Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2014

André Alexandre Nunes Martins

Critical Ethernet baseada em OpenFlow

Critical Ethernet based on OpenFlow

Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2014

André Alexandre Nunes Martins

Critical Ethernet baseada em OpenFlow

Critical Ethernet based on OpenFlow

"*If you don't build your dream, someone else will hire you to help them build theirs.*"

— Dhirubhai Ambani

Universidade de Aveiro
2014

Departamento de Eletrónica,
Telecomunicações e Informática

André Alexandre Nunes Martins

Critical Ethernet baseada em OpenFlow

Critical Ethernet based on OpenFlow

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Rui Luis Andrade Aguiar, Professor associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Pedro Alexandre Sousa Gonçalves, Professor adjunto na Universidade de Aveiro.

Dedico este trabalho ao meu pai pelos incentivos que me levaram a questionar tudo à minha volta, à minha mãe por me ter impelido para a informática e ao meu irmão mais novo por me ter aberto os olhos.

o júri / the jury

presidente / president

Prof. Doutor José Rodrigues Ferreira da Rocha
Professor Catedrático da Universidade de Aveiro

vogais / examiners committee

Prof.ª Doutora Marília Pascoal Curado
Professora Auxiliar da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Prof. Doutor Rui Luís Andrade Aguiar
Professor Associado com Agregação da Universidade de Aveiro (orientador)

agradecimentos /
acknowledgements

I would like to thank professor Rui Aguiar for transmitting his vast knowledge to me over my course and for giving the opportunity of showing me that the first approach to solve a problem is not necessarily the best. To professor Pedro Gonçalves for his support and mostly patience to put up with me. To Daniel Corujo for giving me the guidance to write this dissertation. To Ricardo Marau for the discussions and ideas applied for the implementations developed. To all ATNoG members that helped me directly and indirectly with the creation of this dissertation.

I want to thank the OpenDaylight community, specially Colin Dixon for the knowledge he could offer me. To the Open vSwitch project, specially Ben Pfaff for his availability.

To my family, friends and colleges that fought by my side to finish this stage of my life. A special thanks to Micael for loving kerning as much as I do. Last, but definitely not least, to Sara for supporting me over these last years.

Palavras Chave

SDN, *OpenFlow*, *fast fail-over*, *fail-safe*, tolerância a falhas, redundância, *cloud*, *OpenDaylight*.

Resumo

Hoje em dia coloca-se um valor imenso em redes *Ethernet*, especialmente para operações em *data centers* que fornecem serviços na *cloud* ou em enormes infraestruturas de rede em geral. No entanto, nem sempre é possível existir e garantir comunicações a 100% devido ao facto de a redundância em *Ethernet* ter sido considerada sempre como um problema não resolvido, tendo em conta a grande quantidade de recursos de rede a serem geridos. Ao longo da história têm sido desenvolvidas diversas soluções que tentaram resolver este problema, apenas para enfrentarem o falhanço em fornecer os requisitos adequados.

Software-defined Networking (SDN) é um paradigma inovador e um mecanismo dinâmico e configurável que traz uma natureza programável que permite a implementação de soluções que possam, finalmente, resolver os problemas identificados. Através do uso de interfaces abertas programáveis, o controlo e gestão do comportamento da rede está a tornar-se mais fácil e menos propenso a erros.

O objetivo principal desta dissertação foi a implementação e avaliação de uma solução baseada em SDN à prova de falhas para comunicações críticas, portanto para gestão de falhas em tecnologias *Ethernet* redundantes num cenário típico de gestão de *data centers*.

Esta dissertação apresenta a solução desenvolvida e as principais fases da sua implementação. A solução implementada utiliza uma rede redundante L2 e um controlador SDN para calcular a topologia da rede. A solução faz uso de extensões para o protocolo *OpenFlow* e módulos do controlador *OpenDaylight*.

Durante a fase de avaliação, diferentes cenários foram testados onde ocorreram mudanças na topologia. Os resultados da avaliação mostram que a solução proposta se comporta de forma satisfatória sempre que uma ligação falha, obtendo perda de pacotes nula. Para concluir, a solução mostra-se promissora para as operações em *data centers* críticas tendo em conta o tempo de adaptação obtido nas avaliações.

Keywords SDN, OpenFlow, fast fail-over, fail-safe, fail-tolerant, redundancy, cloud, OpenDaylight

Abstract

Nowadays, we put an immense value on Ethernet networks, especially for data center operations empowering cloud environments or huge network infrastructures in general. However, it is not always possible to bring 100% up-time communications since redundancy in Ethernet has always been an unresolved problem, considering the large amount of network resources to be managed. Through history there have been many developed solutions that tried to solve this issue, only to fail in providing the proper support.

Software-defined Networking (SDN) is a novel paradigm and a dynamic and configurable mechanism that brings a programmable nature for developers to implement solutions that may finally solve the identified issues. Via the use of programmable open interfaces, the control and management of network behavior is becoming easier and less error prone.

The main objective of this dissertation was the implementation and evaluation of a fail-safe SDN-based solution for critical communications, therefore for fault management in redundant Ethernet technologies on a typical data center management scenario.

This dissertation presents the developed solution and the main phases of its implementations. The implemented solution uses a redundant L2 network and a SDN controller to calculate the network topology. The solution makes use of extensions to both the OpenFlow protocol and OpenDaylight controller's modules.

During the evaluation stage, different scenarios were tested where topology changes occur. The evaluation results show that the proposed solution behaves satisfactorily whenever a link fails, obtaining none packet loss. To conclude, the solution shows to be promising for critical data center operations concerning the adaptation time obtained.

Contents

List of Figures

List of Tables

Glossary

ACL Access Control List

AD-SAL API-Driven SAL

AMQP Advanced Message Queuing Protocol

BGP Border Gateway Protocol

BPDU Bridge Protocol Data Unit

CMTS Cable Modem Termination System

CSMA/CD Carrier Sense Multiple Access with Collision Detection

DAG Directed Acyclic Graph

ECMP Equal-cost multi-path routing

ECT Equal Cost Tree

ECU Engine Control Unit

GRE Generic Routing Encapsulation

IaaS Infrastructure as a Service

ICMP Internet Control Message Protocol

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

IPSec Internet Protocol Security

IS-IS Intermediate System to Intermediate System

LAG Link Aggregation Group

LAN Local Area Network

MAC Media Access Control

MC-LAG Multichassis Link Aggregation

MD-SAL Model-Driven SAL

MSTP Multiple Spanning Tree Protocol

NETCONF Network Configuration Protocol

NIB Network Information Base

NOS Network Operating System

OSI Open Systems Interconnection

OS Operating System

OVSDB Open vSwitch Database Management Protocol

PCEP Path Computation Element Communication Protocol

PVST Per-VLAN Spanning Tree

QoS Quality of Service

RBridge Routing Bridge

RCP Routing Control Platform

RFC Request for Comments

RIB Routing Information Base

RPFC Reverse Path Forwarding Check

RSTP Rapid Spanning Tree Protocol

SAL Service Abstraction Layer

SANE Security Architecture for Enterprise Networks

SDN Software-defined Networking

SNMP Simple Network Management Protocol

SPB Shortest Path Bridging

SPT Shortest Path Tree

STA Spanning Tree Algorithm

STP Spanning Tree Protocol

TCP Transmission Control Protocol

ToS Type of Service

TRILL Transparent Interconnection of Lots of Links

TTL Time To Live

UDP User Datagram Protocol

VCS Virtual Cluster Switching

VLAN Virtual Local Area Network

VM Virtual Machine

VXLAN Virtual Extensible LAN

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

The value we put on today's communications causes them, now more than ever, to be more reliable, safe and have almost 100% up time. In order to achieve that criteria the evolution of current paradigms, technologies and standards must be performed. On a company where its main income is directly connected with the amount of time its services are online, it is essential that the internal network failures are covered by backups. Usually, those backups are based on existing technologies without the freedom of changing a standard to suit each company's specific needs. With the advance in the performance of today's CPUs it makes no sense to continue having a network adjusting itself on a distributed way when a company may want a real-time control of it. In addition with the technology currently deployed on the enterprise networks it is not always possible to bring a 100% uptime due to multiple factors such as equipment failure, security incidents (directly or indirectly related with the network of the organization), natural disasters, among other incidents.

Supposing a company has a network similar to the one illustrated on Figure 1.1, in order for it to have redundancy on a data link level it is necessary to have multiple paths to reach the primary and backup machines. For over fifteen years it has been used the same technology to achieve that redundancy. Contrary to what happened fifteen years ago, the bandwidth of each link increased significantly over the years and the requirements back then were different from the today's. Nothing lasts forever and it is relevant to make sure in the case an equipment failure, the communications conducted by that equipment do not fail.

Network protocols are conceptually divided by seven layers: physical; data link; network; transport; session; presentation and application layer. The first layer, the physical layer, defines the electrical, physical and optical specifications as well the bit signaling for the data transmission. One of the protocols from the second layer, the data link layer, is studied on this dissertation: the IEEE 802.3

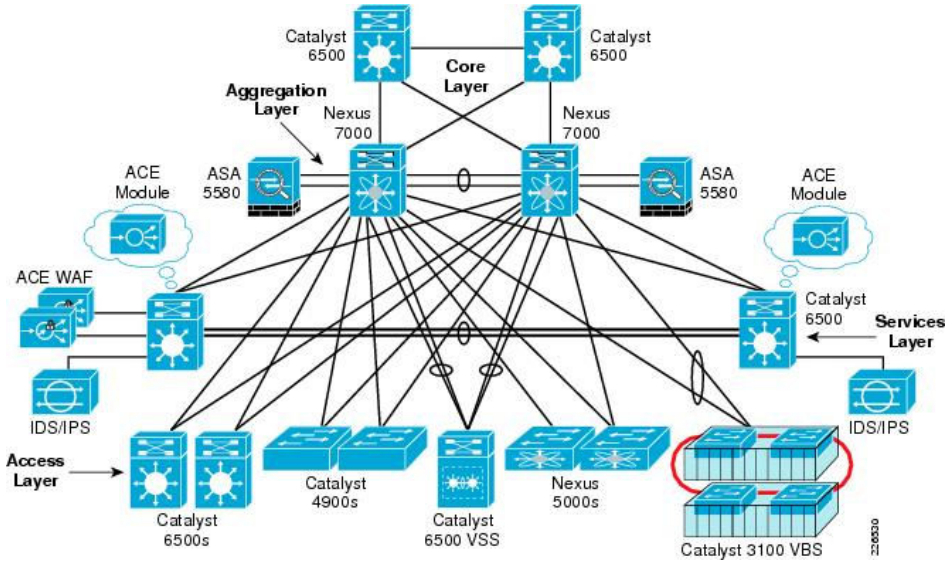


Figure 1.1: A network topology pattern for data centers [1].

(often called Ethernet). This protocol defines how the information is exchanged between two directly connected nodes. However, when it was designed, it was not considered a mechanism for forwarding such as hop count. Thus, it is impossible having redundancy on Ethernet without dealing with the so called broadcast radiation. To deal with this problem, in 1985 appeared an algorithm called Spanning Tree Algorithm (STA) [1]. Later in 1990, that algorithm was used on a protocol called Spanning Tree Protocol (STP) standardized as IEEE 802.1D. On Figure 1.1, the communication between the access layer and the services layer would be made through the aggregation layer. STP creates a topology tree without loops by setting some ports of the switches on a blocked state causing the traffic to go through one of the links attached on an available non-blocking port. In case the main link fails, and since the communication has a backup link, the topology reconfigures itself and the communication between the two machines continues after the spanning tree convergence. Unfortunately, that reconfiguration takes a few dozens of seconds to complete during which, no communication is made on all the topology. Some improvements were made and the Rapid Spanning Tree Protocol (RSTP) appeared in 2001 as a substitute for STP. One of the RSTP improvements focuses on reducing those dozens of seconds in the reconfiguration process to a couple of seconds. Although there were great improvements over those years, there is still a problem on those protocols: they create a tree-based network topology, and there is not a full use of all the equipment deployed on the topology, which consists in a lack of overall efficiency on the network usage. In addition, a single machine transfers all the traffic, designated by root bridge, causing a bottle neck in the network. Besides those problems, there are still those couple of seconds of offline time which are taken for the topology to reconfigure itself after a change. All the previously defined issues do not help in providing a 100% uptime between two or more machines in critical communications.

