plane; (ii) perform part of the state machine implementation in the switch, i.e., it can fall back to a reactive approach; (iii) must be able to report local events to the global tasks at the controller. Nevertheless, the switch can still detect heavy hitters and it should be able to configure the switch to send those events to the controller.

To finish, FAST's data plane contains four tables and a state machine filter. An example extracted directly from the original paper is presented in figure 2. As the name suggests the state machine filter filters where the flows come from and are identical to the OpenFlow Tables, besides being implemented using Ternary content-addressable memory (TCAMs).

Next, is the state table, where are stored the n pairs (Index, State). When a packet has passed through the state machine filter, it was hashed and matched on the state table. If this was successful, and an index for it does not exist, then it is created. Afterward, a new pair is created (Packet, State).

When a match occurs, and the conditions are valid, the state will be updated, and the pair will be sent to the next and final table, i.e., the action table. Once again in the action table, if a match happens and the arriving packet matches the flow rules, the action will be executed. And the packet is sent to the network [10].

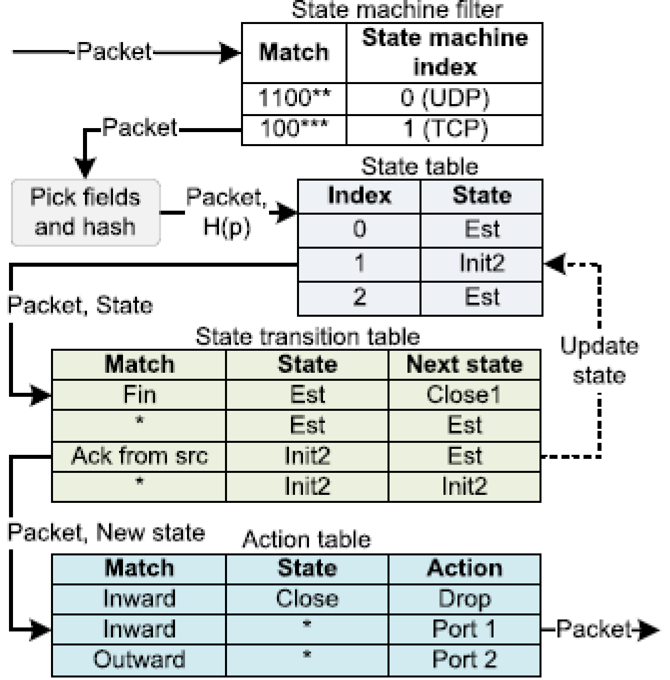


Figure 2: Implementation Example Using TCP in FAST Data Plane (Extracted From: [6])

2.3.2 OpenState

OpenState [19] is a tool that is based on the extended finite state machine (XFSM). XFSM is a Finite State Machine (FSM) that can do a few more actions. In particular, those can be, when a state transition is triggered (i.e., a match) a packet can be forwarded to a given port (i.e., an action), this is no less and no more than a match-action pair. It is also worth pointing out that the match which specifies an event not only depends on packet header information but also depends on the state that the state machine is on at the given moment.

The state machine can be modeled in an abstract form called the *mealy machine*. This machine uses a 4-tuple (*S, I, O, T*) where: *S* is a finite set of states; *I* is a finite set of input symbols, these can be events; *O* is a finite set of output symbols, these can be actions; and, $T : S * I \rightarrow S * O$ is a transition function which maps <state, event> pairs into <state, action> pairs [19].

Concerning how the state management is done in OpenState, two tables are used for this purpose: the State and XFSM tables. Specifically, OpenState features three main actions: (a) the state lookup, which consists of a query to the State Table using as key the packet's header field(s) which should be enough to identify the flow; (b) the XFSM

transition, where the state label is retrieved and added as metadata to the packet to perform a match on an XFSM table, which returns the associated action(s), and the label of the next state. Last but not least, (c) the State update, which consists of rewriting, or adding a new entry to the state table using the provided state label [19].

However, there is still the issue of flow identification, which was not solved. The authors state that the root cause of this issue was that, so far, they have not yet conceptually separated the identity of the flow to which a state is associated, from the actual position in the header field from which such an identity is retrieved.

Finally, the authors state that there is room for improvement by adding timeouts as in OpenFlow to a state table entry that is straightforward, and the API could be extended to permit the programmer to specify a different timeout for each state transition.

2.3.3 FlowBlaze

The authors of [20] proposed an open abstraction for building stateful packet processing functions in hardware, which is based on an Extended Finite State Machine (EFSM), which was previous introduced in [19]. They also added the concept of flow state, which is a sequence of packets from a certain source to a certain destination. ESFM is an extension of the FSM model by introducing several points, variables to describe the state, enabling functions on such variables to trigger transitions and update functions for the variable values.

The machine model simply extends the match-action table (MAT), which is a pipeline with a parser and a variable number of match-action blocks. The packet headers are handled in the pipeline and processed over the pipeline. In the pipeline, it is decided what are the forwarding actions for each of them.

FlowBlaze is based on the MAT abstraction and extends Reconfigurable Match Action Tables (RMT [21]) to perform stateful packet processing. Nevertheless, there a few protocols that are very similar to FlowBlaze, namely FAST and OpenState. Both of them define the FSM abstraction in their ways. However, they do not define the state access model, also do not deal with issues related to the integration of FSMs in an RMT machine model [20].

2.3.4 Discussion

This work aims at looking for solutions that can be efficient and simple so that they can be widespread. Therefore, the extended state machine is not worthy, despite being able to perform more complex actions. On the other hand, FAST might be a very good tool. Since this field is evolving at a very high rate of speed, some of its contents may be outdated, however, their novelty is still substantial and that is why we use FAST. OpenState and FlowBlaze handle the same subject, that is, the extended state machine. FAST introduces and tries to formalize the subject while OpenState and FlowBlaze try to improve and enhance it.

2.4 Programming Languages

In this section, it is provided a bit of background information regarding the state-of-the-art programming language that can be used in programmable networks. Afterward, a brief analysis of them is done to conclude this section, as well as explaining the reason for picking the specific language.

2.4.1 P4

Programming Protocol independent Packet Processor (P4) [7] is a programming language directed towards switches. The two most important factors that contributed to its success are: (i) similarly to OpenFlow, P4 uses the paradigm that there should be protocol independence, and (ii) switches should not be bound to specific protocols, and the controller should tell switches what to do, i.e., switches should receive information from the controller regarding what kind of parser should be used and which match-actions should be used in those specific packets.

Moreover, in other programming languages, the programmer does not need to know the underlying mechanics of the switch. Rather, in this case, it should be the compiler that does the most work.

It is rather simple how the forwarding model works in P4. Given an input (i.e., a packet), it is parsed, in contrast with OpenFlow where this parser is static or fixed. In P4, this can be programmed to support new headers that the programmer might add.

Afterward, the parsed packet is sent to the match-action tables also called the ingress phase (see Figure 3). These can be populated at runtime with pairs that are given by the programmer or are statically done a priori. Next is the egress phase, which is very similar to the previous one. However, here are only done changes to the packet that are related to where it must go. Finally, the packet is deparsed and sent to its destination [7].

An important aspect of P4 is the fact that between the different stages or phases there is a structure called the metadata that can carry additional data. The metadata is handled the same way as the headers.

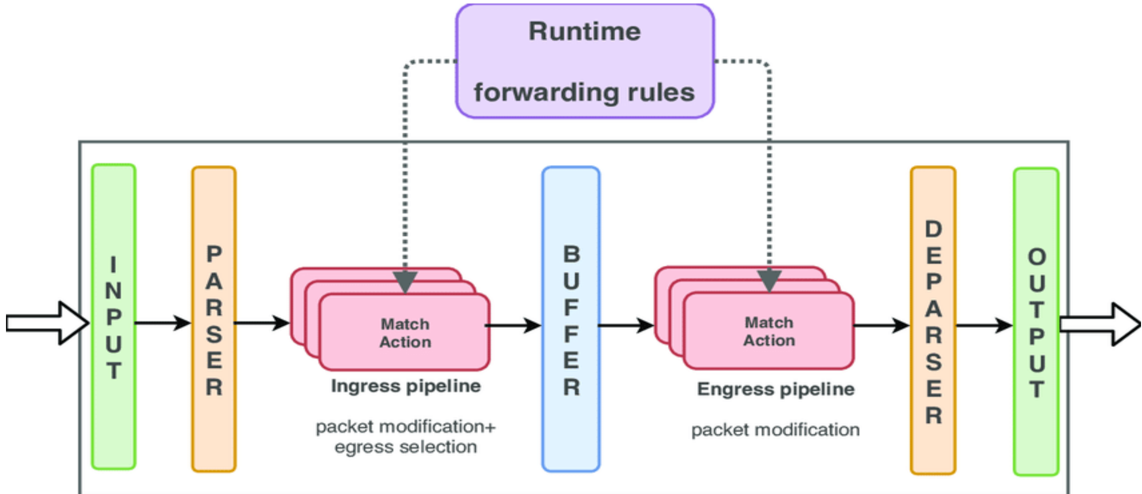


Figure 3: A P4 Pipeline Example (Extracted from: [23].)

Since in the text above we explain how P4 works theoretically, in practice, P4 is rather straightforward to work with. First, we must know what kind of protocols the target will be handling because those respective headers must be defined a priori. After the headers are defined, it must be established the way the packet headers are parsed and what happens with their content, e.g., where they are stored. Given that P4 is a C-like language, it allows for registers, arrays, and variables with a size that is fixed and defined by the programmer or if they are altered in some way [7].

After the tables are defined, it must be explicit what is read and what action these tables are capable of doing. One can argue that the actions can be done before the tables. However, defining tables before their actions makes it clearer what each action must do. To finish the P4 program, the actions must also be defined. Here is where the packet changes happen, whether it is in the ingress or egress because both of them have their tables and actions. This does not mean that both the tables and action are obligatory to be unique.

2.4.2 Domino

Domino is a programming language that introduces a new abstraction for the data plane, i.e., a packet transaction, that is like the distributed systems property, but it is made for the SDN model, to the data plane in particular. Specifically, packets become a sequential code block that is atomic and isolated from other such code blocks [22].

It also contributes with a machine model called *banzai*, a new domain-specific language, and because it introduces a new language it also offers a new compiler for it. In the machine model, the computation is modeled within a stage. An important note is that this machine model assumes that the packets that already arrive at it are parsed a priori. Concretely, it is a feed-forward pipeline, where each stage processes one packet every clock cycle and forwards it to the next stage in the pipeline. Domino features the fact that it never stalls. Thus, it means that it is deterministic and always sustains line rate [22].

To offer atomicity, the Banzai machine model can use multiple atoms within each stage. An *atom* is an atomic unit of packet processing. Concerning these atoms, seeing as they are modeled after their distributed counterpart, they also can only offer simple actions as well. And like their distributed part, they also do not share state within stages and neither with other stages. However, for the latter, there is a way to allow for it to be possible and that is using headers. By simply writing to the header and then later on the pipeline reading it, it is possible to share state [22]. Another important aspect about them is that since there is a need to guarantee atomicity, any state changes must be done before any other packet can be handled.

Regarding its language, the syntax is very similar to C. However, it presents several constraints because we are handling a language that needs to guarantee deterministic performance, and unbounded iteration counts. For restraints to be possible, it removes iterators, unstructured control flow, and does not provide heap access. It also constraints array access in a peculiar way, such that, the array can only be accessed using a single packet field and if that field is altered such access will be denied.

Finally, its compiler has three phases: (i) the first one receives the Domino code and preprocesses it, removes extra branches that would complicate the analysis later on. It also rewrites state variable operations as well as converting complex actions on packets to simpler ones, and finally, it flattens those operations to a three-address code, which is

a representation where all instructions are either reads/writes into state variables or operations on packet fields [22].

The second part is simpler. It turns the sequential code block created for the earlier stage into a pipeline of codelets. *Codelet* is a sequential block of three-address code statements. Because this part simply makes the codelets, it has no respect for computational or resource limits, and as such, another stage is needed.

The third and final part of the compiler is the code generation. Here is where the compiler will analyze the codelets and determine whether or not they can be compiled in the banzai machine. When the codelets are made, they do not take into consideration the limits of the banzai machine. After these are made the compiler simply analyses the codelets for violations on the pipeline width. If there is such a violation, new stages can be inserted as to cover for this mistake. However, if the number of codelets per stage is bigger than the depth, they are rejected. What was described above is called the resource analysis, concerning the computational analysis, this one is way more difficult and is as follows. The codelets most of the time have multiple three-address code statements that need to be executed atomically. And to try and resolve this issue, each codelet can be viewed as a functional specification of the atom they are representing. Since this task was shown to be rather difficult, they used an external software called SKETCH to express the atom templates [22].

2.4.1 Discussion

While P4 is a new language, the other ones have already made their debut for quite some time, which would make them more appealing than P4. However, if we go into more detail, and as stated above, P4 is a C-like programming language directed to programable networks, similarly others, hence not presenting any kind of gain here.

Nonetheless, P4 offers a major advantage as it can support a dynamic parser allowing programmers to define their headers and the packet parser will know how to handle them. Despite the extra work, there is also a lot more freedom to choose with what to work with. Moreover, there is also the fact that with P4 it is possible to dictate what forwarding rules we want.