[1]Image available: `http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_3_0/dc_serv_pat.html`

Multiple manufacturers have created their own STP and RSTP alternatives such as the proprietary protocols QFabric [2], Virtual Cluster Switching (VCS) [3] and FabricPath [4] to overcome those problems and bring some new features that were not possible with the existing protocols from IEEE. This brought another problem such as cross-manufacturing issues: it was not possible to have two products from different brands using another company's algorithm.

Over the years many efforts have been undertaken to develop network management solutions to follow a software controlled network management approach. Since 2005 some projects have appeared as a result of those efforts such as: 4D [5] [6], Routing Control Platform (RCP) [7], Security Architecture for Enterprise Networks (SANE) [8] and Ethane [9]. This led to the creation of a new concept, created by Martìn Casado in 2008, creator of SANE and Ethane, called Software-defined Networking (SDN) [10]. Although the concept is new, it is difficult to create a definition for it since its technical principal is old. Martìn Casado brought it as a new organizing principle with proving concepts [11].

With SDN came the first technology from it called OpenFlow. Its main purpose was to help developers and investigators create and test new protocols on a live network [12]. This brought the definition of two networking planes, the control plane and the forwarding/data plane. The data plane processes packets in the forwarding unit, depending on its layer, such as a layer 2 switch for Ethernet or a router for Internet Protocol (IP). The control plane is more complicated since the control of a network goes from its routing passing by the isolation and finishing in traffic engineering. Furthermore, it is not possible to perform different actions for each individual packets [13]. However OpenFlow brought new possibilities to change and control a network topology in real time from a central controller providing a new paradigm to network administrators. With that control it is possible to have a network that controls itself taking more factors into consideration as well as making decisions faster than a human and legacy technologies such as STP-based protocols.

This dissertation describes the implementation of a fault-tolerant SDN-based resource management solution.

1.2 OBJECTIVES

The main goal of this dissertation is to develop a solution providing, in an SDN, a free-loop layer 2 topology with redundancy. The objectives of this dissertation were:

- Study of current protocols deployed for redundancy;
- Familiarization with SDN concepts;
- Study the OpenFlow protocol;
- Present some of the vast capabilities of an SDN;

- Create an implementation of an experimental prototype that provides a fault-tolerant topology on an SDN topology.

1.3 CONTRIBUTIONS

The work presented on this dissertation has been submitted to the scientific community.

It was defined a fault-tolerant mechanism for network management. Taking advantage on the OpenFlow, this mechanism allows for a fail-safe SDN topology.

The source code for the plugins implemented for the OpenDaylight controller are available online in [2].

The implemented fail-safe Ethernet solution was accepted by the 16th International Telecommunications Network Strategy and Planning Symposium committee (Funchal, Madeira, 2014) [14].

This work was also submitted in the 5th edition of the Fraunhofer Portugal Challenge in the area of "Autonomic Computing - smarter devices, less configuration and maintenance (remote & self-management, configuration and control)".

1.4 STRUCTURE

This document is divided in six chapters. The current chapter outlines the motivation for the work of this dissertation, on redundant communication for Ethernet using OpenFlow, as well as providing information on the contributions already made. The remaining chapters provide information as follows:

- Chapter 2: Presents the current work related to redundancy on Ethernet. It is described an overview of the Ethernet's history and the current protocols developed and applied in the industry such as STP, RSTP and MSTP. The emerging protocols, such as TRILL and SPB, are also considered;
- Chapter 3: Describes the Software-defined Networking concept and its history. It is focused on the architecture of an SDN topology and the primary key components of those type of networks;
- Chapter 4: Starts by enumerating the requirements of an implementation to be developed for SDN in order to solve the primary objective of this dissertation. Later, it describes the key aspects of the solutions developed and the main differences between them;
- Chapter 5: Illustrates and describes the tests made in order to evaluate the different developed solutions. This chapter ends with an overall conclusion about which is the ideal scenario for each developed solution;

[2] https://github.com/aanm/multipath-openflow

- Chapter 6: Wraps up the dissertation with a brief overview of the concepts discussed, the main results obtained and an indication of a direction for future work.

CHAPTER 2

Defining redundancy on Ethernet

Ethernet was designed to become a protocol for point-to-point communications. Later was concluded that was necessary to have more machines connected in order to have more computational power. To connect those machines it was necessary to add bridges, a layer 2 forwarder packet, creating a Local Area Network (LAN). While LANs keep growing it was more difficult to control the communication's collisions, although the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) method was present, it was not enough causing instability in the network. Having multiple bridges laid on a respective way, as one can seen in Figure 2.1, where only a path exists between any pair of machines, it was possible to increase the number of bridges but it would also increase the number of failures over time.

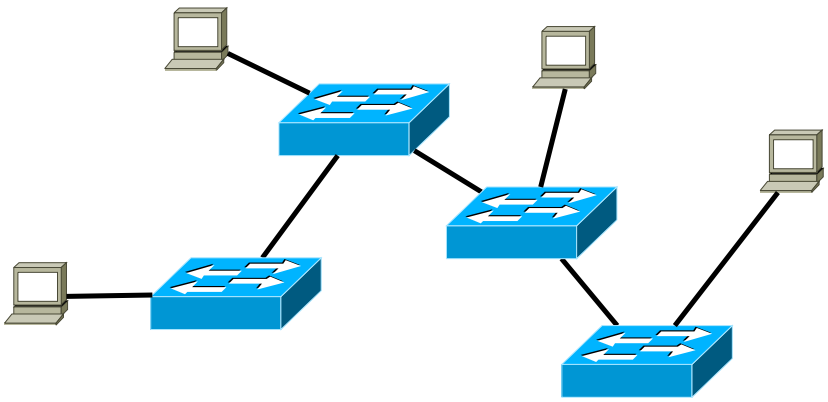


Figure 2.1: Single path topology for all connected hosts.

Unfortunately, Ethernet was not enabled with any mechanism to measure the distance between two network nodes. For example, IP protocol from Open Systems Interconnection (OSI) model third

layer, has a hop count value that decreases by one unit for each router where that packet has passed. On Ethernet there is not such option and with the LANs increasing in complexity, it was urgent to build a mechanism to control Ethernet packets on redundant networks, avoiding the packets to be on the topology until resource exhausting, called broadcast storms.

Being essential to have redundancy for multiple destinations, there is another problem besides broadcast storms. On Figure 2.2 there are two machines, machine A directly connected to switch 1 and machine B directly connected to switch 2. Supposing machine 1 is sending information to machine 2, switch 1 does not know where it should transmit those packets and floods them to all ports except the one where the packets came from. Switch 2 will receive packets from port 1 and have the same behavior as switch 1, sending those packets to all ports except the one it received. Supposing switch 3 knows the port to the destination's packets, it sends it to port 3.

While the flooding process occurred on switch 1, the original packet was multiplied by the number of ports (two, in this case) connected to the topology. This caused one of the packets to pass through switch 2 and the second to be sent directly to switch 3. The problem occurs when the second packet reaches the receiver because it might cause some confusion by receiving the double amount of the information sent.

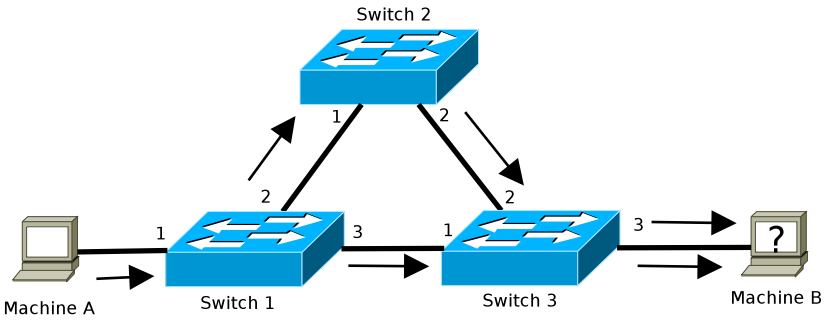
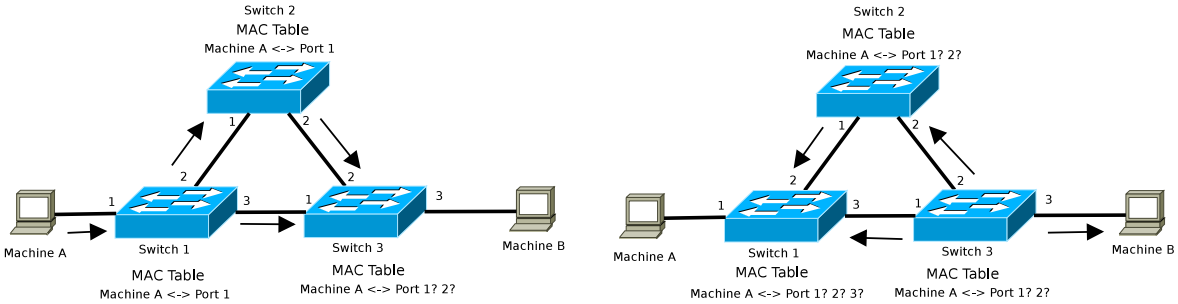


Figure 2.2: Redundant topology where machine B receives the double of the information sent by machine A.

Last but not least, there is still a problem related with the MAC database from the switches called MAC database instability. On the same example from Figure 2.3b, supposing switch 3 does not know where machine 3 is, it will send the same packet through all ports. When switch 1 receives it, it will learn that machine 1 is on port 3, which is not true. When another packet is received from machine 1, switch 1 will learn that machine 1 is on port 1 and not on port 3. This will repeat every time a packet is sent from machine 1, causing the MAC database from switch 1 to become unstable.

In the following sections are explained some algorithms and protocols developed to solve these problems. In the end of this section is recapitulated in form of a table the algorithms listed and some of their major characteristics.



(a) A packet sent from machine 1 passing through switch 1 and switch 2, reaching switch 3 where it will be flooded.

(b) The same packet sent on Figure 2.3a flooded to all ports on switch 3. All switches start having their MAC database unstable.

Figure 2.3: An exemplification of the MAC database instability.

2.1 Spanning Tree Protocol (IEEE 802.1D)

This need for a loop-free topology with redundancy is not new, in 1985 this lead to the proposal of STA developed by Radia Perlman [1], that wrote the following poem, while developing the algorithm.

Algorhyme

I think that I shall never see
a graph more lovely than a tree.

A tree whose crucial property
is loop-free connectivity.

A tree that must be sure to span
so packets can reach every LAN.

First, the root must be selected.
By ID, it is elected.

Least-cost paths from root are traced.
In the tree, these paths are placed.

A mesh is made by folks like me,
then bridges find a spanning tree.

In 1990, Institute of Electrical and Electronics Engineers (IEEE) standardized a protocol as a standard IEEE 802.1D, that includes Spanning Tree Protocol. Since the protocol was standardized, it was possible for different manufacturers to implement STP on their equipment.

On a STP topology, such as the one represented on Figure 2.4, it is possible to have redundancy and still having a free-loop Ethernet topology. The protocol defines a switch designated by root-bridge, the switch 2 on Figure 2.4. The root-bridge is in charge of all traffic that goes through the topology.

As seen, if switch 1 or switch 4 wants to communicate with switch 3, the traffic goes through the root bridge. One of the scalability issues this protocol has, is the circumstance that all traffic needs to go through this particular switch creating a bottle neck for the topology.

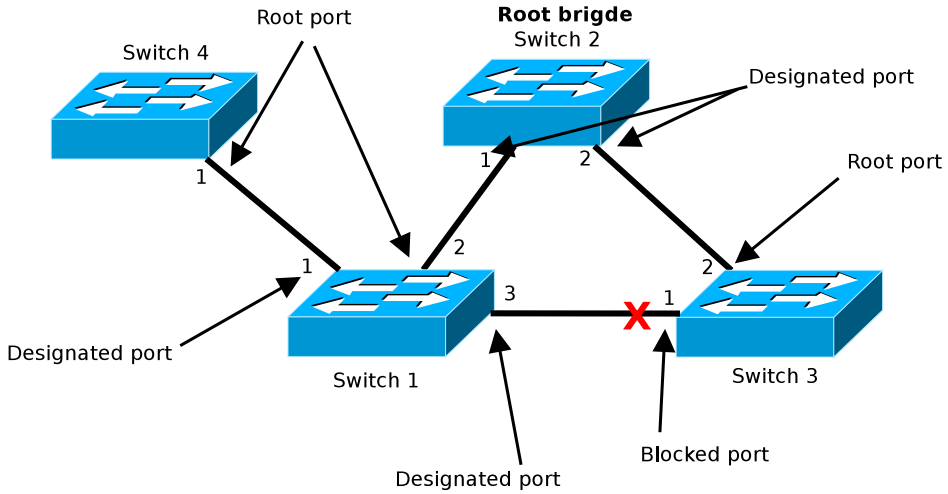


Figure 2.4: A STP topology with its root bridge and all ports status accordingly with the STP.

STP bridges start communicating in order to elect the root bridge, as the bridge with the lowest bridge ID. After the root bridge selection, Spanning Tree Algorithm performs the spanning tree calculation determining the least costs paths from the network bridges to the root bridge. The root port connects the link connected to the root bridge, or the port that has the shortest path to the root bridge. Depending on the cost of the port, the remaining ports are then defined as designated if they have the lower cost; or blocked ports to prevent the loops on the topology. Each port has to pass through different states before having a rule. Those state are defined follows:

- Blocking - A port that is blocked to prevent loops in the topology, as an example it would be port #1 of switch 3 on Figure 2.4. This port only exchange a particular type of packets, designated by Bridge Protocol Data Unit (BPDU), that contains information about ports states, addresses, priorities and costs on each bridge;
- Listening - A port state where it is listening for BPDU packets. This is a state present when the tree is being created. The port stays on this state at least 15 seconds before moving to the blocking state or learning state;
- Learning - A similar state to listening state. On this state the bridges may receive and process data packets without passing to the forwarding state. Only after 15 seconds on this state that the port goes to the forwarding state;
- Forwarding - Final state where the port performs its normal functions;
- Disabled - A state where the port is disabled and does not participate in the STA.

After the tree has been created by the protocol, there is another issue to take in consideration. If the topology changes, STP network topology has to be recalculated, taking around 30 to 50 seconds (15 seconds for listening + 15 seconds for learning + 6 to 40 seconds of a specific timer depending on the manufacturer) before moving from a blocked port into the forwarding state. This is a large amount of time on today's requirements and it started to become unacceptable to use this protocol on data centers.

Finally, there is the problem of unused resources. As seen on Figure 2.4 there are unused links, if they were used, this could bring a more dispersion of the traffic giving a better load-balancing topology.

Spanning Tree Algorithm is great algorithm to solve the broadcast storms on an Ethernet topology. Bridge networks with huge bit rates, as the data center networks have huge amount of traffic that, because of the based logical topology, need to cross through root bridge interfaces, making those switches very expensive. Exposing that, it is not perfect if we have other needs for fast communications that appeared with the evolution of Ethernet. Some improvements were made until RSTP was developed and next section briefly describes it.

2.2 Rapid Spanning Tree Protocol (IEEE 802.1w)

In 2001, IEEE published the Rapid Spanning Tree Protocol designated as 802.1w. RSTP has the characteristic of being more proactive than STP. As described before, STP has essentially four port states: listen, learning, forwarding and blocking. The RSTP substitutes are only three:

- Discarding - similar to the STP blocking state;
- Learning - improving of the STP learning state;
- Forwarding - identical to STP forwarding state.

There are four port roles in RSTP:

- Root port - the same designation used in STP, the port to reach the root bridge;
- Designated port - the port designated for a LAN segment;
- Alternate port - a port that has an alternative route for root bridge in case the root port goes down. It is instantly selected if the root port fails;
- Edge port - Ports that are connected with non-switch devices.

RSTP does not forget the ports like STP, instead the blocking port was removed and the alternate port was added. Many of the timers were eliminated, such the 20 seconds before moving from a blocking state in to a listening state and the 15 seconds for listening BPDU packets to make sure that port is not a loop.

On every topology change, the switches send topology change messages through the network and every switch knows what to do with that information, instead of rediscover the network and finding out which part of the topology had change.

On modern data centers STP is no longer used essentially due to it is large amount of time for reconfiguration after a change in the STP topology. Using RSTP it is also possible to recalculate a new tree for the new topology in around 3 seconds, which is a significant improvement from STP.

Those improvements have filled the needs of data centers and brought a fast recovery on topology changes. There was still a problem regarding the usage of links from the alternate ports and the overload of the designated root bridge.

2.3 Multiple Spanning Tree Protocol (IEEE 802.1s)

Having one of the issues from STP solved by RSTP to the non real-time scenarios, the large amount of reconfiguration time, there was other main issue that had yet to be solved: the unused links from the blocked/alternative ports.

Cisco created Per-VLAN Spanning Tree (PVST) [15] as a solution to solve that problem. It was possible to have a spanning tree instance for each Virtual Local Area Network (VLAN) allowing a load balance traffic at layer 2. Later, IEEE created the Multiple Spanning Tree Protocol (MSTP), designated by IEEE 802.1s, with some similar functionalities of PVST. Both PVST and MSTP had the purpose of giving STP the scalability needed for large enterprise networks, without having a bottle neck in only one bridge as visible in STP. This protocol is an extension of RSTP and the convergence times did not change. If only one VLAN is used on all topology, the problem of unused links are not solved and the unused links keep existing as in STP.

Besides MSTP there was another improvement made by IEEE known as Link Aggregation Group (LAG) (IEEE 802.3ad). This standard allows two or more links to be connected between two switches causing those links to become a single logical link. Besides LAG, it was also standardized IEEE 802.1ax, an extension to LAG, designated as Multichassis Link Aggregation (MC-LAG) [16]. Having two links connected to two different machines, this extension provides interconnection and redundancy between these machines.

2.4 Transparent Interconnection of Lots of Links

New technologies are emerging from IEEE and IETF, they both promise to solve all problems Spanning Tree Algorithm-based protocols did not solve. On this section it will be discussed IETF TRILL and on later it will be discussed IEEE 802.1aq.

Transparent Interconnection of Lots of Links (TRILL) was initially proposed by Radia Perlman to IEEE 802.1 as a substitute for all Spanning Tree Protocols. It was rejected due to the fact that it was not very useful. The STP is reckoned to be still good and the idea of having hop counts and the routing mechanism are both unpleasant [17]. Then, Radia Perlman organized a Birds of a Feather session and Internet Engineering Task Force (IETF) accepted TRILL.

Perlman also created a new poem describing this new protocol and it is available in the RFC that standardizes TRILL [18] [19] [20] [21] [22].

Algorhyme v2

I hope that we shall one day see
A graph more lovely than a tree.

A graph to boost efficiency
While still configuration-free.

A network where RBridges can
Route packets to their target LAN.

The paths they find, to our elation,
Are least cost paths to destination!

With packet hop counts we now see,
The network need not be loop-free!

RBridges work transparently,
Without a common spanning tree.

Intermediate System to Intermediate System (IS-IS) [23] is a routing protocol designed for an administrative network. IS-IS have the same shortest path between all Routing Bridges (RBridges). It is not necessary to replace all topology since TRILL keeps interoperability with STP. In fact, as more bridges are replaced by RBridges, the better the bandwidth usage and more stable the topology become [24].

Contrary to SPB, TRILL contains two new headers: an outer header and the TRILL header. The outer header is only used to send packets between RBridges, and it essentially specifies the source and

destination of a packet. The TRILL header contains multiple fields such as: ingress RBridge (16 bits), egress RBridge (16 bits), hop count (6 bits), and a multidestination flag bit (1 bit).

On Figure 2.5 is illustrated a topology with six RBridges. The basic operation for a packet sent from machine A to machine B consists in the following steps:

- Machine A sends a packet with destination machine B;
- RBridge 1 receives it and encapsulates it with a TRILL header, where ingress is RBridge 1 and egress is RBridge 4;
- RBridge 1 then puts an outer header, different than TRILL header, to send that packet to RBridge 6;
- RBridge 6 receives it, removes the outer header, decreases the hop count present on TRILL header, applies a new outer header and sends it to RBridge 5;
- RBridge 5 performs the same procedure of RBridge 6 and forwards it to RBridge 4;
- When the packet finally reaches RBridge 4 it verifies that the egress bridge is itself and sends the packet to the appropriate port, reaching Machine B.

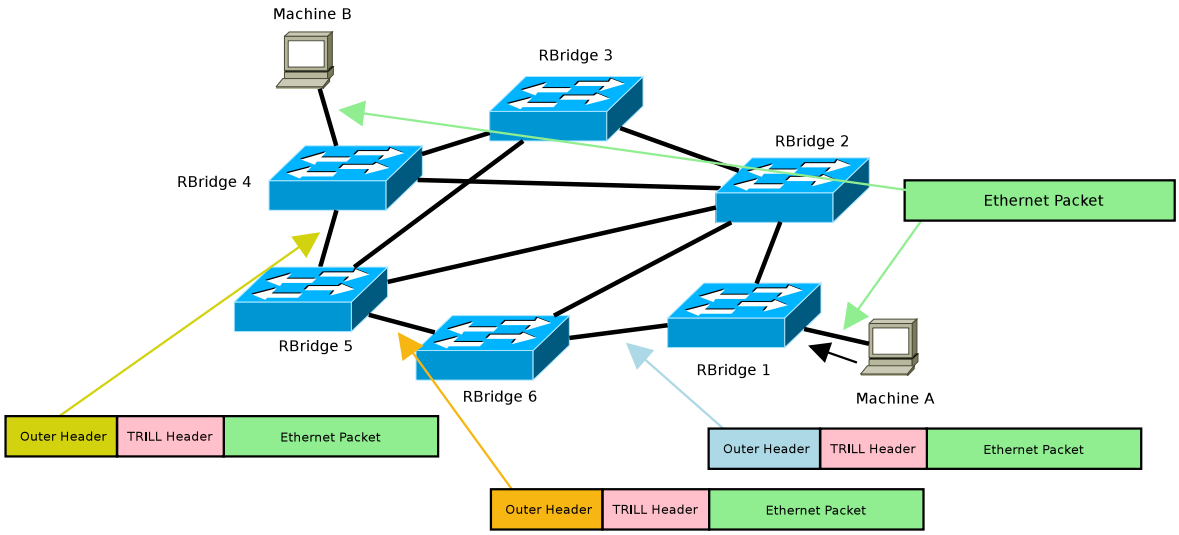


Figure 2.5: TRILL topology with the illustration of the packets' content on different sections of the topology.

The example described assumes that RBridge 1 knows the RBridge destination to send the packet in order to reach Machine B. If RBridge 1 does not know where to send the packet it simple sets the multidestination flag and sends the packet through a pre-defined tree, reaching all RBridges and consequently the destination RBridge. Since TRILL uses IS-IS as a link state protocol, all RBridges have information about the remaining RBridges in the topology. Once each one of the RBridges have the same information, all of them calculate the same tree for the distribution of multidestination packets.

2.5 Shortest Path Bridging (IEEE 802.1aq)

Shortest Path Bridging (SPB), specified in IEEE 802.1aq and approved in 2012, was created has a replacement for STP, RSTP and MSTP. This protocol would not be only used on enterprise networks but also in carrier networks. Similar to STP, SPB will be specifically used on data centers as well the campus of enterprise networks. In STP, RSTP and MSTP there is one single topology tree (per-VLAN on the last two) for all the traffic on a redundant topology, on SPB is used Shortest Path Trees (SPTs). With SPT is guarantee that all traffic between two bridges is sent through the shortest path [25].

IS-IS is also used in SPB as link state protocol to exchange information in order to calculate the SPTs between bridges. SPB also calculates multiple Equal Cost Trees (ECTs) to provide support for load balancing. Each ECT uses a different SPT algorithm. Having multiple SPTs that share the same ECT it is possible to distribute multiple VLANs by SPTs. This is similar to MSTP and there is still no differentiation of different flows for load-balancing. However IEEE is working on a new standard designated by Equal-cost multi-path routing (ECMP) (IEEE 802.1Qbp). This standard has some improvements, regarding the network size, in the scaling properties, by allowing to use many more equal cost paths than 802.1aq's current ECT mechanism [26]. The standard may also include a Time To Live (TTL) field to provide loop mitigation.