Next, the rationale underneath the selection of P4 is presented. First, there is much more support from the community regarding any kind of question in P4. Unfortunately, the same cannot be said for the other ones. Second, P4 is a rising star in the scientific

community and thus more appealing to work with. Third, there is the support that the language itself provides to its programmers. Since it is a C-like programming language, its learning curve is much smaller, and, therefore, more enticing for programmers to use.

2.5 Simulation and Emulation Tools

In this section, it is provided a bit of background information regarding the state-of-the-art tools that can be used in programmable networks to either emulate or simulate them. Afterward, a brief analysis of them is given, and to conclude this section, we will explain the reason for selecting the tool utilized in the implementation and evaluation sections.

2.5.1 NS-3

Regarding this software, it is a ladder software, with this, it means that it was built on top of previous software, the NS-2 simulator, and made lots of improvements that made it independent from its predecessor. The most important and prominent one is the improvement in the core with major regards to scalability: it was written in C++ with an optional Python scripting interface [23]. It also leverages a lot of C++ design patterns to improve the scalability of the core, such as smart pointers, callbacks, and many others [23].

With the increasing need for realism and faithfulness of the tests, NS-3's nodes were designed to include sockets and IP addresses. And, with the previous in mind, they also included both software integration and a testbed with their release. With this, they aimed to support more open-source software that could use kernel protocol stacks or even routing daemons. When they said they wanted to support more open-source software, they meant to enable the testbed-based researchers to experiment with the protocol itself [23].

Finally, they also added support for virtualization. Here, they offer lightweight virtual machines that could be run over a simulation network. This network can be wireless.

2.5.2 Mininet

Mininet [24] is a software that allows for rapidly prototyping large networks on a single computer. This network is scalable, however, according to [25], it is not very much so. Nonetheless, it uses a feature that NS-3 also uses, a lightweight virtualization mechanism.

Other very important characteristics of Mininet are: (i) it tries to be flexible, that is, to support new topologies and features using programming languages and common operating systems; (ii) it is applicable in many case-type, that is, if the implementations are done properly, then the prototypes based on its implementations can also be used in real networks, without any changes whatsoever; and lastly, (iii) it tries to be realistic, in the sense that, the behavior present in the simulations should represent real-time behavior with a high degree of confidence [24].

Mininet's network elements are hosts, switches, controllers, and links. A host on Mininet is a simple process with its virtual network interface, ports, addresses, and routing tables (such as Address Resolution Protocol (ARP) and IP). The OpenFlow switches created by Mininet provide the same packet delivery semantic that would be provided by a hardware switch. In the simulation, the controllers can be run on the real or simulated network. The Mininet emulator implements a connection between switches and different controllers. Figure 4 illustrates a Mininet topology example with the help of a GUI developed by Mininet developers (MiniEdit).

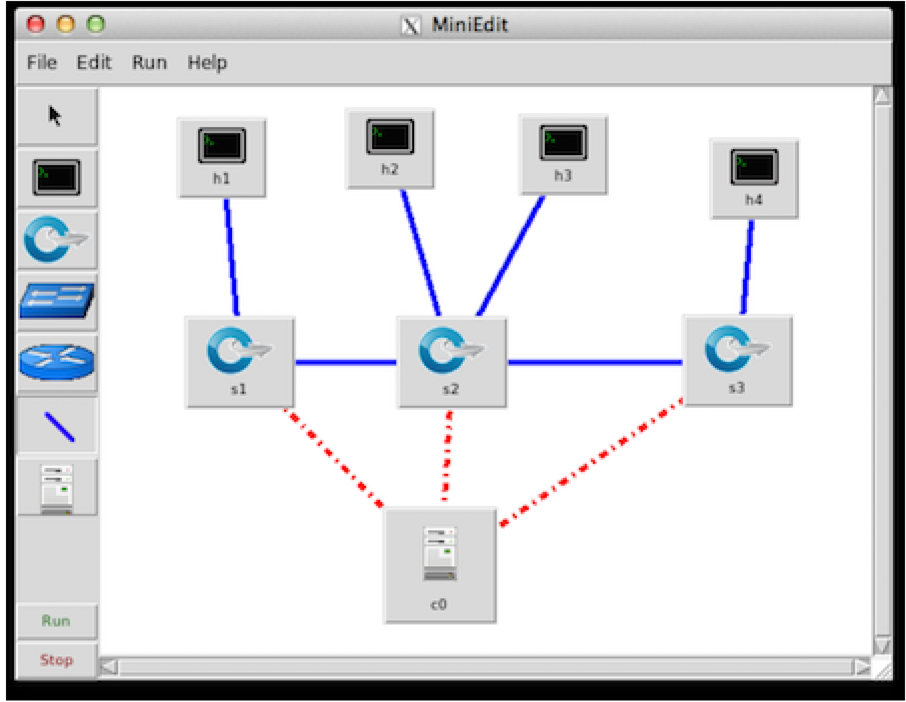


Figure 4: A Topology Example Using a GUI for Mininet (MiniEdit) (Extracted From: [24].)

As stated before, in Mininet, the hosts should be isolated. Switches use either the default Linux bridge or the Open vSwitch [12] running in kernel mode. Switches and routers can run in both the kernel space and in the userspace. Lastly, the links use the Linux Traffic control (TC) to ensure the data rate of each link is what the programmer wanted it to be. However, Mininet as with every other software has its drawbacks, such as Mininet-based networks cannot exceed the CPU or bandwidth available on a single server. Also, Mininet cannot run non-Linux-compatible OpenFlow switches or applications [24].

2.5.3 EstiNet

Similar to the previously presented tools, EstiNet [25] is also an OpenFlow network simulator and emulator. However, it uses a unique approach with regards to checking the OpenFlow controllers and analyzing the functions. This approach is named *kernel re-entering* in which simulation and emulation are merged. What this means is that each host can have a real Linux system running and in this system, any application that is UNIX-based can be run and simulated without any modifications.

Specifically, for EstiNet to be able to do *kernel re-entering,* it uses some tunnel network interfaces to automatically catch the packets sent by two real applications and

transmit them into the EstiNet simulation engine [25]. With this technique, it can guarantee that the results achieved by simulation are reliable and accurate as those that are obtained by emulation. Another good feature that this software offers is the fact that the simulations can be run at different speeds, i.e., faster, or slower than real-time. This trait can be very important since it enables the correct simulation of an OpenFlow network with a very large number of switches and hosts.

EstiNet works as follows. First, the packets are sent out, by let us say Host 1. These are put into an output queue of the tunnel interface, from which the simulation at a later part of the protocol will fetch them. After the fetch is done, the processing is simulated over the protocol stack for Host 1, i.e., to simulate PHY/MAC and many other protocols that the host can have. Afterward, the packet is reprocessed in the reverse order. Then, it is delivered to the queue and finally delivered to the final application of Host 2 [25].

2.5.4 Discussion

First and foremost, when comparing Mininet with the other tools, one could say that Mininet would have more drawbacks. Nevertheless, and with that in mind, Mininet would still be selected over the others, mainly because of previous experience with the tool. There is the fact that Mininet directly supports the use of OpenFlow, meanwhile, in other approaches, there is a need to use add-ons to even make it possible to support OpenFlow.

Overall, even though others could be better in some respects, Mininet has broad community support and presents a more welcoming environment, whether for a more experienced programmer with background knowledge or for new programmers that are trying to learn from scratch.

Chapter 3

The proposed solution

In this section, we will cover every aspect of the design and implementation of the proposed solution. First, we start with the design choices, and then we will explain the evaluation scenarios and their implications. Later, we will go into detail regarding the specifics of the proposed solution.

3.1 Solution overview

First, we started by investigating both P4 [7] and, more importantly, its models (i.e., PSA and V1), that we would employ in practice. In particular, this model was the V1 model [8], as it was the basis for the whole implementation. But even more important than that model was the whole specification of P4 [26]. The use of the V1 model over the PSA model was due to its simplicity in implementation compared to the PSA one.

Referring to the state machine, the state machine's implementation took place using the various match-action tables, sometimes changing their contents depending on the purpose of the table.

The starting point of our implementation was FAST. However, and considering the difference between Open vSwitch and P4, we needed to add tables to ensure that actions related to keeping the information in the packet and/or switch itself were properly implemented.

The most efficient way to explain how we took care of a state machine in P4 is to present the following figure.

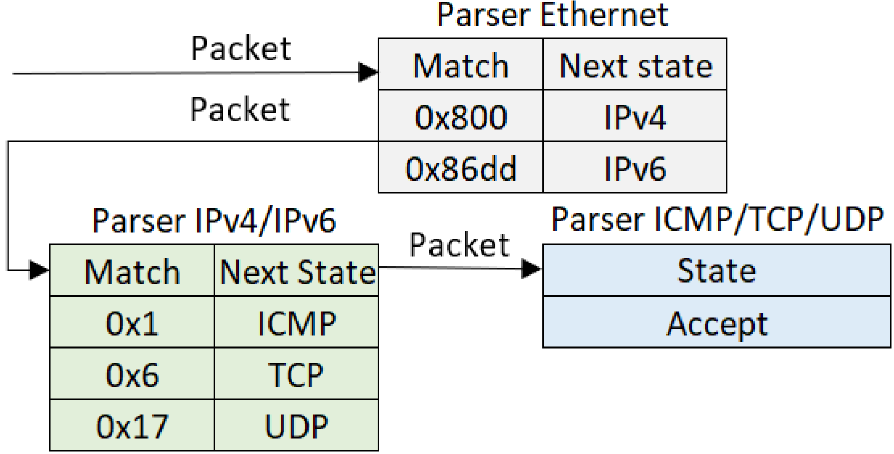


Figure 5 - State Machine Example Using our Parser

With Figure 5, we can see that when a packet arrives at a switch it will first try to match the contents of the ethernet header to know where to proceed next. Afterward, it will match the contents of the IPv4/IPv6 header to again know where to advance, only after that will it accept and will only finish when it is done with the contents of the ICMP/TCP/UDP header.

And to further augment the previously shown picture, we will also provide the following piece of code that shows how we implement a table in P4.

```
1. table ipv4{
2.     actions = {
3.       setNextHopIPv4;
4.       drop;
5.       NoAction;
6.     }
7.     key = {
8.       hdr.ipv4.dstAddr: lpm;
9.     }
10.      size = 1024;
11.      default_action = NoAction;
12.    }
```

The design of our solution is rather straightforward. Given that P4 requires data structures, such as the headers and the metadatas, they were created to enable access to the content of the packet. To give a better understanding, we present the following lines of code to better illustrate.

```
1. header ethernet_t {
2.   macAddr_t dst_addr;
3.   macAddr_t src_addr;
4.   bit<16> eth_type;
```

```
5. }
```

Above we present a header type, i.e., the ethernet header, while below we show all the header types we utilize in our solution.

```
1. struct headers {
2.    ethernet_t      ethernet;
3.    ipv6_t          ipv6;
4.    ipv4_t          ipv4;
5.    infoKeeper_t    infoKeeper;
6.    tcp_t           tcp;
7.    udp_t           udp;
8.    icmp_t          icmp;
9. }
```

We now show an example of how to access information inside the packet.

```
1. hdr.ethernet.dst_addr = dstAdd;
```

```
TCP Timestamps option (TSopt):

Kind: 8

Length: 10 bytes

      +-------+-------+---------------------+---------------------+
      |Kind=8 |  10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
      +-------+-------+---------------------+---------------------+
          1       1              4                     4
```

Figure 6 - A Timestamp Example (Extracted From: [27])

Therefore, when packets arrive, they are parsed, i.e., run through the ingress pipeline where the maintenance of the state would occur via special tags that would be added in the TCP options, i.e., the Timestamp option [27], as is shown in figure 6 above. We decided to use this option because it provides a very easy and straightforward way to send timestamps across switches. Also, in case we do not need to send them, this option provides space for two 4 byte sized timestamp fields that could be filled with information [27].

```
1. bit<32> indexCounter = 0;
2.     indexForPackets.read(indexCounter, 0);
3.     networkPortsEgress.write((bit<32>)indexCounter,
   meta.ingressMetadata.egress_port);
4.     networkPortsIngress.write((bit<32>)indexCounter,
   meta.ingressMetadata.ingress_port);
```

In the code shown above, we show how state maintenance occurs in our solution First, we initialize a variable where we will store our index to access the registers. Afterward, we read from the register where the indexes are stored and place the value in the *indexCounter* variable.

Then, we write in the registers the values of the ports to create a fictional knowledge of the state of the network.

```
TCP SACK Option:

Kind: 5

Length: Variable

                  +--------+--------+
                  | Kind=5 | Length |
+--------+--------+--------+--------+
|      Left Edge of 1st Block       |
+--------+--------+--------+--------+
|      Right Edge of 1st Block      |
+--------+--------+--------+--------+
|                                   |
/            . . .                  /
|                                   |
+--------+--------+--------+--------+
|      Left Edge of nth Block       |
+--------+--------+--------+--------+
|      Right Edge of nth Block      |
+--------+--------+--------+--------+
```

Figure 7 - A SACK Example (Extracted From: [27]).