2.6 CHAPTER OVERVIEW

The evolution of different protocols to suit the requirements of data center and enterprise networks by solving some problems present on a layer 2 network is remarkable. This evolution was made essentially because the lack of a field in the Ethernet header, the hop count, something the emerging standards, such as TRILL and SPB, will have. On Table 2.1 are describe the major differences between STP, RSTP, MSTP, TRILL and SPB.

Characteristics	**STP**	**RSTP**	**MSTP**	**SPB**	**TRILL**
Organization	IEEE				IETF
Encapsulation	None			SPB-M - MAC-in-MAC (802.1ah) SPB-V - Q-in-Q (802.1ad)	TRILL header
Loop Prevention	Block Redudant Ports			Reverse Path Forwarding Check (RPFC) for unicast and multicast	TTL for unicast & RPFC for multicast
Load Balancing	No	Yes		Yes (Per VLAN) & work in progress 802.1Qbq	Yes
Time for topology convergence (seconds)	30	5		0*	0*

Table 2.1: Comparison of major differences between existing redundant standards.

Both TRILL and SPB are similiars in some aspects and they are both promising for data centers and large enterprise networks. Since they both use IS-IS, their convergence time is theoretical zero

seconds. TRILL's advantage is a new approach for layer 2 networks and provides better load balancing. The needs of a new header, causing some frame overhead and a hop-by-hop check-sum calculation does not give TRILL advantages on its use. SPB has been deployed over the years in the carrier market and used on 2014 Winter Olympics, providing an inherent advantage over TRILL in terms of proof of concept [27] [28]. Although SPB did not initially had a hop count, IEEE is working on a TTL mechanism [26].

Accordingly with [16], TRILL and SPB can be used in three specific cases:

- Whenever the MC-LAG capacity is exceeded, for instance by thousands of ports, or an additional switch is required that is not included in the MC-LAG configuration;
- Whenever a network is implemented using different manufacturers' switches for example having different switches on the access and core layers;
- The last one is related to having a single vendor producing different switches which are incompatible and they cannot participate in each others' fabric.

Two of the three use cases for TRILL and SPB are interoperability related, this is actually being solved by SDNs, as described in the next chapter.

CHAPTER 3

Software-defined Networking

Data centers have evolved to network virtualization increasing the need for an efficient use of bandwidth, workload mobility and failure control. So far, all the discussed protocols have the Spanning Tree Algorithm and IS-IS in their nature. It has been previously mentioned on this document, on Section 2.6, that these protocols already have already solved issues, such as broadcast storms and unused equipment. However, especially for critical communications on SDNs, there are still relevant issues to be solved. One of the main focuses of this dissertation is to study critical communications on layer 2 with technologies provided by SDNs.

First of all, network components are often divided in two planes the control plane, where the decisions on how and where to perform route traffic take place; and the data plane, which is associated with the transferred data based on the decisions learned by the control plane. These two planes often require different abstractions. For instance, related to the data plane are multiple layers often based on the OSI model where each layer is designed for a single task. On what refers to the control plane there are different goals to fulfill, such as: routing, isolation, traffic engineering, among others. Nowadays, those goals are achieved by different mechanisms, protocols and network architectures. For example, SDN, as the name also suggests, allows for the fulfillment of these goals through software, using multiple tools such as: OpenStack [29], OpenNebula [30], Apache CloudStack [31], Eucalyptus [32] and Open Compute Project [33]. Some of these tools use OpenFlow to control the network topology.

Taking in consideration what was previously mentioned, and according to [34], Software-defined Networking, a paradigm associated to network management is recalled where the control plane located is in one centralized remote controller and the data plane is positioned in a different network forwarding devices; as opposed to existing protocols where their decisions are destination-based, the forwarding

devices conformed to the SDN paradigm are flow-based allowing for a more flexible network; the remote controller must have several abstractions of the topology and be programmable enough in order to enable the creation of more functionalities for the network by developing applications to run on that controller. In short, the data plane is the physical network infrastructure, that consists of a forwarding devices and the control plane consists on the physical network infrastructure abstraction provided by the controller. In addition, the control plane has multiple applications that offer more functionalities by using those abstractions.

Contrary to previously mentioned protocols, SDN introduced innovation that allows for investigators to study new protocols and create a more programmable network. According to [35] this type of networks has followed three main stages in history: from the mid-1990s to the early 2000s, the active networks were introduced. They allowed the inclusion of programmable functions that enable fairer networking innovation. Another stage is related to the control and data plane separation through the development of open interfaces, which is situated between the period of 2001 and 2007. The last stage is related to the OpenFlow API and network operating systems, dated from 2007 to, approximately the year of 2010. It was in this stage that the development of scalable means for control-data plane separation began, as well as practical means. This was made possible with the first instance of widespread adoption of an open interface.

This dissertation will essentially focus on the last and most recent programmable networks stage, introducing the OpenFlow API and related work.

3.1 OPENFLOW

OpenFlow was initially created in 2008 for researchers to run experimental protocols in the networks [12]. Since then, it has been used in several research projects. It is an open-source implementation and it is currently deployed on various vendors' products. On Figure 3.1 is shown a typical OpenFlow architecture. The main purpose of OpenFlow consists on being a southbound protocol to communicate with network devices capable of interpreting OpenFlow and a controller that has a centralized overview of all events and configurations that occur on the topology.

On the one hand, the control plane of these devices is in command of the OpenFlow controller that performs operations according to the OpenFlow protocol. On the other hand, the data plane is implemented by a table called *flow table* where the packets are matched against the rules previously installed by the controller on that *flow table*. The first version of OpenFlow [36] was designed in a way that a OpenFlow forwarding device had a single *flow table*. This single *flow table* was represented by the following six fields, which still exists in most recent versions of OpenFlow, among others:

- **Header fields** - Are used to match the arriving packets based on their headers' values such as: Ingress port; Ethernet address source; Ethernet destination source; Ethernet type; VLAN

ID; VLAN priority; IP source; IP destination; IP protocol; IP ToS; TCP/UDP source port and TCP/UDP destination port;

- **Counters** - Are used for statistics purposes. There are twenty-two counters in existence, including: Packet Matches per Table; Received Packets and Duration (nanoseconds) per Flow; Received Packets, Receive Errors, Collisions per Port; and Transmit Packets per Queue.
- **Actions** - Where the actions are applied for the matching packets that arrive on the switch. There are five required actions and two optional actions for manufacturers to implement. Some of the required actions are: **ALL**, where the matched packet is sent to all interfaces except the one where it was received; **CONTROLLER**, where the matched packet is sent to the controller using the OpenFlow protocol; **IN_PORT**, where the matched packet is sent to the same port where this current packet was received.
- **Priority** - To define different priorities for different *flow entries*.
- **Timeouts** - Where there are hard and idle timeout fields. The hard timeout defines how many seconds a *flow entry* can be active before expiring, while the idle timeout defines how many seconds of absence of traffic it is necessary before the rule expire.
- **Cookie** - It is used to distinguish different *flow entries*, mainly for controller use.

These six fields representing the single flow table allowed for the definition of specifications that helped demonstrate the OpenFlow potential as seen in [37].

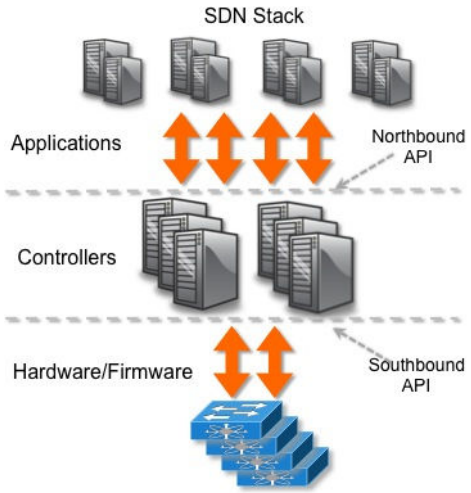


Figure 3.1: Overall architecture of OpenFlow on an SDN [1].

Since 2008 the protocol has evolved and it is currently on version 1.4 [38]. With the presentation of new versions more actions were added, some of which were important for the implementations developed in light of this dissertation. One of the improvements made for the switches was the opportunity to have a pipeline of tables instead of having one single table to control all implemented rules. This

[1]Image available: http://networkstatic.net/the-northbound-api-2/

added the possibility to include legacy software by using a table for each functionality, offering more abstraction and modularity for the switches. Legacy network topologies would have one machine for each functionality, for example an Intrusion Detection System (IDS), a firewall, a load balancer, among others. With OpenFlow it is possible to implement those functionalities on a single network device. Figure 3.2 illustrates this abstraction and it is possible to observe that it still presents the possibility to have an ACL policy flow table offering similar functionalities to a firewall.

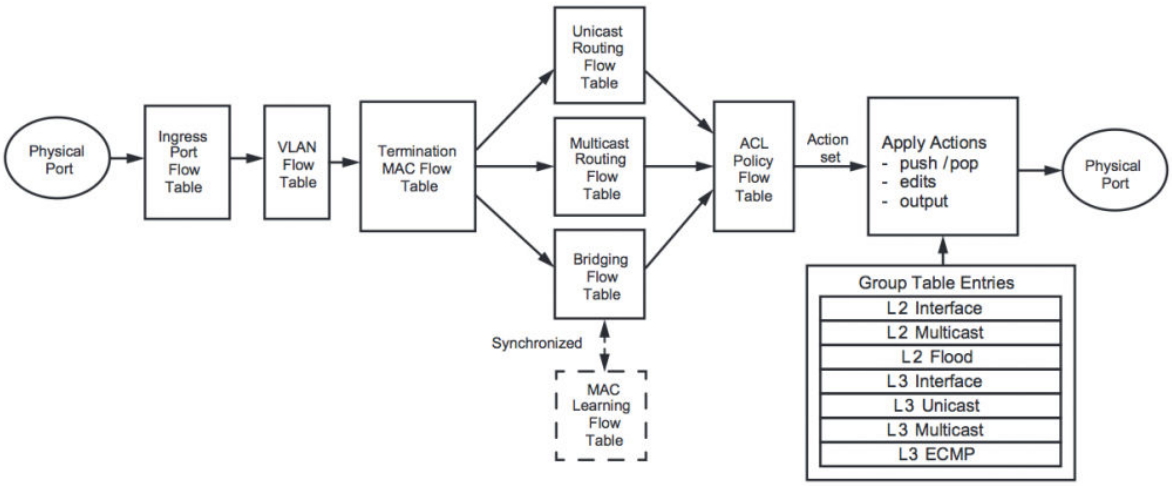


Figure 3.2: Pipeline abstraction for OpenFlow switch [2].

One of the crucial factors for critical applications on a OpenFlow topology is the controller itself as it may represent a bottle neck and a single point-of-failure. However, this is not necessarily true since the controller does not have to be a single physical machine and it can be seen as a single virtual instance with multiple machines for backup or for parallel processing. For this reason, the specification has a controller role change mechanism that is enforced by each controller implementation. The only function for each switch on the topology is to remember the role of each controller, considering the latter's possibility of being regarded as multiple machines. Taking into consideration the controllers' performance it is possible to query parallel controllers. Although some controllers' implementation are unable to handle high speed networks with 10 Gbps links [39], it has been proven that improving the performance of an existing controller can tolerate the handling of 1.6 million requests per second [40].

Another implemented feature on OpenFlow was the creation of groups. They represent a set of ports that act as a single entity made from group buckets. A group bucket contains a set of ordered actions that can be applied on a packet before it is sent out to a port or even another group. There are four types of groups in which two of them are required and the other two are optional:

- **all** - All buckets are executed. The received packet is cloned and executed on every action. (Required)

[2]Image available: http://bigswitch.com/blog/2014/05/02/modern-openflow-and-sdn-part-i

- **select** - Only one of the buckets from the group is executed. The choice of the bucket that is to be used is based on a selection algorithm, external to OpenFlow, implemented on the switch. (Optional)
- **indirect** - This executes the only defined bucket present on this group. It is similar to type **all** if operated with a single bucket. This allows for multiple *flow entries* or groups to have the same action by offering, for example, a faster and more efficient convergence in next hops for IP forwarding [38]. (Required)
- **fast failover** - Executes the first live bucket. Every bucket is associated with a group or a port for the respective actions. On this group it is only possible to apply an action of the bucket while the respective port or group is respectively on-line or active. (Optional)

From all existing groups there is one which is fairly interesting for critical applications, the fast failover. With this group it might be possible to have a backup port for a principal port. Once the first port is put offline cannot be activated and the next port on that group will be used instead, without losing any packet. In the past years there have been developed several projects related to fast failure recovery mechanism using OpenFlow [41] [42]. Although they achieved good results, packet loss was still evidenced, essentially due to the lack of use of fast failover groups on their mechanisms.