Please note that we used the SACK option, however, as is shown in figure 7 the SACK option [28], but not in its intended way. Instead of containing the planned blocks of data, i.e., two 32-bit unsigned integers, we used this space to put the egress and ingress ports in the packet. We use these specific ports so that when switches have a few pairs of them, they can be aware that traffic flows from these ports. Consequently, they can have a fictional knowledge that it is safe to send packets through them. We use the term fictional knowledge here because it is not the switches themselves that have this knowledge, but the program that is running inside them, that is, the program receives information from the switch, but the reverse does not apply. Finally, we use the idea of piggybacking because the switches themselves do not allow for packet creation and also to avoid creating new packets and, and, as such, we get better performance from the overall network.

```
// intrinsic metadata
@alias("intrinsic_metadata.ingress_global_timestamp")
bit<48> ingress_global_timestamp;
@alias("intrinsic_metadata.egress_global_timestamp")
bit<48> egress_global_timestamp;
```

Figure 8 - V1 Metadata that is Used

Conversely, the Timestamp option is used normally. In its TS Value, we put the timestamp of the packet so that after the first switch, every other can make the necessary interactions with it. Other switches can compare their timestamp values to the one present in the packet, and depending on the value of the difference, they will then take the actions needed when certain conditions are met. These conditions and actions are rather simple. If the difference between the timestamp that is stored on the TCP Timestamp option and V1 model metadata is greater than an average, it is assumed that the link can be having connectivity issues, and as such, we start making clones.

In both figures 9 and 10, it is possible to observe that there is no difference between a cloned packet and a regular packet.

```
▶ Frame 2: 82 bytes on wire (656 bits), 82 bytes captured (656 bits)
▶ Ethernet II, Src: 00:00:00_00:01:01 (00:00:00:00:01:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▶ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.7.4
▼ Transmission Control Protocol, Src Port: 53339, Dst Port: 1234, Seq: 0, Len: 0
    Source Port: 53339
    Destination Port: 1234
    [Stream index: 1]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 48 bytes
  ▶ Flags: 0x002 (SYN)
    Window size value: 8192
    [Calculated window size: 8192]
    Checksum: 0xbc2f [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  ▶ Options: (28 bytes), SACK, Timestamps
```

Figure 9 – Example of a Cloned Packet

```
▸ Frame 2: 82 bytes on wire (656 bits), 82 bytes captured (656 bits)
▸ Ethernet II, Src: 00:00:00_00:01:01 (00:00:00:00:01:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.7.4
▾ Transmission Control Protocol, Src Port: 53339, Dst Port: 1234, Seq: 0, Len: 0
    Source Port: 53339
    Destination Port: 1234
    [Stream index: 1]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 48 bytes
  ▸ Flags: 0x002 (SYN)
    Window size value: 8192
    [Calculated window size: 8192]
    Checksum: 0xbc2f [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  ▸ Options: (28 bytes), SACK, Timestamps
```

Figure 10 – Example of a Normal Packet

This is the only to ensure that a packet is going through a given egress port that is defined a priori without having ways to change it. There is no other way to modify the value later in runtime besides stopping the network, this is because P4 itself does not allow any interaction to be done with the tables during the execution of the programs.

A clone is merely a complete copy of the packet, and it is sent through a different egress port that is defined as a priori. The figure below further illustrates this.

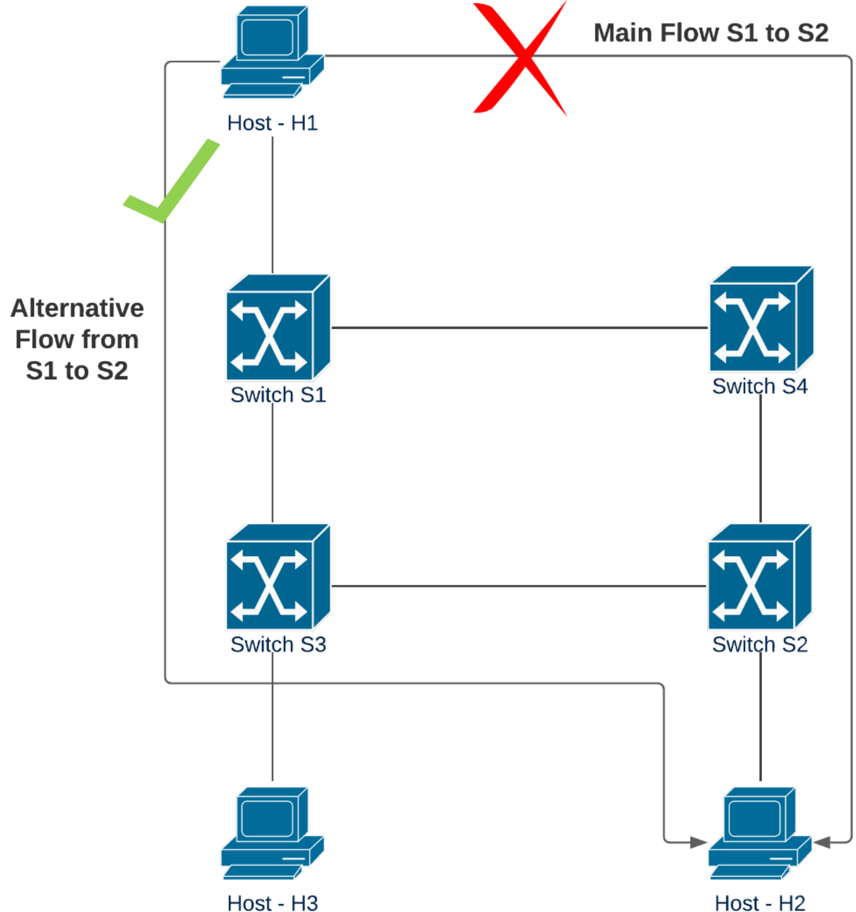


Figure 11 - An alternative path for packets to go from H1 to H2

As was stated before we utilize an exponential weighted moving average (EWMA) as shown in the equation (1). When the EWMA reaches a value that greater than the current timestamp present in the packet, then packet cloning starts, and they are sent through an alternative route.

$$\mathrm{mv} = \mathrm{mv} \ * \ 875 + (ct - lt) \ * \ 125; \tag{1}$$

Equation 1 - EWMA Calculation

where *mv* is the value of the moving average, *ct* is the timestamp present in the metadata, and *lt* is the timestamp that is kept in the switch.

Afterward, we do not do anything in the egress pipeline, and every change is made during the ingress phase of the implementation. This will be fully explained in Section 3.2.5. Finally, we remake the hash of the packet as we insert new data to it because if we

did not and another switch was to verify the hash of the packet with its contents, it would not match and the packet would be discarded,

Next, we deparse the headers in the correct order, otherwise, this would mean that the packet would be poorly formed and would be discarded by the switches.

Please note that before achieving this particular design, we first thought of several ways to concretize this implementation.

First, we tried to use probe-type packets that would be generated by the switches, and as the network changes, e.g., increase or decrease the number of network devices, we envisioned that the packets would be sent by the switch and they contained information so that remaining switches, could be aware of the events at had happened.

We also came with the idea of potentially using a new header. Specifically, instead of using the piggybacking method, our information that would be held within the TCP options [27] would be instead kept in a brand new header. The header would be the simplest possible and its position in the header stack would be after the TCP header. At first, this idea was the main focus but after implementing and testing it extensively, we had to come to terms with it and put it away as there were too many problems with it.

3.2 Implementation

In this section, we will be covering most of the aspects necessary to understand the implementation of the proposed solution. However, we first introduce the system model.

3.2.1 Network Configuration

For this experiment, we used a simulated network through Mininet that would set up for both switches and hosts. There were four switches (Switch 1 – S1, S2, S3, and S4), each switch would be directly connected to a host. There is one exception that is the switch S4, which would be an intermediate switch between S1 and S2. This is better shown in figure 12. There are three hosts (Host 1 – H1, H2, and H3), where each one is connected to their match number switch. For example, H1 is connected to S1. There are two main flows in this network: Flow 1 – F1 that is from H1 to H2 passing from S1 to S4, and finally from S2. And F2 from H3 to H1. F2 is what we call a secondary flow since its packets are mainly used to ensure that packets that arrive at S1 can have multiple ways to check whether or not there is something wrong with the network or not (see Figure 8 for more information).

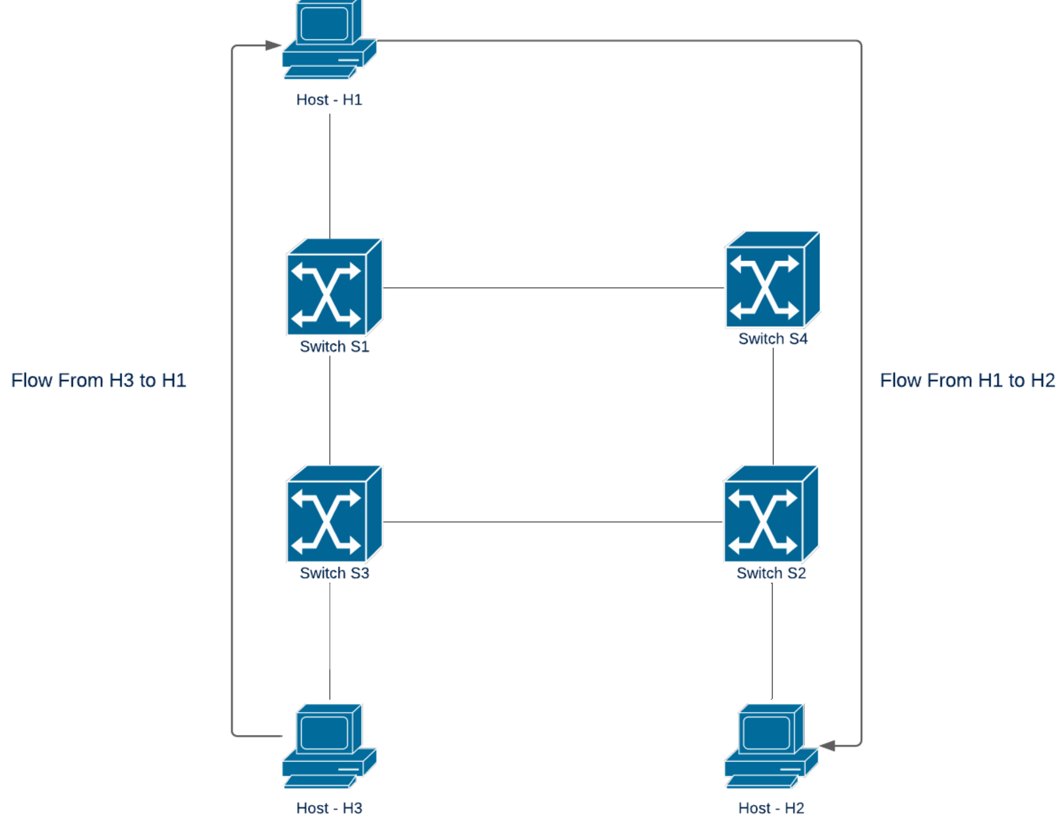


Figure 12 - An example scenario

To simulate the host sending the packet, we use a simple set of commands in Python version 2.7, that would send a number of packets that the user would insert in a command line or that could be inserted when launching the Python file. We show the following line of code to help better understand the inputs that need to be done for the Python script to work correctly.

```
$ ./send.py IPAddress SendingMode NumberOfPackets
```

There are two ways of sending the packets: (i) a burst way, where the packets are sent as fast as possible, which means as fast as the link permits it, or (ii) periodical way, where the user must also input the amount of time that must wait between sends, to simulate a more congested network. Please note that the user must input the IP address of the receiving host.

After we run the file, we start by creating the packet one layer at a time, starting from the link layer, where we make a simple ethernet header, and specify both the source and destination MAC addresses. Afterward, we add the internet layer, where we add the

IP header, and simply add the destination address. And, finally, we add the transportation layer, where we put both the ports, destination, and source, as well as TCP options used in the program. However, here we simply make space for where the information will be placed when the packet arrives at the switch.

Regarding the receiving side, another Python script (`receive.py`) is used that is run when receiving a packet arrives. In it, we do a few checks to know if it has the correct fields, such as the timestamp, which is the main one. If it has that one it will do a few checks and calculate an average to make sure everything is running smoothly. Lastly, it prints to the command line the whole packet alongside its fields. We keep the file running while it is waiting for more packets to arrive. Like with the sending script we will also provide how it needs to be run for it to work correctly with the following line of code.

```
$ ./receive.py
```

First and foremost, we used the idea of piggybacking of other packets that are circulating in the network because the switches themselves do not allow for packet creation to do done. To do so, we needed to make space in the packet to use, this is because we need to be able to send from one switch to another all-needed information and that would take some space. In our case, it would take the size of a timestamp and four ports. And as such, we used the TCP options [29], in particular both the SACK and Timestamp options among others like the No Operation Option [30] or the Window Scale Option [27].

We use the SACK and the Timestamp Option, simply because they provide the most amount of free space to the packet and more importantly, we use the Timestamp Option because it provides an already made way to send the timestamp without any issues. The latter is used in the following way. Since the TCP Options uses a fixed size for its timestamps of 32 bits [27] and the size of the timestamp that is provided by the model we use, the V1 model, are 48 bits, we must change its size. The code below shows how we take care of formatting the timestamp.

```
1. bit<224> newTimestamp = ((bit<224>)((bit<32>)
   standard_metadata.ingress_global_timestamp)<<32);
```

Please note that the other alternative provided by the same model after thoroughly experimenting with it, was done by simply changing the value of the variable that was

being cast to the one *enq_timestamp* from the V1 model [8], but since the value was always 0 we discarded this option. So, we simply cast the 48 bits into 32 bits and shifted them into the right position, to be more precise the 12 most left bits would be discarded for it to fit in the premade space.

Only afterward would we cast the timestamp into the size of the TCP Options (which is 224 bits). This is only possible since P4 when we cast something from a smaller size to a bigger one it fills the remaining space with zeros.

After investigating the Timestamp option, it was clear that the left-most 32 bits were relative to the answer timestamp, so it would be needed to skip these and put our timestamp in the correct place. To further aid in understanding this part we present figure 13.

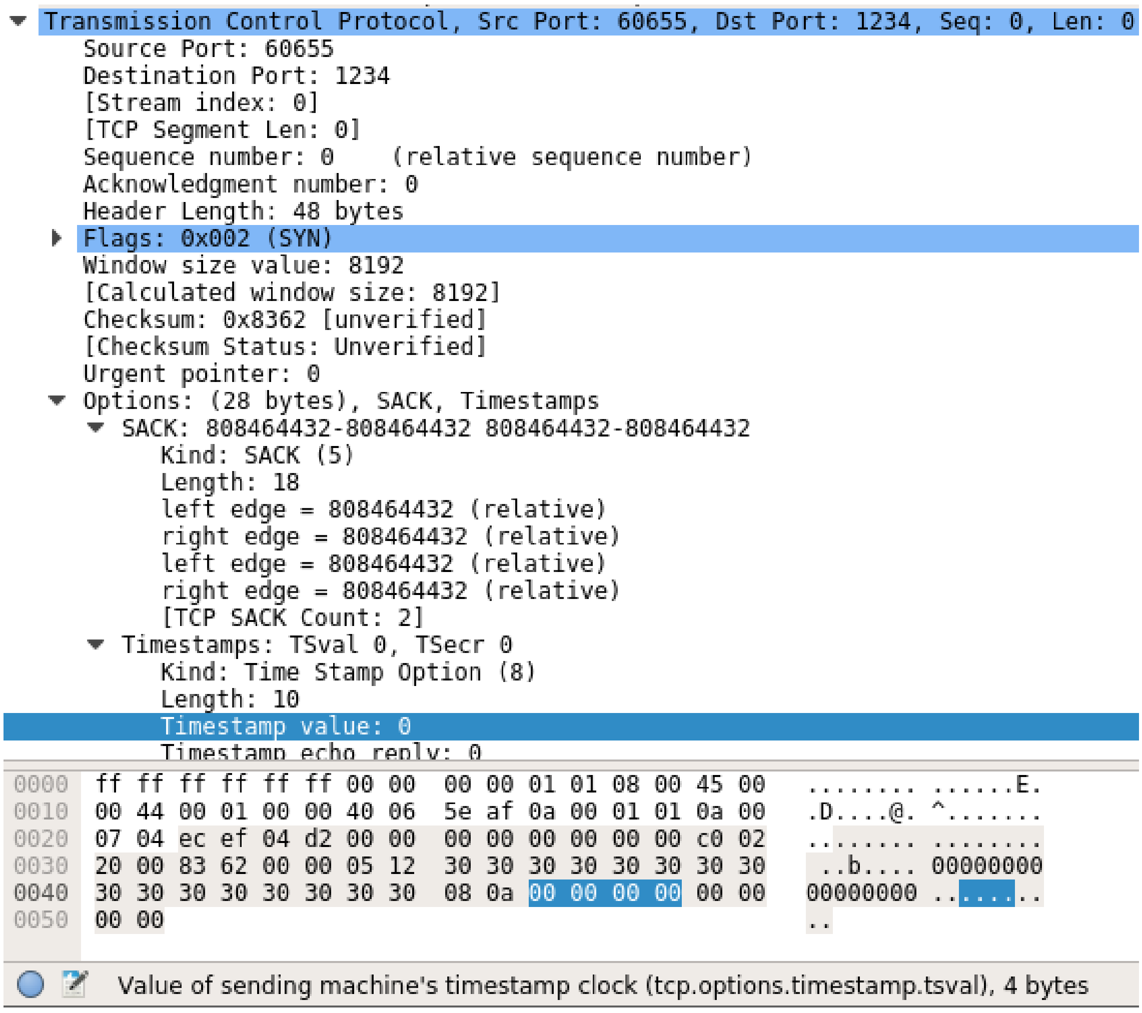


Figure 13 - Example of TCP from a pcap file in Wireshark

In it, we can see that the last bits present in the packet are for the timestamp echo reply and need to be skipped.

With this done the timestamp would be in place and the part left would be putting the egress ports. Since these are 9 bits long, we used shifts of size 12 to put them in their correct place. This is regarding the former part of the options. This allowed for four multiple size 32 bits where we can put our egress port ID in to share with the other switches.

Some information checkers are related to verifying whether the switch has the same values as the egress port ID. This was only done since a way to directly interact with these values was not provided by P4, and as such, we had to use such methods.

3.2.2 Registers and Metadata

Considering the use of registers in P4 [7], also the way they are used in this implementation is very peculiar because since we consider those two main flows, we employ the first possible index of them for the first flow (index number 0) and the next one (index number 1) for the second flow. The flows are respectively, F1 and F2 as was described in chapter 3.2.1.

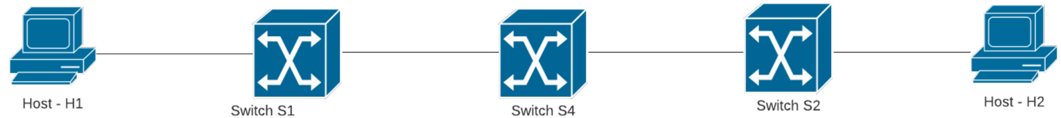


Figure 14 - Flow from H1 to H2 – F1

Figure 15 - Flow from H3 to H1 – F2

```
1. register<bit<size>>(size) variableName;
```

Before moving the main object that is the match action tables, first, we need to include everything that they could make use of, in particular, there are the actions, that are all mostly, made by us, except for the drop action, which, tells the switch to drop the

packet, and the metadata the is provided by the V1 model [8] and the metadata that we make to, yet again help with the program.

For the metadata related to the V1 model, we make use of the variables related to the ports of both the egress and ingress, lastly we also make use of the ingress global timestamp variable as this is where the timestamp is stored in this metadata. For our metadata, we could work with two structs, but since the need to interact with the egress part of the program was not raised, it is not utilized.

Also, we work with two, simply because if there is a need to expand this metadata to be used in both ingress and egress arises, it is much easier to proceed than at that time try to offload and make the needed structs. By doing so, the possibility for variables to be misplaced would also arise.

```
1. struct metadata {
2.     ingress_metadata_t ingressMetadata;
3.   //egress_metadata_t egressMetadata; this can added for egress
   support
4. }
```

More concretely, our ingress metadata has 8 variables:

1. one that marks if the packet is to be dropped;
2. one marks if the packet is from the host H3;
3. one that marks if the packet is the first packet that has passed through the switch so far;
4. one that marks which type of packet it is, basically this is used to simplify the identification of the internet layer in later operations;
5. the egress port, this one simply stores the egress port from which the packet will leave after all the processing is done;
6. the ingress port, similarly, to the one before, this one also stores a port. however, this one stores the ingress port, which is the port from where the packet came from. both this and the last one will be very important later on as they will be part of the way we use to make sure the consistency in the network is kept;
7. A variable that simply stores the last timestamp.

```
1. struct ingress_metadata_t {
2.   bit<size> drop;
```

```
3.   bit<size> fromH3;
4.   bit<size> firstPacket;
5.   bit<size> packet_type;
6.   bit<size> egress_port;
7.   bit<size> ingress_port;
8.   bit<size> lastTimer;
9. }
```

3.2.3 Actions

Now, we cover the actions of the program.

They are many and vary a lot themselves, from simply setting the few bits in a variable to a value to read from registers and writing in the TCP options the timestamp, and many other factors.

```
1. action actionName(actionParameter1, … ,actionParameterN) {
2.     //events to do in the action;
3. }
```

As was stated before, there are:

(i) drop action, which tells the switch to drop the packet, for whatever reason, whether it is malformed or there is an error, the switch simply drops the packet.

(ii) there is the action of identifying the type of internet packet, which again only sets a value in our metadata to a simple value. This action is used in a switch case that when this action is to be triggered, allows for others to be done as well.

(iii) the regular IPv4 action, since our testbed employs only IPv4 packets there is the need to make sure that everything is done correctly, so we subtract one from the Time to leave (TTL). More importantly, we set the egress port, without this particular line in the code even if everything else was done properly, without it, the switch would not know where to forward the packet. Also, this egress port as was mentioned before needs to be set a priori in another file. Lastly, here we set the destination MAC address, like before this also needs to be set a priori.

(iv) the keepStatus action, which as the name suggests is used to keep track of the status of the network. Here, we read from a register that keeps the data of where to write the information that needs to be preserved.

We keep both the address of the source and destination of the packet and both the ingress and egress ports where the packet is headed and came from, as to create a sub information network inside the switch. Since this can be heavy on the switch, we employ a FIFO queue (First In, First Out), of sorts. This is done by the number that is kept in the

register when it gets to a certain value, it is reset to its initial value (that would be 0), and all the information contained in both the address and ports is overwritten, for the sake of space efficiency.

(v) Last of all, there is the keepData action, where the information is stored inside the packet.

As was described before, this is done in a somewhat hardcoded way, via shifts. Firstly, we read from the register where the egress and ingress port are stored away and then we make a new variable that needs to be the same size as the TCP options, this is where all the information that needs to go on the packet is put in storage.

Before explaining how we proceed to put the information (the egress ports and ingress ports that have been stored previously), the easiest part is the storing of the timestamp where we simply diminish its size and put it in the correct location inside the previous variable.

```
1. bit<224> newTimestamp = ((bit<224>)((bit<32>)
   standard_metadata.ingress_global_timestamp)<<32);
```

After That, we just put the information in temporary variables. These variables in turn will be put together into a bigger variable.

```
1. bit<224> firstEgressPort = ((bit<224>)egress)<<176;
2. bit<224> firstIngressPort = ((bit<224>)ingress)<<188;
3. bit<224> firstPorts = firstEgressPort + firstIngressPort;
```

After all the information is collected into the bigger variables together with the timestamp they are put inside the packet.

```
4. hdr.tcp.options = newTime + firstPorts + secondPorts +
   thirdPorts + forthPorts + newTimestamp;
```

3.2.4 Tables

Now, we explain the idea behind the many match-action tables and the way they are separated.

```
1. table tableName {
2.     actions = {
3.       actionName1;
4.       actionNameN;
5.     }
6.     key = {
```

```
7.        headerName: typeOfMatching;
8.      }
9.      size = number;
10.       default_action = actionName;
11.     }
```

Firstly, we do the regular tables that are in charge of dealing with the normal parts of the packet, such as, IPv4, IPv6, TCP, and UDP. These are done in the normal and regular manner, that is, for the IPv4 and IPv6 protocols, we use the destination address as a match with the longest prefix matching of 32 bits, which is only used in this specific test case. For the TCP protocol, we utilize the destination port for the matching with the need for it to be exact to the one inserted in the table beforehand. However, this is not a problem since the main focus of this mainly lies in the Internet layer. If necessary, this can be done easily. The same can be applied to UDP. We also use the destination as a match and employ an exact match for it.

Last but not least, there are many tables we introduce for the sake of simplification. These are mainly used to help break the code into parts so that is it easier to both read and separate what each part does.

First, there is a table that simply does, the separation of the type of Internet packet (whether is it IPv4 or IPv6, so to enable support to IPv6), even though the standard metadata present in the V1 model, already contained this information. We thought it would be beneficial to use a separate table and for it to use the action related to the identification of the packet. In this table, the type of packet present in the data layer (MAC address) is used as a match, we needed for it to be exact.

Two more tables are very important to the overall program which are the updateStatus and the updateData tables. The *updateStatus* uses as a match the same as the IP for ease of use. However, what happens when the trigger is done is very much so different. In this case, the *keepStatus* action is prompt with all its related effects. In the *updateData* table, it is very similar to the *updateStatus* in the sense it also utilizes the same match as the IP table. However, like the previously explained table, it activates its action that is called *keepData,* and all the associated effects.

It is also worth noting that all the tables have a capacity of 1024 entries. And all the tables in case of failing to have a match they all do not do anything.

3.2.5 The workflow of the program

After the packet is properly parsed and ready to be handled by the switch, this is where we run our code. Although packet parsing rarely fails, and since that possibility exists, we must make appropriate verifications. To do so, we run the table that determines which kind of internet layer we are dealing with. Again, since we only use IPv4, we only refer to this type. We check to see if the IPv4 header has any errors. If it does, we do not do anything and we skip any execution. But if it does not, we run all the important tables that handle both the consistency and the information kept in the switch.

However, since we only want switches that are directly speaking with hosts to keep track of the network, and for the others to simply work as forwarding switches, we add a check for the execution of the table keepInfo. Then, we continue to check the packet and examine its headers. Since we are done with the internet layer, we proceed to the transport layer, where we consider the TCP and UDP protocols.

For this purpose, we yet again make verifications to see if the header contains any errors. If it does, we skip any actions that would be taken in it. But, if there are not any errors in this part of the packet, we apply the TCP table which does not do much. The main portion of the program comes after where we verify the TCP options and make a moving average of the timestamps. We will now explain in further detail how this part is done.