Among a huge amount of new features, there is one last significant feature for this dissertation. As described previously, OpenFlow was initially created for researchers to test new protocols on a large scale. Thus, there is a type of message that can be transferred between the controller and the switch: the *Experimenter Message*. It is essentially defined by the OpenFlow header, an experimenter ID and an array, and offers the possibility for researchers to implement new experimental features that are not covered on OpenFlow. The reason that makes this type of messages relevant for this dissertation will be further explained on Section 4.

Finally, on the OpenFlow protocol is defined an essential feature for critical applications. The protocol defines that the OpenFlow switches can operate if the connection with the controller is lost. On a worst case scenario, even if the connection between the switch and the controller is lost, the switch, having the proactive rules installed by the controller, does not lose the flows already present. Only new flows that are not defined on the switch are discarded.

3.2 CONTROLLERS

An SDN controller can be regarded as an Operating System (OS) for a network. According to [43], the primary functions of an OS are to provide: "application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources". On a network topology, that makes use of the SDN paradigm, the hardware abstraction is given via OpenFlow and the abstract set of resources is given by the controller to the

programmers. Nowadays, this hardware abstraction does not exist and network management is done via closed Network Operating System (NOS) such as [44] [45]. However, with recent developments there were developed open-source NOSs for OpenFlow. The first NOSs for OpenFlow were NOX [11] and Maestro [46] [47].

The overall architecture of existing NOSs consists of five different hierarchical layers:

- Network applications layer - Layer containing applications developed by programmers to give more functionalities to the SDN;
- Northbound layer - Offers an API for the applications so they can communicate with the NOS core;
- Core layer - This layer contains modules that wrap up the overall information about the topology. This layer has the necessary hardware abstractions for the developed applications;
- Southbound layer - Offers an API to the lower layer so the core can communicate and control the hardware from the physical topology;
- Southbound applications layer - Provides applications to control the physical hardware via management protocols. These applications do not need to be OpenFlow, there are other protocols that can be used to configure the physical forwarding devices on this layer such as: OVSDB [48], NETCONF [49], SNMP [50], among others. Their existence is due to legacy hardware and they cannot provide the same functionalities offered by OpenFlow.

Additionally, on the southbound layer there is the possibility of having modules to communicate with other different NOSs. This brings more modularity, scalability and interoperability for a SDN that uses different implementations of a controller. Table 3.1 indicates the major differences and functionalities of some existing NOSs.

Controller	Management	Processing type	Northbound	Southbound	OpenFlow Version	Language	License	Flows/s
DISCO [51] [52]	GUI/CLI	Distributed	REST	OpenFlow, AMQP	V1.0	Java	-	-
Floodlight [53]	GUI/CLI	Centralized	REST	OpenFlow	V1.0	Java, Python	Apache	600K
Maestro [46]	CLI	Centralized w/ parallel computation	Internal (DAG)	OpenFlow	V1.0	Java	LGPL v2.1	2M
NOX [11] [54]	GUI/CLI	Centralized w/ parallel computation	Internal	OpenFlow	V1.0	C++, Python	GPL	100K
Onix [55]	CLI	Distributed	Internal	OpenFlow, OVSDB, NIB	V1.0	C++, Java, Python	-	-
ONOS [56]	GUI/CLI	Distributed	Internal	OpenFlow	V1.0	Python	GPL V2	-
OpenDaylight [57]	GUI/CLI	Distributed	OpenStack Neutron, REST, Internal	OpenFlow, OVSDB, NETCONF, BGP, PCEP, SNMP	V1.3	Java, C, C++	EPL-1.0	100K
POX [58]	GUI/CLI	Centralized	Internal, REST	OpenFlow	V1.0	Python	GPL	30K
Ryu [59]	GUI/CLI, Web UI	Centralized	Internal	OpenFlow	V1.4	Python	Apache 2.0	-
The Beacon OpenFlow [60]	Web UI	Centralized w/ parallel computation	Internal, REST	OpenFlow	V1.0	Java	GPL V2	6M

Table 3.1: Characteristics and functionalities offered by SDN controllers. (Adapted from [34])

The majority of the considered NOSs only offer OpenFlow as their southbound API in order to control the network. The only exceptions are: DISCO, Onix and OpenDaylight. DISCO controller was built on top of Floodlight to offer a distributed approach for scalability on enterprise environments. The Advanced Message Queuing Protocol (AMQP) is used for DISCO controllers to exchange information between themselves. Onix, also being distributed, transfers its network state by exchanging a structure called Network Information Base (NIB). As the authors state, it is analogous to the Routing Information Base (RIB) used by IP routers [55]. Last but not least, OpenDaylight is, by far, the controller that provides more southbound APIs offering more integration for more heterogeneous and even legacy topologies on a single controller. For example, there has been work done considering the creation of a new southbound API for Cable Modem Termination System (CMTS) networks [61].

Regarding the OpenFlow version, there are only two projects within the considered controllers that support recent versions of OpenFlow. This can reveal how bonded a project is in following the evolution of OpenFlow.

One last thing to take into consideration is the number of flows a controller can process per second. According to [62], a network with 100 edge switches can produce up to 10 million flows per second. Although the scalability is an important issue to take into account, when dealing with a centralized machine to control an entire network, it is more relevant to offer functionalities that can be performed on that network. As pointed out previously, some researchers have discovered that it is possible to tune a controller to enhance performance [40]. Those researchers tuned the NOX controller and reached up from 100K flows per second to 1.6 million. Other researchers created a controller called Mapple [63], that can reach up to 20 million requests per second. They also tested The Beacon controller and reached up to 15 million flows per second. Both of those values were reached using 40 cores of computation power.

3.2.1 OPENDAYLIGHT

Considering the developed implementations, the chosen controller was the OpenDaylight as it is the controller with more contributions and developments using OpenFlow. Its overall architecture is similar to a typical NOS. As illustrated on Figure 3.3, the network application layer includes applications to connect with other projects such as OpenStack [29]. In this case, OpenDaylight is the management tool for the network topology and OpenStack is the Infrastructure as a Service (IaaS) solution.

The northbound layer from OpenDaylight uses REST in order to communicate with the core. The controller platform offers some basic network functions analogous to the abstractions offered by a usual OS. This is managed by the Service Abstraction Layer (SAL) API, on the Hydrogen release called API-Driven SAL (AD-SAL). On the southbound domain there are plugins with support for OpenFlow v1.3, which means it has OpenFlow groups fast-failover, and other protocols that give the possibility to manage legacy networks although this last subject is not the primary focus of this dissertation.

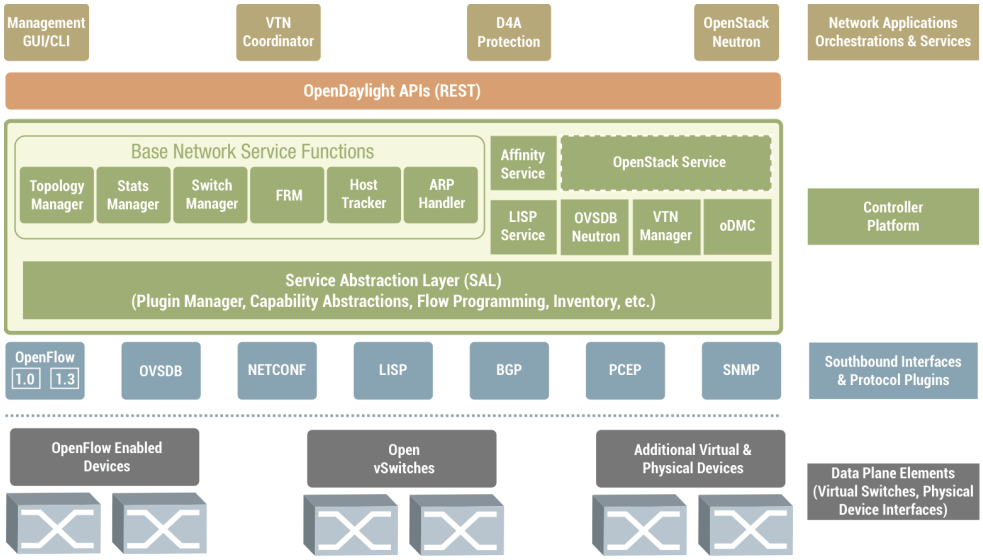


Figure 3.3: Overall architecture of OpenDaylight Hydrogen [3].

The OpenDaylight Hydrogen was released on February 2014, and the next release, called Helium, is expected to be ready on September 2014. One of the major differences between the first and second versions is the change of the SAL architecture. As an example illustrated on Figure 3.4, on AD-SAL API if a northbound application wants to consume information from the southbound plugin, typically they both have to include services or functions for each one at compile/build time. AD-SAL is also stateless, its services provide asynchronous and synchronous of the same function/service and lastly, it is restricted to flow-capable devices. The newest core architecture, called Model-Driven SAL (MD-SAL), provides the same services by defining models. Instead of having the data adaptions statically defined at compile time, those data adaptations between providers and consumers are defined by models. This allows for multiple northbound or southbound plugins to access the information created by producers. Often the producers are the southbound plugins and the consumers are core applications and northbound applications. MD-SAL is not stateless and can store data for models defined by the plugins. That storage, called MD-SAL storage, provides the possibility for consumer applications to read data from provider plugins.

Since the plugins on this dissertation were developed between releases, it brought some challenges related to the fact that some of the OpenDaylight plugins, which the developed plugins depended on, were developed for the AD-SAL architecture and others were developed for MD-SAL.

[3]Image available: http://www.opendaylight.org/project/technical-overview

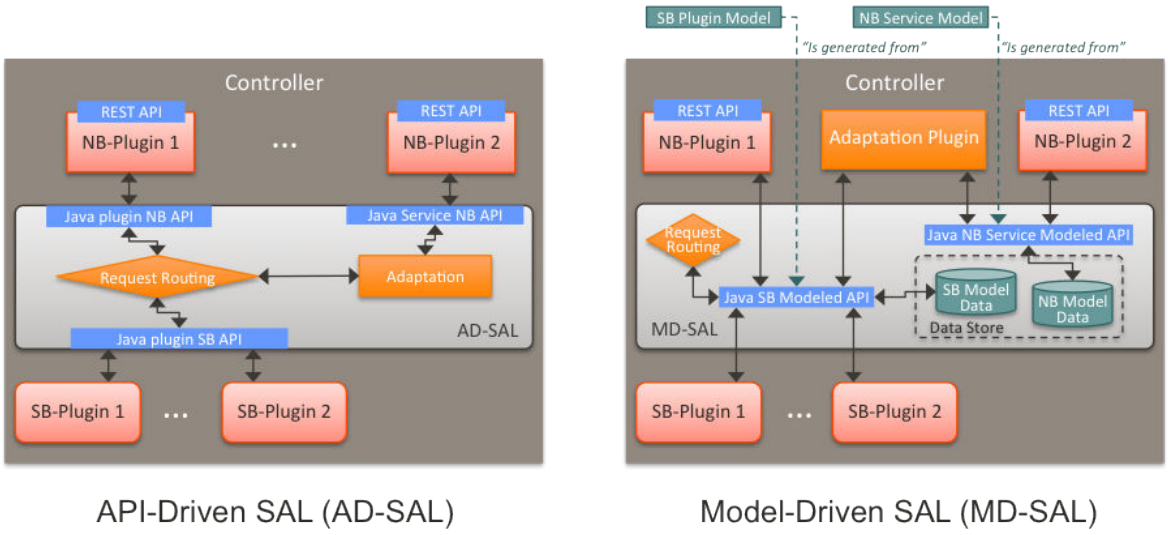


Figure 3.4: Differences between AD-SAL and MD-SAL of OpenDaylight [4].

3.3 OPENFLOW FORWARDING DEVICES

There are multiple OpenFlow forwarding devices deployed on the market. The term "forwarding device" will be used as a replacement for both the terms "switch" and "router", since OpenFlow can also be used in routers. The primary focus of this dissertation did not include a deep study of this subject, so only a brief explanation will be taken in consideration. The developed forwarding devices have given a clear sign that their flow table size has grown at a considerable pace that aims to meet the needs of future SDN deployments [34]. Among the various implemented solutions of OpenFlow forwarding devices, there are hardware and software implementations. These two implementations can be performed on different types of devices, such as switches, routers, chassis and cards. With the amount of virtual access ports on data centers already exceeding the number of physical access ports, software-based devices have shown to be very promising solutions for network infrastructures ([64] [65], also referenced in [34]) and are also used on IaaS. Some of the hardware products already deployed are: NetFPGA from NetFPGA [66], NoviSwitch 1248 from NoviFlow [67], 8200zl from HP [68] and MLX series from Brocade [69]. Some of the existing software implementations are: ofsoftswitch13 from CPqD [70], Open vSwitch from Open vSwitch [71] and contrail-vrouter from Juniper Networks [72]. Most of the solutions presented have OpenFlow v1.3 implemented. Although only a small number of forwarding devices are enumerated, it is possible to see the acceptance of different manufacturers to OpenFlow. The software-based devices are revealing to be a promising solution for virtualized networks since they offer more modularity than hardware based devices and allows legacy networks to be managed by OpenFlow [73].

[4]Image available: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:FAQ

3.3.1 OPEN VSWITCH

The first forwarding device used for this dissertation was the *ofsoftswitch13*, since it was the only switch with the OpenFlow v1.3 implemented and, therefore, the fast-failover group was also included. Unfortunately, due to an implementation issue, the *ofsoftswitch13* did not recognize when a port was back online, when a link was removed and then re-added. This issue was reported to its developers and acknowledged by them as an issue identified by other developers in the past. It was relevant to perform this port removal and re addition recognition, especially during the evaluation stage, so another switch was chosen as replacement, the Open vSwitch. While recognizing the relevance for a port status, the recommendation of the OpenDaylight community to use Open vSwitch with OpenDaylight were also taken into consideration.

Open vSwitch is an open source project to virtualize environments with multiple servers. As observed in Figure 3.5, these multiple servers are represented as Virtual Machines (VMs). This allows for a single physical machine to offer the services provided on those servers via Open vSwitch. This virtual switch has the capability of forwarding the traffic between different VMs and from a physical network to a VM. Its main features are: STP; QoS; IPv6 support; OpenFlow and extensions for virtualization; multi-table forwarding pipeline with flow-caching engine; multiple tunneling protocols (GRE, VXLAN, IPSec, GRE and VXLAN over IPSec) among others that can be found in [74].

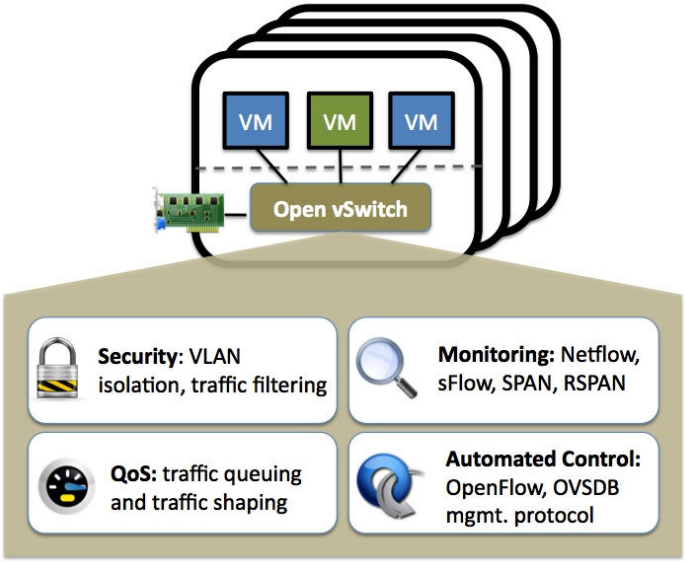


Figure 3.5: Overall architecture of Open vSwitch [5].

The first Open vSwitch implementation and specification appeared in 2009 [75] and it is currently on version 2.1.2. The version chosen for this dissertation was the version 2.1.0. Regarding this version, it should be noted that it already include a stable implementation of the OpenFlow v1.0 and that, in addition, the groups feature was already implemented and available for testing and development purposes [76].

[5]Image available: http://openvswitch.org/

CHAPTER 4

Implementation

The protocols studied in the previous section 2 do not suit for critical applications. SDNs are emerging with a huge amount of new functionalities and is essential its reliability does not fail under critical communications. This work supplies three different approaches both of them with the purpose to have critical communications working in case of a link failure on a SDN-topology based.

This section aims to explain technical details on the implementations developed in order to solve the main problem of this work. The first subsection is dedicated to clarifying the changes made on some OpenDaylight bundles[1]. The remaining subsections are intended to explain the three different approaches established.

The first approach, entitled Implementation 1, focuses on topologies that have two completely full disjoint paths between every two hosts on a topology.

The second approach, developed under the name Implementation 2, included all topologies that either have or do not have two completely disjoint paths between every two hosts.

The third and final approach, designated Implementation 2.1, refers to an improvement on Implementation 2 by omitting one of the matching fields sent to the topology nodes[2].

Both the principal implementations methods and their overall function will be explained on a specific subsection.

4.1 OpenDaylight

As previously described, in order for a fast recovery algorithm to work, there is the need for a controller to account for some requirements. With OpenDaylight it is possible to have all of those

[1]On this section, the word *bundles* has the same meaning of *plugins*.
[2]On this section, the word *nodes* has the same meaning of *switches* or *OpenFlow forwarding devices*.

requirements fulfilled with some of the bundles available. The bundles that possess the features that are necessary to cover those requirements are the following:

- ✓ Support for OpenFlow 1.1+.
  - `org.opendaylight.openflowplugin`
- ✓ Bundle that has an overall notion of the network topology.
  - `org.opendaylight.controller.md.topology-lldp-discovery`
  - `org.opendaylight.controller.md.topology-manager`
- ✓ Bundle to track hosts.
  - `org.opendaylight.controller.hosttracker`
  - `org.opendaylight.controller.arphandler`
- ✓ Bundle with a graph searching algorithm.
  - `org.opendaylight.controller.routing.dijkstra_implementation`

Although the bundles worked as expected, it was necessary to apply a few modifications in their source code as well as to create non-existing features and methods needed for the bundles developed in order to cope with the ambitions of this dissertation. Those modifications are described on the following subsections.

The sequence diagram that represents the communications between the bundles when an event occurs in the topology can be found on Figure 1 which is located in Appendix A-Sequence diagrams related to the implementations on OpenDaylight.

4.1.1 OPENFLOW PLUGIN

The overall objective of the OpenFlow Plugin is to bring support for OpenFlow 1.0 and 1.3.x [77]. Without changing its main purpose, the introduced changes were made on the code extracted from the openflowplugin project available online [78].

The first modification conducted referred to the addition of the missing `buffer_id` field from the OpenFlow messages translated by this plugin from the OpenFlow library bundle. The `buffer_id` refers to a buffer ID destined for a packet buffered at the switch and is sent to the controller in form of a *packet-in* type message [79].

This `buffer_id` field is essential for both implementations due to the fact they will make use of it when the controller receives a *packet-in* message (an unmatched packet from the switch). Also, when the *packet-in* reaches the controller, there is expected a delay in the calculation of the primary and backup path. In between that period, the switch will not stop working while waiting for an answer from the controller and will save the unmatched packet in a buffer. To prevent any packet loss for

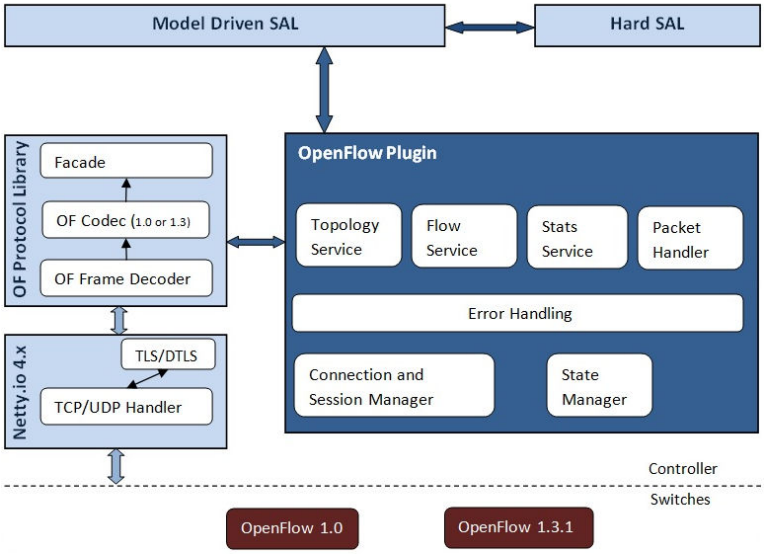


Figure 4.1: Overall architecture of OpenFlow Plugin [3].

those unmatched packets, when the controller sends the rules for the switches, it registers the same `buffer_id` value the switch sent in the *packet-in* message on the rule the controller sends to that same switch.

The second modification processed was also an addition in the OpenFlow Plugin project. The OpenFlow specification defines the structure of an experimenter message. Experimenter messages, furthermore, provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space [79]. Although it was not completely put aside by its developers, the experimenter message was omitted in the first release of this plugin because they intend to create an extensibility support for different manufacturers in the OpenDaylights' next release [80].

In light of this project, the main purpose of the experimenter message was to use the learning feature provided by Open vSwitch. Learn feature is an experimenter action for Open vSwitch often referred to as `learn`. With this action is possible for nodes to reconfigure themselves, without contacting the controller, due to the controller's capability to predict what to do, on each node, in case an interruption occurs in the primary path. A usage for this type of message will be described in section 4.2 - Implementation 1.

4.1.2 TOPOLOGY MANAGER AND LLDP DISCOVERY

The LLDP discovery bundle and the OpenFlow plugin are essential to the topology manager as they are used to find links over a topology. As acknowledged in Figure 4.1 the OpenFlow plugin contains a topology service. That service gets notified by the OpenFlow plugin whenever a port change

[3]Image available: `https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:Overview_Architecture`

occurs in the topology. If the port changes to up, the service adds it to the lists of ports and sends an LLDP message to that port. The `topology-lldp-discovery` bundle has an LLDP listener, which listens for LLDP packets and, if they match a format sent by the OpenFlow plugin topology's service, it emits a `LinkDiscovered` notification. Until the port is set down, LLDP messages are sent every 5 seconds. The `topology-manager` is listening for `LinkDiscovered` events and whenever it receives a notification this bundle updates the topology accordingly with the given description. This happens when either a link is added or removed.

The `topology-manager` is also responsible of sending notifications when the topology changes. One of the listeners is the `Dijkstra`'s bundle that needs to know whenever a change in the topology occurs so it can update its internal graph.

4.1.3 HOST TRACKER AND ARP HANDLER

The `hosttracker`, as the name indicates, tracks the hosts over the topology. Host tracker is characterized as a passive process as it only saves the node connector where a host is attached by analyzing the different packet-in messages received by the controller. This enables other bundles, including the topology manager, where a host is connected. It is similar to a MAC table on a layer 2 switch but instead of serving only one switch, it is destined for the whole OpenDaylight topology.

The ARP handler is a more active bundle. Similar to an ARP table on a layer 2 switch, the ARP handler saves the respective MAC address that corresponds to an IP Address.

When a host sends an ARP request, that packet will be captured by the node directly connected to it. Because there is not any rule to deal with ARP requests, the nodes send these packets to the controller. When the ARP handler bundle receives this packet one of two things happen: either the ARP handler knows the questioned MAC address or it does not.

If it has it, the ARP handler sends an ARP reply on behalf of the ARP request's destination to the ARP requester.

Though, if it does not have the address, the ARP handler produces a MAC flood. It will replicate the ARP request for every node connector that is not directly connected to another OpenFlow node in the topology. As result, it will receive an answer from its respective host without the ARP passing by the OpenDaylight topology core. Similar to ARP request, the node does not have a rule for ARP replies and will send it to the controller. When this unmatched packet is received in the controller, the ARP handler will send the ARP reply to the node connector where the ARP request came from. At the same time `hosttracker` will save the location for the host that sent the ARP reply.