Firstly, we make sure that we are not working directly with the timestamp present in the packet, and as such, we make a temporary copy of it. That is what we will work with extensively. Next, we access the registers that contain the information where to get both the current packet number, which is the number of packets that have passed through the switch so far, since the switch was turned on and an index to access the other registers where the information relative to the port is stored. As was stated before this index is used to access those registers and to try to optimize the memory of the switch. We utilize the FIFO queue and make sure that if the index reaches a certain value it is reset to its base value.

```
1. registerName.read(variableName, index);
```

Then, a few verifications are performed, because in this test case, we only want to store the value of the main flow (F1), which is from H1 to H2. The other flows' values will not be stored in our case. Here, we set the value of the current timestamp to the value

present in the variable from our metadata and the V1 model metadata as well. Next, we update the values in the registers and set the value of the variable present in our metadata to the appropriate value.

After That, we make a crucial check that ensures that the current packet is the first packet passing through the switch. Then, we need to set the value of the moving average to its value, i.e., it is the value of the current time present either in the metadata or the register. Lastly, we set a variable that serves as a Boolean. This is done because when we used a Boolean, here the program simply would not compile. This value will be used later on in the calculation of the moving average.

```
1. bit<size> nameOfTheChecker =
   hexadecimalRepresentationOfTheTCPoptions;
2. bit<size> timestamp = hdr.tcp.options & nameOfTheChecker;
```

Following, via a bitwise checker that negates all the bits present in the TCP options except the ones that should represent the timestamp, and with that, we get the timestamp present in the packet. Before making any changes with this value, we make a check if the packet was cloned by another switch, and if it was we do not do anything. We also check if the packet was sent directly by a host, again if it did not pass this check, we do not do anything. Lastly, we check the timestamp to see if there is literally a value.

If it managed to pass through these checks, then we extract the ports from the packet, once again via bitwise checkers that first negate every part of the TCP options besides the area where the port should be and place the value in a variable, only after it was shifted and cast into an appropriate size. We make this for every port that should be present in the packet.

```
1. bit<size> nameOfTheChecker =
   hexadecimalRepresentationOfTheTCPoptions;
2. bit<size> firstEgressPort = (bit<size>)(hdr.tcp.options &
   checker>>numberOfWhereToPutThePort);
```

When this is done, we carry on with the needed check for the storage of the ports within the switch. For this, we simply read the value that would be kept inside the proper register and compare it with the one that was previously extracted from the packet.If they are different, we assume that the ones contained in the packet would more recent and simply write their value in the register. This is done once again for every port that is retrieved from the packet.

We arrive at the part where we take care of the moving average. We must say that it is done peculiarly since P4 does not allow for some actions to be done on values that are input at runtime such as the timestamp that we are dealing with and per se. We needed to work around this limitation. The way we found was instead of using decimal values, we simply multiplied the values by a constant and that solved the problem.

There is also another factor that needs to be taken into consideration in this part and this is the fact that P4 supports neither doubles nor division over non-integer variables. Besides, there is also the fact that P4 does not check for under or overflows, and due to that fact we need to work with bigger variables. To that effect, we decided to work with variables related to the average having double the needed size and that seemed to solve the problem.

Now, we clarify in detail how we do the EWMA. Like before, we try to optimize this part of the program, since it would be intensive on the switches. We only consider the packet that comes from the main flow (e.g., F1 between H1-H2) and the ones that do not match this condition we merely utilize the average calculated beforehand and check to see if the difference between the timestamp present in the packet is bigger or equal to the average. If it is, we continue to check if the packet was directly sent by a host.

If none of these conditions are met no action is taken. Although, if both are, we know that there is a problem within the network since the delay between the packet in question and the previous one is large enough to trigger the condition mentioned above and we need to utilize another route. This is done by cloning the packet and when that is done we tell it to go on another egress port that needs to be different from the original one. We employ a register that simply exists for this sole reason if the value present in it is 1 then there a problem in the network, otherwise, that statement is false.

If the packet comes from the main flow, first we need to check the packet if the first one or not. If it is, we need to change the value of the auxiliary variable present in the metadata made by us, and nothing else should be done in the ingress pipeline. However, if that is not the case, we need to calculate the moving average. To do so, we calculate the difference between the timestamp extracted from the packet and the one that is in the switch. Once this is done it is stored in a variable.

Subsequently, we calculate the average via multiplication of the difference calculated beforehand by 125 and the current average amount by 875 and store it in the same variable - EWMA Calculation- EWMA Calculation). When this is concluded, we

make the same verification as when the packet is not from the main flow and check to see if the difference between timestamps is greater or equal to the average. If it is, we check to see if it is directly from a host and if it is, we, like last time, know that there is a problem in the main flow, and we need to utilize another way of communication.

Additionally, we activate a variable present in the standard metadata to know that even if the previous condition is not met, this is to know that we need to make clones. This latter part is done after we check the UDP header. Here, we verify it for errors, and if none are present, we run the UDP tables. If there are, we do not do anything. Only after all this, we make sure that if the previously set flag is triggered, we need to make clones of the packet. We read from our metadata and check the appropriate variable for it and if it passes it, we check the packet itself to see if it already is a clone. If it is, it will only create an infinite cycle of creating clones that would not only crash the switches but also the hosts and the whole network.

```
1. clone3(CloneType.I2E, I2E_CLONE_SESSION_ID, standard_metadata);
```

We now cover the egress part of the program and justify why we barely utilize it.

But, for the sake of ease of counting packets at the destination, we make a check in the egress and set a part of the packet to a certain value that will be checked in the Python script (`receive.py`) at the receiver. To make all the above changes to the packet, we need to remake the checksums present in it, which is done in the control block that is responsible for it. It is called *MyComputeChecksum* and merely calls the method *update_checksum*, upon both the IPv4 and TCP protocols. It receives as an argument whether or not the header is valid, and then, the fields that need to be taken into consideration, where to put the result into, and finally, which type of algorithm to use. This is done for both protocols sequentially.

```
1. update_checksum(methodThatValidatesTheHeader(),
2.             { field1,
3.               field2,
4.                 … ,
5.               fieldN },
6.             headerSpaceToPutTheChecksum,
7.             TypeOfAlgorithmToUse);
```

To finish, we deparse the packet. This an essential part of the program that needs to be done properly. By this, we meant that if the headers are deparsed in an incorrect order the packet would ultimately malformed and would be discarded at the next arrival.

```
1. packet.emit(headerName);
```

The order we use is data link layer followed by the internet layer, and lastly the transport layer.

3.2.6 Difficulties

Moving Average

First and foremost, we would like to approach the moving average part since it is one of the most important parts present in the code and it is not being done the most regular way.

As was stated before, instead of decimal numbers, we were forced to do matters in reverse, by this we mean that since decimals are not supported, the values that need to be multiplied were also multiplied themselves. Because of that, we also had to multiply the value that would, later on, be compared to it, otherwise, it would be wrong and that whole part of the program would not be working as intended. We feel that this was one of the most lacking parts of the languages we were working with.

Bitwise Checks

Multiple instances of our proposed solution use some form of bitwise checks. We had to check which values to enter directly in a pcap file on Wireshark and as such if anything were to change in the packets themselves, some parts of the proposed solution would not work. There is no other way because the language does not provide any form of access to the inside of TCP options. We know that this is not a good procedure. However, it was needed.

Another aspect that needs improvement is the fact that the EWMA calculation can be too heavy on the switch if it is under stress. However, this is not something that can only be tested in a more specific setup.

Chapter 4

Evaluation of the Proposed Solution

The following sections describe the network scenarios considered. For both scenarios, we varied the packet loss percentage from 10% to 100% at intervals of 10%, and for each of these values, we performed the 10 runs for statistical confidence, making 130 runs.

4.1 First Test Scenario

In this first test scenario, as was indicated in chapter 3.2.1, we employed four switches with three hosts each, connected to a singular switch. All the switches would be connected to each other.

In this scenario, H1 is connected to S1, H2 to S2, and H3 to S3. From H1 to H2, there are two possible flows: H1 - S1 - S4 - S2 - H2 and H1 - S1 - S3 - S2 - H2. Before we start analyzing the results, we must first tell what they are and why we choose them and how they represent the performance of the proposed solution.

We obtained information on how many packets were received at the end host (i.e., H2) to compare it with the number of packets retransmitted using our method. This enabled us to know how many packets arrived through the regular and alternative (i.e., retransmissions) paths.

With that in mind, we also needed to collect information on the number of retransmitted packets. These two go hand in hand with each other.

We will start with the number of packets, that were received at the end host H2.

4.1.1 The number of received packets

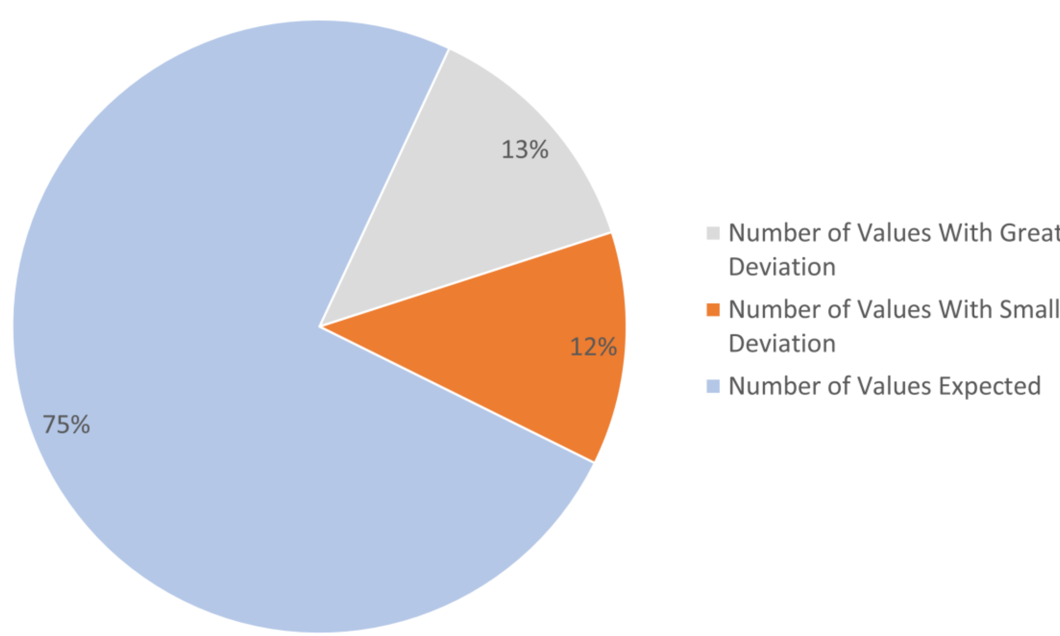


Figure 16 - Distribution of received packets for the first scenario.

First, we start analyzing the number of packets received at H2.

Over the 130 runs, 75% correspond to 97 runs with near 100 received packets (99 to be exact); 13% correspond to 17 runs of received packets with great deviation (i.e., < 95 packets and > 105 packets); and 12% correspond 16 runs of received packets with small deviation (i.e., between >= 95 and < 97 packets and between > 103 and <= 105 packets). This is shown in Fig. 16.

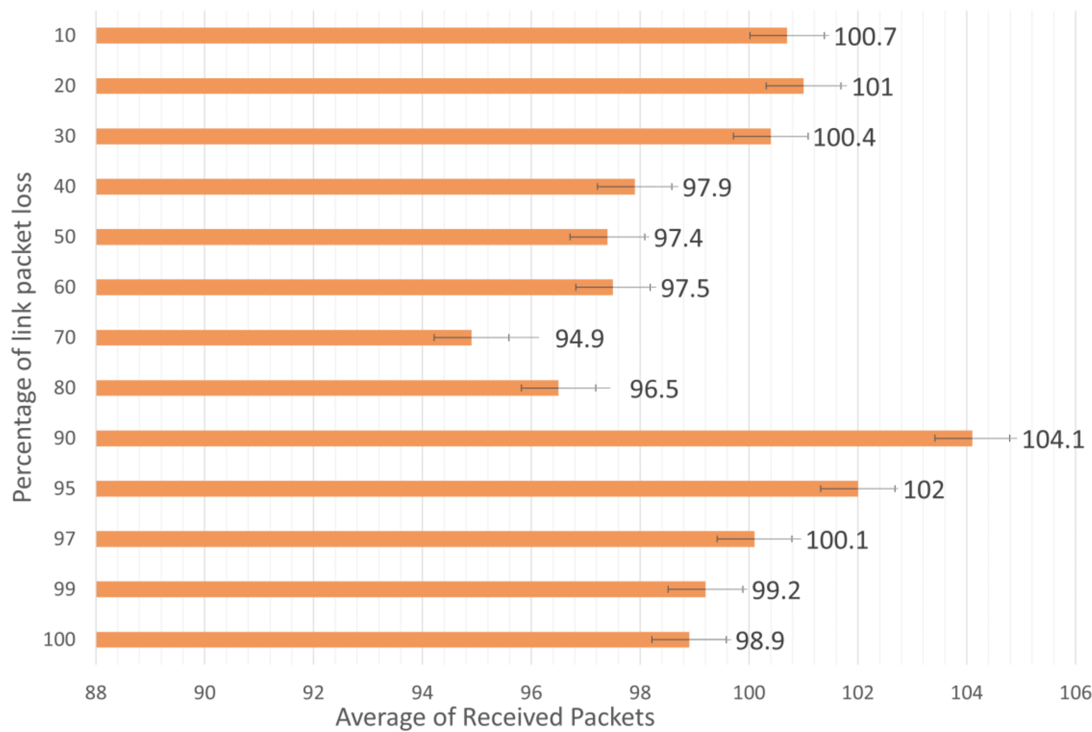


Figure 17- Average number of received packets to H2 for the first scenario

Figure 17 shows the average number of received packets per percentage of link packet loss for the first scenario with 95% confidence intervals. One can conclude that most of the time we are receiving the same number of packets that are sent from the source, that is, 100 packets.