Those two bundles work in a way that offer the possibility to see the OpenDaylight topology as a big layer 2 non-OpenFlow switch. Unfortunately, both of them as well the `Dijkstra`'s bundle, are developed for the AD-SAL architecture. This situation brought some challenges on every implementation developed for this work. For instance, due to the fact that every other bundle was already developed for

MD-SAL, it revealed to be more difficult to create a bundle that could interact with both architectures at the same time. There is a proposal to create a better `hosttracker` [81], as well as to develop it in MD-SAL, with its conclusion dated for the next release of OpenDaylight.

4.1.4 DIJKSTRA'S BUNDLE

The `Dijkstra`'s bundle uses the `Dijkstra`'s shortest path algorithm and equally offers a good interface to calculate a path between two nodes in the topology. The topology is the same located in the `topology-manager` bundle. When an event related with the topology occurs, such as a port down, a link removal, a node addition, among others, the `topology-manager` triggers that change to the other bundles through AD-SAL. When that notification is received, the internal graph of `Dijkstra`'s bundle is updated. Then, triggers all bundles listening for `Dijkstra`'s topology changes. Since the implementations developed are consumers, therefore listeners, they will listening for those changes when the internal graph of `Dijkstra`'s bundle is updated.

Although this bundle provides an interface to calculate the shortest path between two nodes, some methods were added to provide essential functionalities for the developed implementations. One of the existing methods was: `getRoute(Node src, Node dst)`.

This method provides a path between 2 nodes (`src` and `dst`) for the current topology. However it was not enough due to the need of some features that could be provided by this bundle. Therefore 2 more methods were added:

```
getRouteWithoutAllEdges(Node src, Node dst, List<Edge> edges);
getRouteWithoutSingleEdges(NodeConnector srcNodeConnector, Node dst, List<Edge> edges);
```

The `getRouteWithoutAllEdges` method is used to return a path without all the given edges between `src` and `dst` nodes. The implementation of this methods starts by copying `Dijkstra`'s bundle internal graph and removes all `edges`, then tries to get a path between `src` and `dst` nodes, returning an alternative path from the current graph. This is only used for implementation 1 as it was needed to find two disjoint paths between two nodes.

The second method, `getRouteWithoutSingleEdges`, returns a hash map with a list of available paths for each node connector between the node where `srcNodeConnector` node connector is attached and `dst` node. On `Dijkstra`'s bundle internal graph edge has an head and tail node connector as seen in Figure 4.2. On the illustrated graph, `T1` from node `1` is the `srcNodeConnector` and node `3` is the `dst` node. The implemented code for this method starts by removing the first edge from the given list of edges `edges` from `Dijkstra`'s bundle internal graph, on this example the first edge is the one connecting node `1` and node `4`. Then calculates a path (if available) between the node where the `srcNodeConnector` node connector is attached and the `dst` node, on this case would be $\langle S1, S2, S3 \rangle$. Afterwards, it puts in the hash map, to return the path found for that node connector. The process is then repeated for every tail node connector from the given list of edges as the `srcNodeConnector` and

`dst` node being the same. The end result will essentially be a list of alternative paths for every edge failure from the primary path in the topology, the gist of implementation 2.

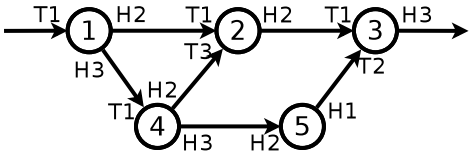


Figure 4.2: Representation of Dijkstra's internal graph with head and tail connectors for each node. (H# represents a head node connector for that edge and T# represents a tail node connector for that edge.)

Although this bundle has a method on its interface to give a route with a particular bandwidth, the same was not implemented. If this bundle had notion of the bandwidth traffic circulating in the topology, it would be possible to have a proper load balancing for the links.

The code changed and produced is available online in [4].

4.2 IMPLEMENTATION 1

All the described bundles in the previous sections are used on all the developed implementations. After OpenDaylight starts, the `topology-manager` creates a topology when it receives LLDP messages and triggers the `Dijkstra`'s bundle to update its internal graph. When a communication between 2 hosts start, the packet reaches the node and since it does not have any rule associated to the node, contacts the controller. When the controller receives that unmatched packet the implementations are notified by calling `onPacketReceived` method. The activity diagram for this method on this implementation is the one represented on Figure 4.3 as well a more detailed explanation.

When a link is removed, the `topology-manager` receives a port down message that will trigger the Dijkstra's bundle that will trigger the `recalculteDone` method also explained later on this section. The sequence diagram for this event is described on Figure 2 from Appendix A-Sequence diagrams related to the implementations on OpenDaylight.

4.2.1 UNMATCHED PACKET PROCESS

Since there is one instantiation of `onPacketReceived` for each unmatched packet received by the controller, it was necessary to create a collection of semaphores to prevent the controller from recalculating a path while the same criteria for the rules was used. Thus, when an unmatched packet is received, the first thing done is a calculation of an hash value based on the node ID where the packet came from, the source MAC address and the destination MAC address for that unmatched packet.

[4]https://github.com/aanm/multipath-openflow

That hash is used to try acquiring a semaphore from the collection of semaphores, each one for a different match. The reason between trying for it and waiting indefinitely is mainly because in the event of a calculation of a route is already being processed, it means all nodes from that path are being configured. Therefore the acquisition will be more likely to fail at this point for the remaining unmatched packets. In case of failure for the first semaphore acquisition, the implementation tries one more time but with 1 second timeout. Having a timeout prevents the delay for being longer than 1 second for each unmatched packet to be processed, the trade off is after that 1 second passes, the packets are discarded and therefore lost.

On both cases when the semaphore is acquired, the implementation searches on cache if there was any rules for the given node ID, source MAC address and the destination MAC address. If the cache does not contain any rules it means it is necessary to create them as it will be explained after activity diagram from Figure 4.4. After the rules are successfully found on cache, the implementation obtains them.

On the left side of the activity diagram of Figure 4.3 is represented the case where a semaphore was acquired at the first try but the rules were not present on cache, this means an unmatched packet was sent by the node when a rule expired from it but is still present on cache. When this happens, the implementation sends the rules for the backup path and only after sends the rules for the primary path. Both procedures of those activity diagram are present on Figure 4.7 and Figure 4.5 and will be explained in the following.

The reason for the backup path is programmed first is to prevent, in case of a link failure on the primary path while sending the rules, the backup path's nodes to contact the controller. The way it is done, if that failure occurs, the backup path's nodes are already prepared to send the incoming traffic from the primary path's nodes.

On the right side of the diagram is represented the case where a semaphore was acquired at the second time, meaning the implementation was already calculating and sending the rules for the primary and backup path for the source MAC address and destination MAC address. Since those rules were sent for every node, there is only need to resend the rule for the node that contacted the controller.

On both cases, if the rules were found on cache and also to prevent a delay for the following unmatched packets that want to acquire the semaphore, there is no need to wait for confirmation of rules received from by the nodes. Thus, the release of the semaphore is done immediately.

At the beginning of the activity diagram represented on Figure 4.4 the rules were not present on cache therefore is required to create them.

First is necessary to contact the `hosttracker`'s bundle in order to find out on which node connector the host with the destination MAC address is connected. If it is not found the semaphore is released and the process stops for this unmatched packet.

Second, the `Dijkstra`'s bundle is used to find the shortest path between the source node, where the unmatched packets are coming from, and the destination node, where the destination MAC address is directly connected. If a path was not found it does not mean that the hosts could not communicate

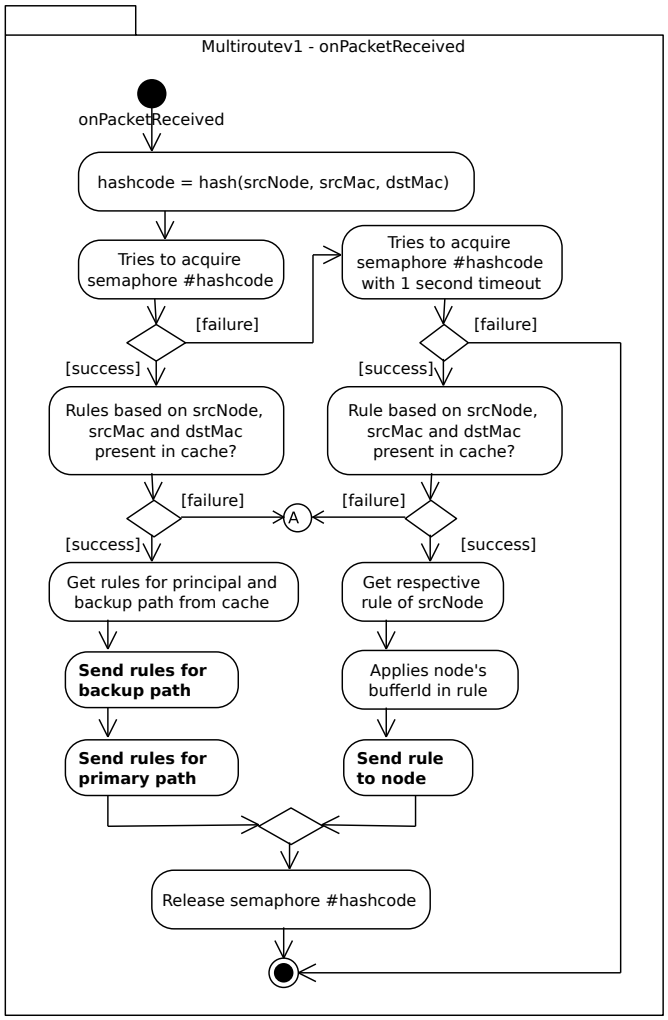


Figure 4.3: Activity diagram when an unmatched packet is received for implementation 1.

with each other. For that reason it is necessary to verify if the source and destination node are the same, which technically means there is no path on a vertex and the hosts are connected to the same node. Thus, there is only need to apply the `buffer_id` value received from the node on the rule for that node, send it and release the semaphore.

If a path was found, it will become the primary path for the communication between those hosts. Since on this implementation the backup path must be disjoint from the primary path, is necessary to find that disjoint path. That will be the function of `getRouteWithoutAllEdges` method. If a disjoint path is found, it will be the backup path for this communication. This path as well the rules associated to it, will be saved on cache for the future unmatched packets received by the controller and those rules will be sent for every nodes on the backup path.

If a disjoint path was not found, it will be only a single path for the communication between those hosts. Similar to the backup path, the rules from the primary path as the path itself, will be saved on cache before sending the rules for every respective nodes present in the primary path.