From figure 17, it is possible to see that when the link has a 100% packet loss rate, the number of packets received is expected to be equal to the number of packets retransmitted, since based on our implementation, the first packet is never retransmitted. Also, due to the moving average, it unclear how to handle this packet, i.e., whether it arrives or not at its destination.

For the other cases, this does not apply. As we lower the link loss rate, the number of received packets varies, especially when dealing with the case where there is only a 90% packet loss rate.

For instance, when a cloned packet is received from the alternative route, the packet counter on the main route should stop. If not, more packets than the ones that are originally sent will be received. Link delay can also trigger packet cloning aiming to prevent packet loss.

From link losses between 30% and 90%, one may see that the proposed solution had a rough time dealing with them. Nonetheless, our solution still provides high packet delivery rates.

Throughout the different runs, the lost packet is not always the same. This can deter the moving average hence making the switch not to activate packet cloning, and consequently leading to some packet loss.

Last but not least, from 30% to 10%, the link has a very low packet loss rate. Since the packets are not lost, and as such, their time difference will be rather small, and with that, the switch will not trigger packet cloning. However, there are always some exceptions to the rule, happening most of the time a (one) packet is retransmitted.

Why not two? Because if almost no packets are lost, probabilistically speaking, almost every time 100 ± 1 packets will arrive at the end host, thus justifying our results.

4.1.2 The number of retransmitted packets

Now that we have covered all the information about the received packets we will transition to the information regarding the retransmitted packets.

In that sense, we present figure 18 which shows what types of retransmissions have happened during the first test scenario.

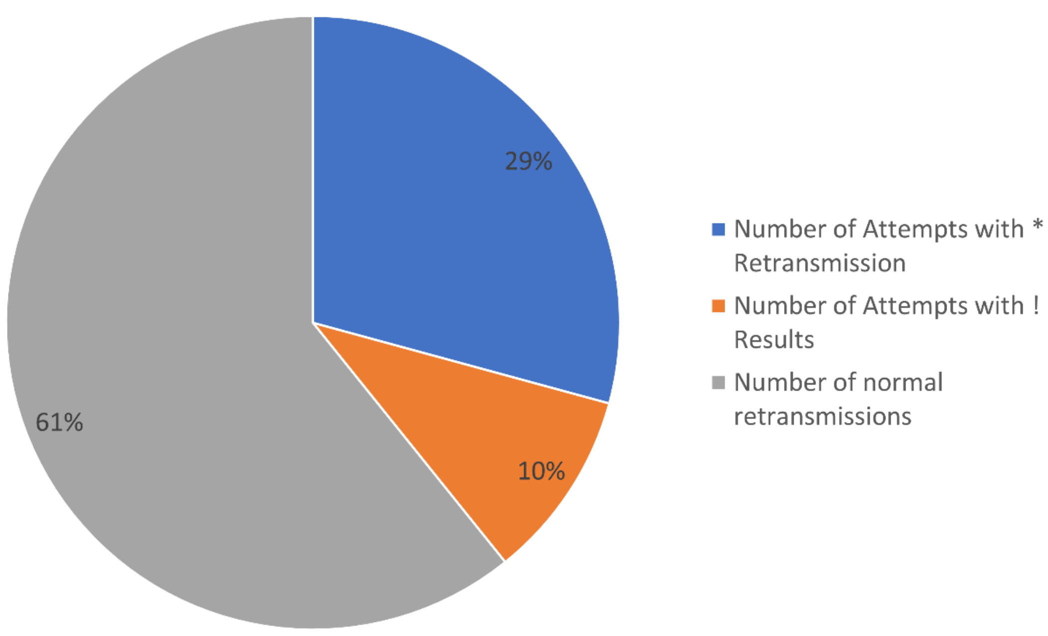


Figure 18 - Distribution of retransmitted packets for the first scenario.

In this scenario and regarding the types of retransmitted packets, we identified the following types: (i) retransmissions with spurious retransmissions in them, (ii) retransmissions that were affected by the problem of packet congestion, or busy CPU cycles, and (iii) the normal retransmissions.

The latter retransmissions, as the name suggests, are those where nothing out of the ordinary happens, i.e., simply cloned packets are sent via the alternative route to their original destination.

We noticed that in the retransmissions with spurious packet retransmissions in them are truly random and either repeat 1 or 2 packets throughout all those sent.

Lastly, there are the retransmissions that have happen when the problem might have occurred. They were simply marked to distinguish them from others.

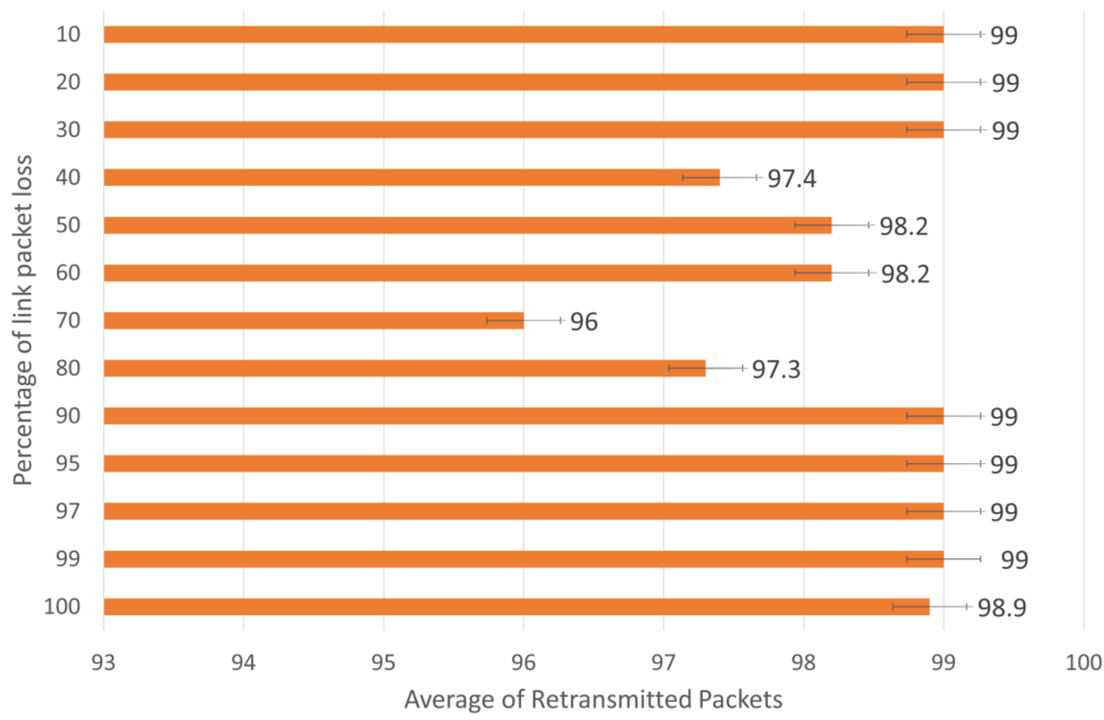


Figure 19 - Average number of retransmitted packets to H2 for the first scenario

Following up the previous graph we present figure 19, which shows even more clearly the values, that have a greater deviation from the expected value.

Here we can see that that in most cases (i.e., 75% of the time) 98 packets were retransmitted. It is also possible to see a higher deviation from the expected value.

4.2 Second Test Scenario

The second scenario consisted of 8 switches (S1 to S8) and 4 hosts (H1 to H4) aiming to represent a more complex network with many more flows (see Figure 17). Here, there are three main flows.

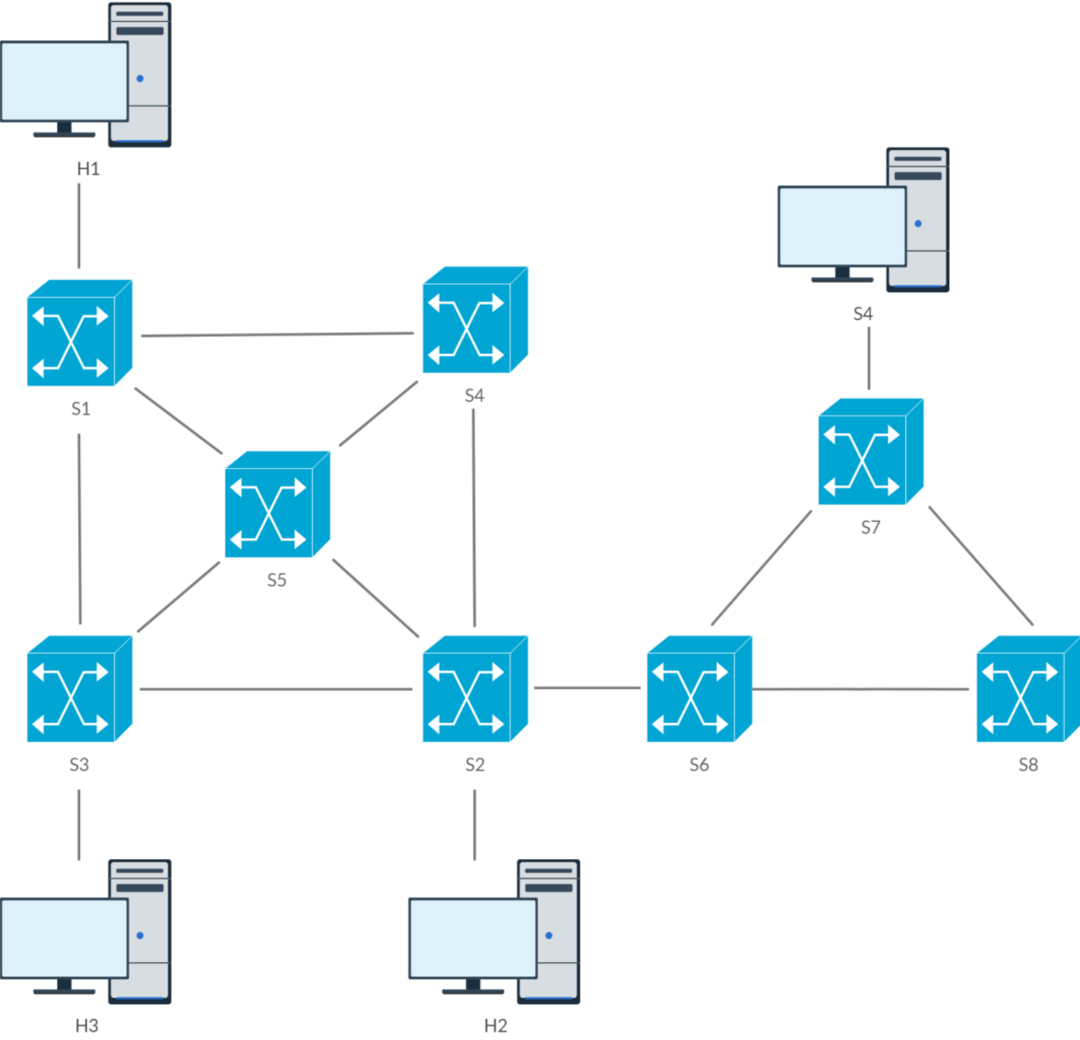


Figure 20 - Second Test Scenario Network

With that in mind, we will also give figure 21 to help illustrate the new flows and how they work in this new network.

Most of them are the same as before however, to test this new network structure we need to provide this new flow that is from H1, going through S4, then to S2, to S6, followed by S7, and finally arriving at H4.

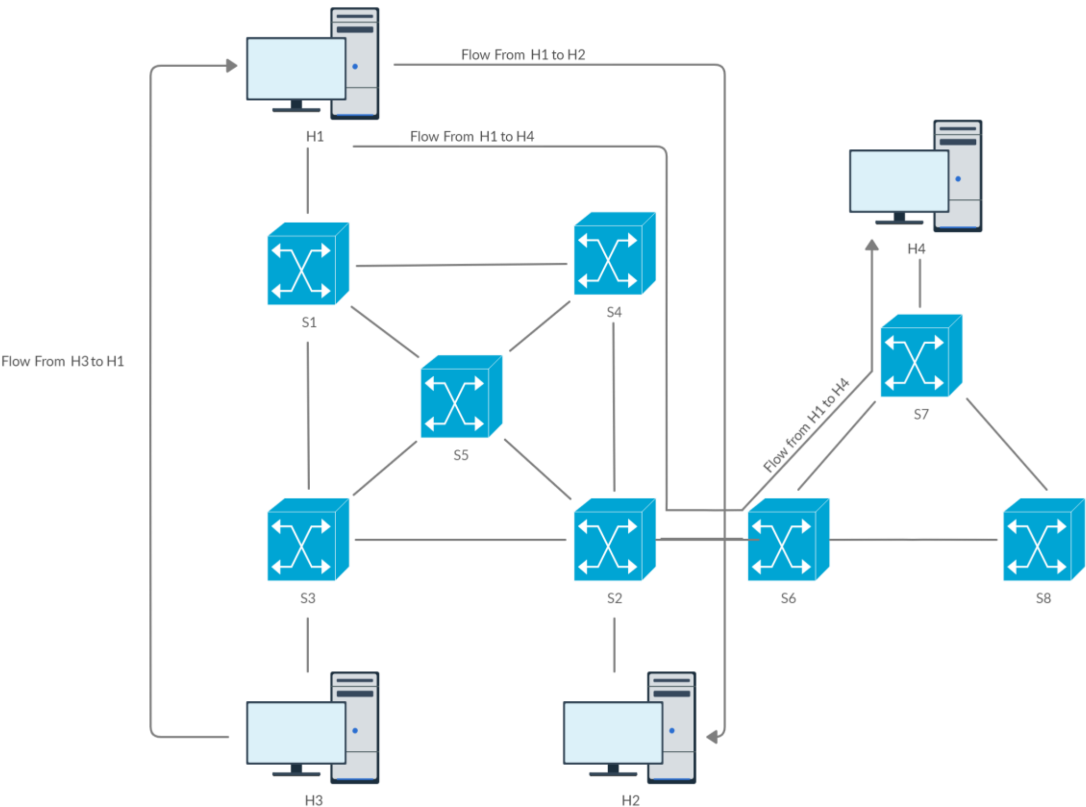


Figure 21 - Second Scenario Network with Flows

Now that we have described the new network we will proceed to show the results that were gotten with it.

Starting with the received packets at H4 they go as the graph shows.

4.2.1 The number of received packets

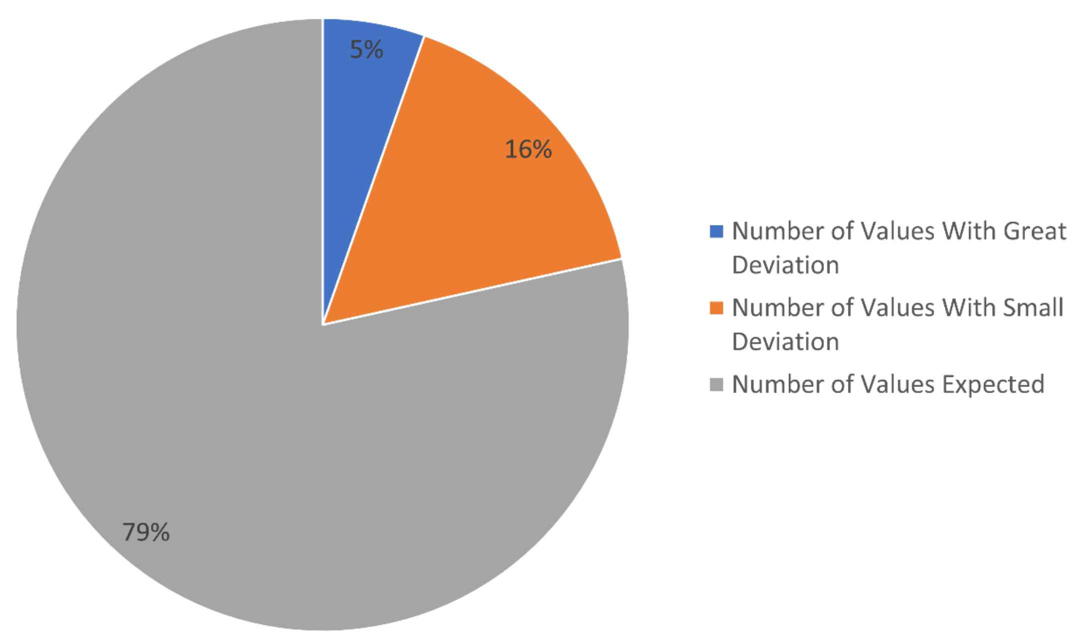


Figure 22 - Distribution of received packets for the second scenario.

Concerning the distribution of received packets at H4 in the second scenario, we noticed an overwhelming lower number of packets with great deviation. 79% of the time, 99 packets were received.

By using a similar approach to the previous scenario, the expected value was between >= 97 and <=103. If between >= 95 and < 97 and between > 103 and <= 105 packets, we considered having a small deviation. Finally, there are the values that fit into the category of < 95 packets and > 105 packets are considered values with great deviation (see Figure 22).

Now covering the number of received packets, with more detail, we will use a bar chart to help us with that in mind we will present the following figure to help illustrate the results.

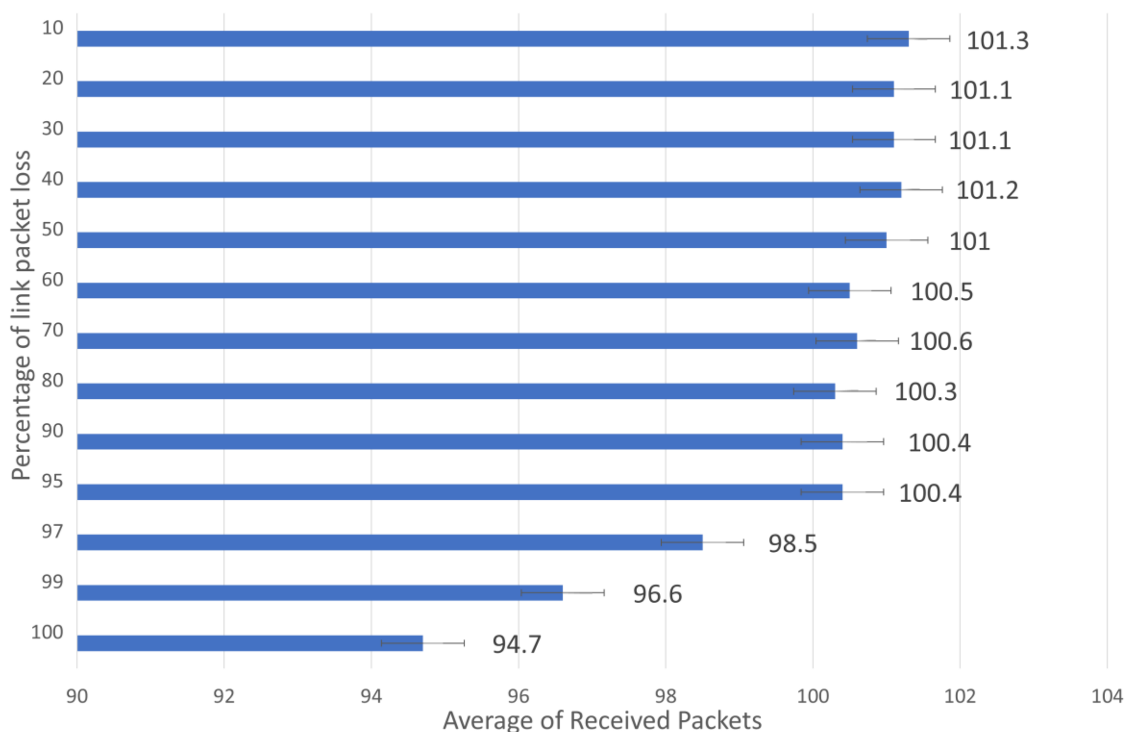


Figure 23 - Average number of received packets at H4 for the second scenario

Figure 23 shows the average number of received packets at H4 for the second scenario with different percentages of packet link loss. Please note that some packet loss within the network can also be attributed to either the switch being congested with other packets or the PC's CPU being under heavy load at the transmission moment.

Nevertheless, we still managed to maintain more than 85% packet successful delivery rate at H4, even in the worse cases.

Similarly, to the first scenario, the moving average issue here is also present. Besides, more packets than those sent are received as we are also counting every retransmitted packet. However, we discarded every packet that comes from the original flow, which arrives after the first retransmitted packet.

Now that we are done with the data regarding the received packet we will move to the retransmitted packets. In that sense, we will provide the next figure that shows the distribution of types of retransmissions in this set of results.

4.2.2 The number of retransmitted packets

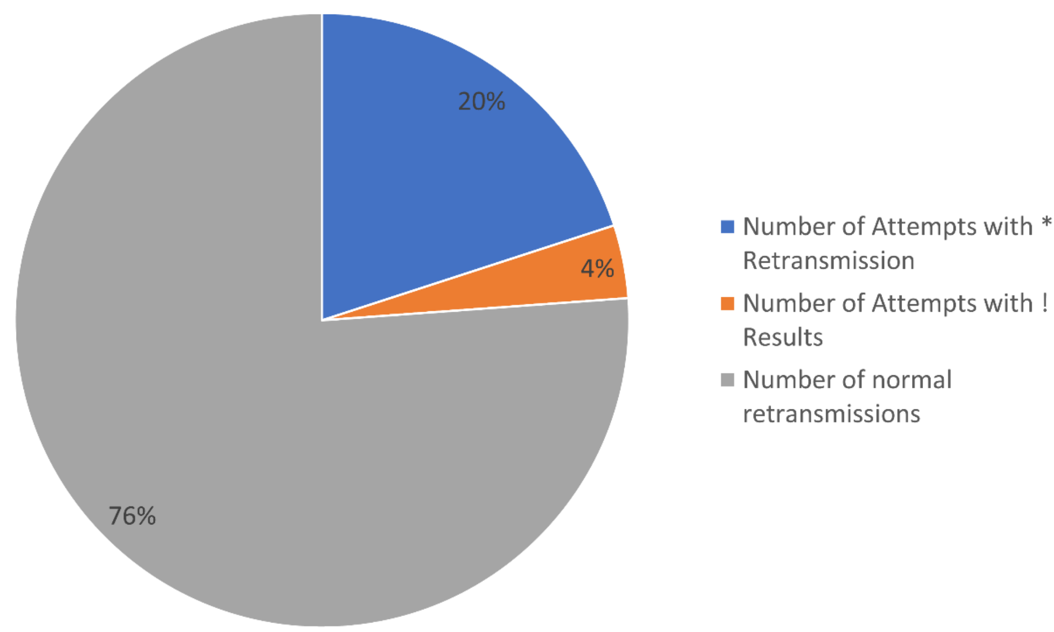


Figure 24 - Distribution of Type of Retransmissions in the Second Scenario

For this scenario, spurious TCP retransmissions represented 20% of all results.

Retransmissions, where there may have been problems in the network or the PC's CPU, may have been under heavy load represented 4% of all results.

Normal retransmissions, in which the switch simply triggers packet cloning, represented 76% of all results.

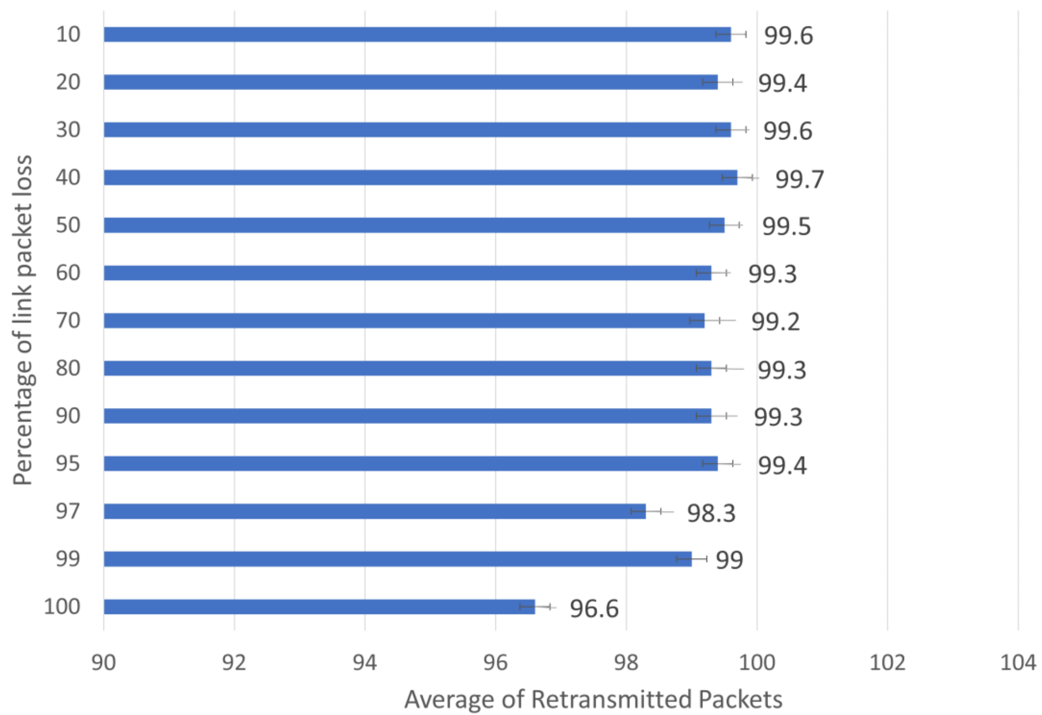


Figure 25 - Average number of received packets at H4 for the second scenario

Figure 25 contains information related to the number of retransmitted packets at the sending host. One can see that the number of retransmitted packets varies between 96 to 100. This was mainly influenced by the fact that we counted every retransmitted packet, but we are only counting regular packets. Specifically, packets that came through the main path are counted if and only if they arrive before the first retransmitted packet.

Lastly, there is also the possibility that the normal packets arrive all after the first retransmitted packet. This can happen because of the way the simulator handles the randomness of loss packets.

Chapter 5

Conclusion and Future Work

In this chapter, we will present the conclusion of our report. Afterward, we will cover the main aspects that can be improved as future work.

5.1 Conclusion

This dissertation proposed a state consistency framework leveraging packet cloning and piggybacking for programmable network data planes using P4.

Two scenarios were considered to evaluate the proposed solution. Performance evaluation has shown that despite link failures higher than 70%, our solution still managed to deliver more than 95% of packets successfully.

Discriminating the results, we can verify that when we introduce bigger networks, our framework starts having a tougher time. Nonetheless, it can still provide 85% packet arrival rate at the destination host. On the other hand, when we used a smaller network it delivers a 92% packet arrival rate at the destination host.

5.2 Future Work

In this section, we will cover everything related to future work that can be done in the proposed solution.

5.2.1 Registers

Now, we consider the registers that are used in the proposed solution. The use of tuples again would only clean the code and make it easier to access the correct spots to insert the data and thus limiting the space for errors.

5.2.2 Offloading and Separation

Offloading the content of the proposed solution to the egress part might become better. However, we have no way of proving this concept. Possibly it can allow for future work to be easier if the different parts of the program are in different areas like different

parts of a program in java are in different files. Here, perhaps separating them could be beneficial as well since there is not a direct way of speaking between ingress and egress. The only way being through the standard metadata both the one created by us and the one provided by the V1 model could be a bottleneck that might create problems if not used carefully and skillfully.

Appendix

A.1 – Retransmitted packets for first scenario

The following table shows the results for the number of retransmitted packets related to the first test scenario.

% of lost packets	1	2	3	4	5	6	7	8	9	10
100	99	99	99*	99*	99	98	99*	99	99	99
99	99	99	99	99*	99	99	99	99*	99	99
97	99	99	99	99	99*	99	99**	99	99**	99
95	99	99	99	99	99	99	99	99	99*	99
90	99	99*	99	99	99	99	99	99	99	99
80	99*	93!	99!	99	93!	99	99*	99!	99	94!
70	91	99!	99	95	91	99	99!	99*!	99	89
60	99*	99	99!	99	99!	99	99	91	99	99
50	99*	99	99	99!	99	99!	95	99	95	99**
40	99	99	99	99**	99*	93	99!	99*	89	99*
30	99*	99	99*	99	99	99	99	99	99	99
20	99	99	99	99*	99	99*	99	99	99**	99
10	99	99	99**	99*	99*	99	99	99*	99	99

Table 1 - Table for the Number of Retransmitted packets from the Switch S1 during the first set of results

Legend:

* Note this one had one spurious retransmission

** Note this one had two spurious retransmissions

! Please note that some packet loss within the network can also be attributed to either the switch being congested with other packets or the PC's CPU being under heavy load at the transmission moment.

A.2 - Distribution of retransmitted packets for the first scenario

The following table shows the results for the distribution of types of retransmitted packets related to the first test scenario

Table 2 - Totals for Retransmitted Packets during the first set of results

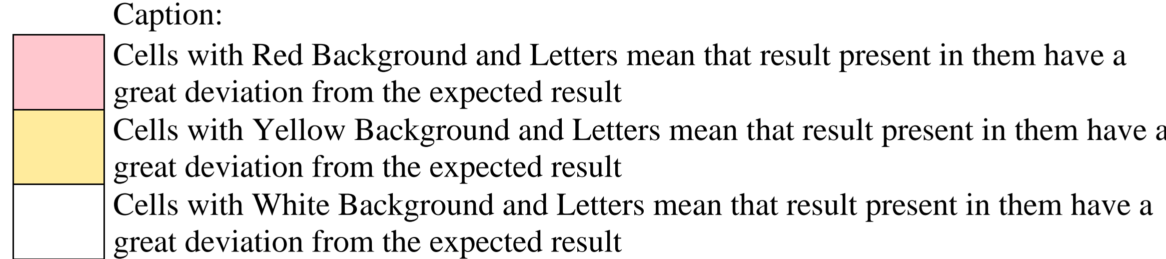
Number of Attempts with * Retransmission	38
Number of Attempts with ! Results	13
Number of normal retransmissions	79
Total	130

A.3 - Received packets for first scenario

The following table shows the results for the number of received packets related to the first test scenario

% of lost packets	1	2	3	4	5	6	7	8	9	10
100	99	99	99	99	99	98	99	99	99	99
99	99	100	99	99	99	100	99	99	99	99
97	100	99	99	102	100	103	101	99	99	99
95	108	103	99	99	106	99	99	108	99	100
90	104	99	113	99	99	110	111	99	99	108
80	100	93	91	99	93	100	99	97	99	94
70	92	97	100	95	92	100	93	91	99	90
60	99	99	93	99	91	101	101	92	100	100
50	99	100	100	91	100	93	96	100	95	100
40	101	99	100	100	101	93	92	101	90	102
30	100	100	101	100	101	101	101	99	101	100
20	102	100	101	101	101	101	102	101	101	100
10	101	99	100	101	101	101	101	101	101	101

Table 3 – Table for the Number of Received Packets at the end host H2



Caption:

Cells with Red Background and Letters mean that result present in them have a great deviation from the expected result

Cells with Yellow Background and Letters mean that result present in them have a great deviation from the expected result

Cells with White Background and Letters mean that result present in them have a great deviation from the expected result

A.4 - Distribution of received packets for the first scenario

The following table shows the results for the distribution of types of received packets related to the first test scenario

Number of Values With Great Deviation	17
Number of Values With Small Deviation	16
Number of Values Expected	97
Total	130

Table 4 - Totals for Received Packets during the first set of results

A.5 - Retransmitted packets for the second scenario

The following table shows the results for the number of retransmitted packets related to the second test scenario.

% of lost packets	1	2	3	4	5	6	7	8	9	10
100	99	100	85	99	88	99*	99	99*	99!	99!
99	99	99	99	99	99	99!	99!	99	99	99!
97	96	99	100	99	99*	99*!	99*	94	99	99
95	99*	99	100*	100*	100	99	99	99	100	99
90	100	99	99	100	99	99	100	99	99*	99
80	99*	99	100	99	99	100	99*	99	99*	100
70	99	99	99*	99	99	100	99	99	100	99
60	99*	100	99	99	99	99	100	100	99	99
50	99	100*	100	99	100	99***	100	99	99	100
40	100	99	100*	99	100	100	99	100	100	100
30	100**	100*	99*	100	100	99	99	100**	100	99
20	99*	100	99	100	99*	100	99	99*	99	100
10	99	100	99	99*	100*	100	100	100	100	99

Table 5 . Table for the Number of Retransmitted packets from the Switch S1 during the second set of results

Legend:

* Note this one had one spurious retransmission

** Note this one had two spurious retransmissions

*** Note this one had three spurious retransmissions

! Please note that some packet loss within the network can also be attributed to either the switch being congested with other packets or the PC's CPU being under heavy load at the transmission moment.

A.6 - Distribution of retransmitted packets for the second scenario

The following table shows the results for the distribution of types of retransmitted packets related to the second test scenario

Number of Attempts with * Retransmission	26
Number of Attempts with ! Results	5
Number of normal retransmissions	99
Total	130

Table 6 - Totals for Rertransmitted Packets during the second set of results

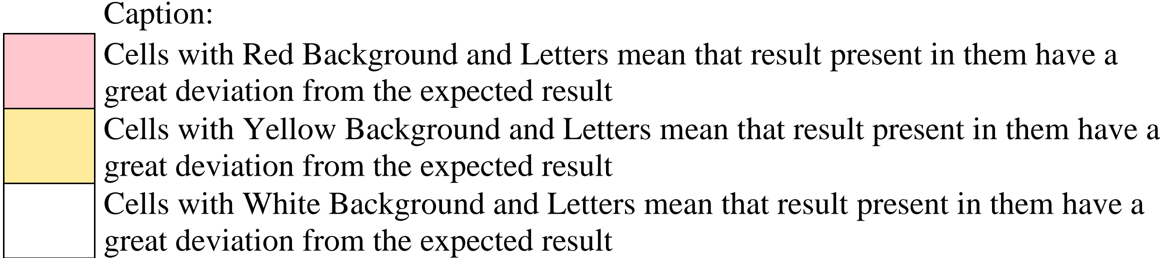
A.7 - Received packets for second scenario

The following table shows the results for the number of received packets related to the second test scenario

% of lost packets	1	2	3	4	5	6	7	8	9	10
100	99	100	85	99	88	99	99	99	86	93
99	100	101	99	99	100	89	88	99	99	92
97	96	101	101	100	100	88	99	97	102	101
95	100	100	101	101	101	100	100	100	101	100
90	101	100	100	101	101	100	101	100	100	100
80	100	100	101	100	100	101	100	100	100	101
70	102	100	100	100	101	101	101	100	101	100
60	100	101	100	101	100	100	102	101	100	100
50	100	101	102	100	101	101	102	101	100	102
40	102	101	102	101	101	101	100	102	101	101
30	102	102	100	101	102	100	100	101	102	101
20	101	101	101	102	101	102	101	100	101	101
10	101	102	100	100	102	101	102	102	101	102

Table 7 - Table for the Number of Received Packets at the end host H4

Caption:



A.8 - Distribution of received packets for the second scenario

The following table shows the results for the distribution of types of received packets related to the second test scenario

Number of Values With Great Deviation	7
Number of Values With Small Deviation	21
Number of Values Expected	102
Total	130

Table 8 - Totals for Received Packets during the second set of results

Bibliography

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, 2014.

[2] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A Survey on Software-Defined Networking," *IEEE Commun. Surv. Tutorials*, 2015.

[3] P. J. Roig, S. Alcaraz, and K. Gilly, "Formal specification of spanning tree protocol using ACP," *Elektron. ir Elektrotechnika*, 2017.

[4] J. John and E. Katz-Bassett, "Consensus routing: The Internet as a distributed system," *Proc. Symp. Networked Syst. Des. Implement.*, 2008.

[5] E. C. Molero, S. Vissicchio, and L. Vanbever, "Hardware-accelerated network control planes," *HotNets 2018 - Proc. 2018 ACM Work. Hot Top. Networks*, pp. 120–126, 2018.

[6] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, and M. Conti, "A Survey on the Security of Stateful SDN Data Planes," *IEEE Commun. Surv. Tutorials*, vol. 19, no. 3, pp. 1701–1725, 2017.

[7] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[8] "p4c/v1model.p4 at master · p4lang/p4c · GitHub." [Online]. Available: https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4. [Accessed: 29-Sep-2020].

[9] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "RFC 3746: Forwarding and Control Element Separation (ForCES) Framework." pp. 1–40, 2004.

[10] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," *Comput. Commun. Rev.*, vol. 44, no. 4, pp. 377–378, 2015.

[11] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM'11*, 2011, pp. 254–265.

[12] "Open vSwitch," 2013. [Online]. Available: https://www.openvswitch.org/.

[Accessed: 04-Dec-2019].

[13] ONF, “OpenFlow Switch Specification Version 1.5.1,” *Current*, vol. 1, p. 35, 2015.

[14] S. B. Davidson, H. Garcia-Molina, and D. Skeen, “Consistency in Partitioned Network,” *ACM Comput. Surv.*, 1985.

[15] S. Gilbert and N. Lynch, “Perspectives on the CAP Theorem,” *Computer (Long. Beach. Calif).*, vol. 45, no. 2, pp. 30–36, 2012.

[16] C. Nate Foster Cornell Jennifer Rexford Princeton Cole Schlesinger Princeton David Walker Princeton, “Abstractions for Network Update,” pp. 323–334, 2012.

[17] S. Landis and S. Maffeis, “Building reliable distributed systems with CORBA,” *Theory Pract. Object Syst.*, vol. 3, no. 1, pp. 31–43, 1997.

[18] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs, “R-BGP: Staying Connected In a Connected World,” *NSDI’07 Proc. 4th USENIX Conf. Networked Syst. Des. Implement.*, p. 14, 2007.

[19] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, “Openstate: Programming platform-independent stateful openflow applications inside the switch,” *Computer Communication Review*, vol. 44, no. 2. pp. 44–51, 2014.

[20] S. Pontarelli *et al.*, “FlowBlaze: Stateful Packet Processing in Hardware This paper is included in the Proceedings of the,” *Proc. NSDI*, pp. 531–547, 2019.

[21] P. Bosshart *et al.*, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN,” *SIGCOMM 2013 - Proc. ACM SIGCOMM 2013 Conf. Appl. Technol. Archit. Protoc. Comput. Commun.*, pp. 99–110, 2013.

[22] A. Sivaraman *et al.*, “Packet transactions: High-level programming for line-rate switches,” *SIGCOMM 2016 - Proc. 2016 ACM Conf. Spec. Interes. Gr. Data Commun.*, pp. 15–28, 2016.

[23] T. R. Henderson and G. F. Riley, “Network Simulations with the ns-3 Simulator,” *Proc. Sigcomm*, p. 527, 2006.

[24] Mininet, “Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet,” *Mininet.Org*, p. www.mininet.org, 2014.

[25] S. Y. Wang, C. L. Chou, and C. M. Yang, “EstiNet openflow network simulator and emulator,” *IEEE Commun. Mag.*, vol. 51, no. 9, pp. 110–117, 2013.

[26] The P4 Language Consortium, “P4 16 Language Specification v1.2.1,” p. 129, 2018.

[27] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, “RFC 7323 TCP Extensions for High Performance,” 2014.

[28] J. Mahdavi, “RFC: TCP Selective Acknowledgment Options,” 2018.

[29] “Transmission Control Protocol (TCP) Parameters.” [Online]. Available: https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml. [Accessed: 27-Nov-2020].

[30] “RFC 793 - Transmission Control Protocol.” [Online]. Available: https://tools.ietf.org/html/rfc793. [Accessed: 27-Nov-2020].