Afterwards, the implementation will wait for a confirmation that every rule was sent for the

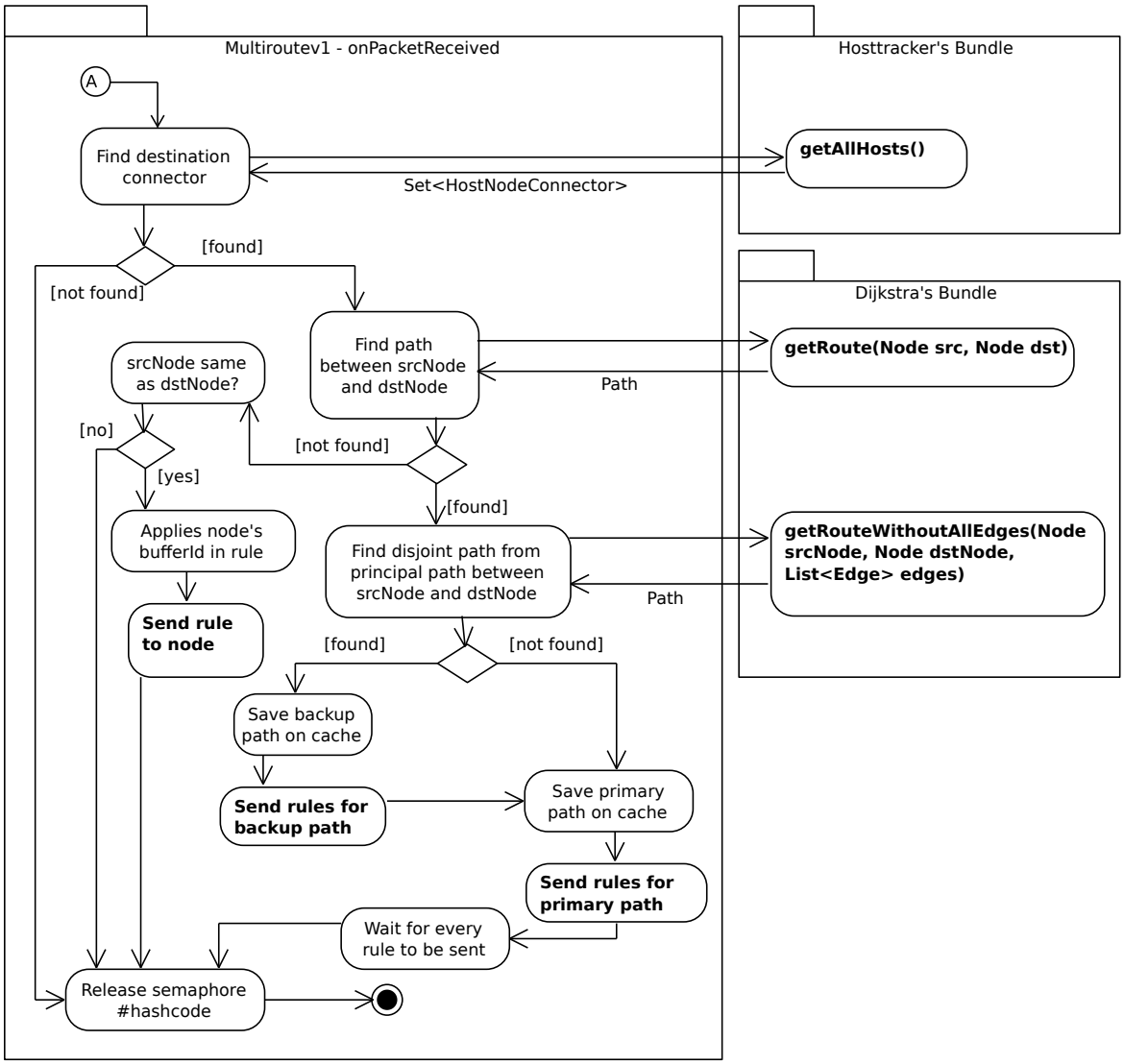


Figure 4.4: Activity diagram when an unmatched packet is received for implementation 1 (Continuation).

primary and also, if exists, for the backup path. Finally the semaphore will be released so the other unmatched packets that have the same criteria for the rules' match could be processed.

4.2.2 PRIMARY PATH RULES PROCESS

The `sendFlowsForPrimaryPath` method, which its activity diagram is represented on Figure 4.5, is used to create and send the rules for the nodes on a given path. The first thing this method checks for is the absence or existence of a backup path. If a backup path does not exist, the method starts by creating the rules that match a given source and destination MAC address as well the incoming port from where the traffic comes from. Each match has a respective action that redirects the matched traffic to the port that is directly connected to the next node from the given path.

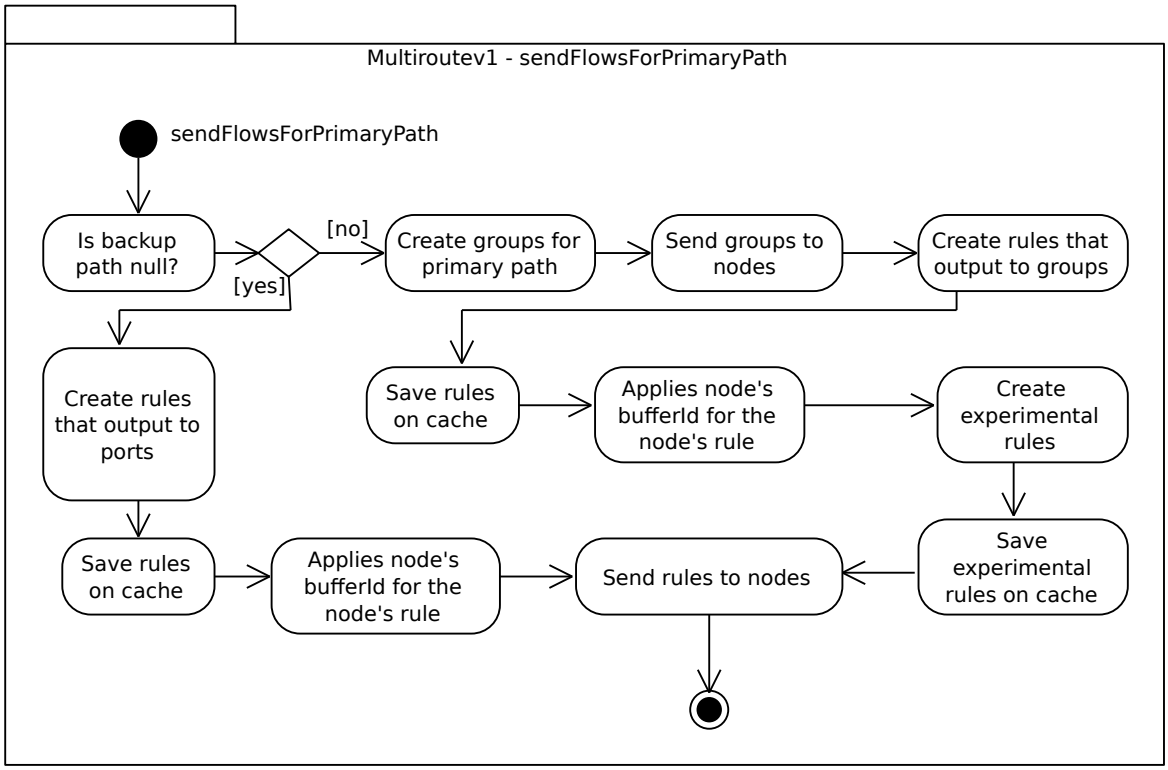


Figure 4.5: Activity diagram of primary path's rules configuration for implementation 1.

If there is a backup path, it is necessary to create OpenFlow groups for the primary path. On this implementation a group from the primary path will always have a first port, the one that directly connects to the next node from the primary path, and a second port, the one where the traffic came from. Supposing the primary path from Figure 4.6, located between node 1 and 3 is $\langle 1, 4, 5, 3 \rangle$, and the backup path is $\langle 1, 2, 3 \rangle$, on node 4 and 5 the groups created will have a first port #3 and a second port #1, and for nodes 4 and 5 a first port #1 and a second port #2. This means that if the link between node 4 and node 5 or node 5 and node 3 fail, the traffic begins to go the way back. On node 1 its first port is port #3 and its second port #2, the one that is directly connected to the second node of the backup path, being node 1 the first node.

After the controller constitutes the groups it is necessary to send them before the rules or, on the other hand, if it was sent an OpenFlow rule with an action that would send the traffic to a inexistente group (in the node), the nodes would respond with an error, `OFPBAC_BAD_OUT_GROUP`, specified by OpenFlow [79].

Once the groups are sent to the nodes, the rules are created, based on their source and destination MAC address, as well as the ingress port, from where the traffic is prevenient, and then act to output that traffic to the respective group (created previously). To prevent an increase of the nodes' memory, the primary path's rules will have a 30 second `idle_timeout` that will cause an automatic deletion 30 seconds after the traffic stops. After being created them, they are saved on cache for future use.

Afterwards, the same `buffer_id` value, received by the controller from the unmatched packet, is

applied on that specific rule for the node were the unmatched packet came from.

Only on this implementation was an experimental feature from Open vSwitch designated by `learn` tested. This feature allowed the Open vSwitch to reprogram itself without contacting the controller. In alignment with the example stated on Figure 4.6, `learn` rules can only be sent to the nodes' primary path. As explained previously, in case a primary port fails the traffic is automatically sent to the second port from that group. In the previous example, where the primary path is ⟨1, 4, 5, 3⟩, if ⟨5, 3⟩ link fails the traffic will be redirected to node 4 and then 1, as expected. To prevent the traffic from following that path, because one of the links' path is broken, reporting to the situation where it is installed on node 1 and 4, the `learn` rule will reconfigure node 4 to send the traffic coming from port #1 to that same port, instead of sending to port #3. That reconfiguration happens without the controller's help because the `learn` rule, installed on node 4, had the following condition: if there is traffic coming from port #3, with the origin and destination MAC address the same that were sent to that port #3, meaning there was a disruption ahead on this path, the node should stop sending traffic to port #3 and start sending to port #1. On node 1 the `learn` rule was similar to the one installed on node 4. When the traffic was received from port #3, instead of reconfiguring itself to start sending the traffic back to port #1, like it would happen on node 4, it reconfigured to send the traffic to port #2, the port directly connected to backup path. This is only a temporarily situation because once the controller receives the port status from node 5 and 3 informing the port is down, the paths are recalculated, as well the rules for that paths' nodes, and then they are sent to the nodes, overlapping the temporary rules previously installed by `learn` rules.

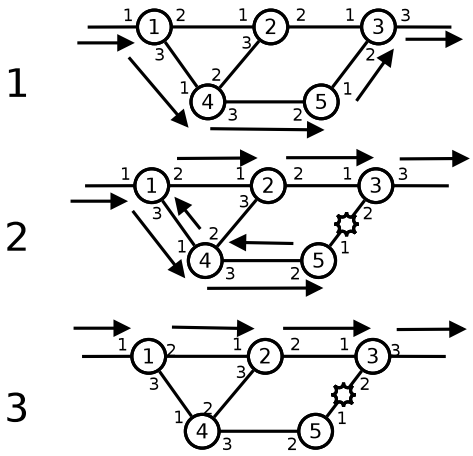


Figure 4.6: Topology behavior by using implementation 1 approach upon a disruption on ⟨S5, S3⟩ link without contacting a controller.

Now, proceeding with the activity diagram analysis, after the creating of the experimental messages, they are also saved in cache for future unmatched packets that are received by the controller.

Finally, all rules are sent to the respective nodes, while installing every rule on the specific node with no particular order and leaving only the rule for the node that contacted the controller to last. This solution prevents other nodes from the primary path to contact the controller for unmatched

packets in cases in which the first node received a rule previously to other nodes and forwarded the matched packet.

4.2.3 BACKUP PATH RULES PROCESS

Although simple, the method in charge of sending the rules for the backup path, with its activity diagram in Figure 4.7, is no less important. If a backup path exists, the method starts by creating the respective rules from that path's nodes. On the example from Figure 4.6, the backup path was ⟨1, 2, 3⟩. Since this is the backup path, the rules installed on those nodes are simpler than the ones of the primary path. The rules will simply have to output the incoming packets to the port that is directly connected to the next node from this path.

Unlike the rules from the primary path, these rules do not have any `idle_timeout`. It was omitted to prevent them from expiring before the backup path was used. If the rules did not exist on the backup path, when the primary path would be disrupted, the nodes from the backup path would start requesting the controller for rules for the incoming unmatched packets, causing delays and maybe loss of those packets.

In the end, those rules are also saved on cache for upcoming unmatched packets in the future. The rules are sent to those nodes and since it is the backup path they are sent with no particular order.

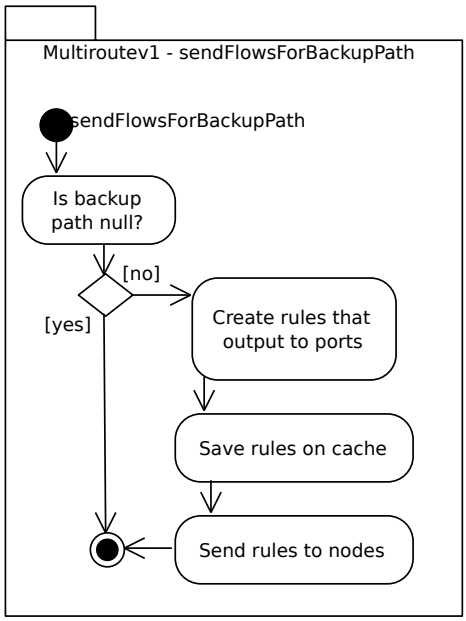


Figure 4.7: Activity diagram of backup path's rules configuration for implementation 1.

4.2.4 TOPOLOGY CHANGE PROCESS

As seen in sequence diagram from Figure 1, present on Appendix A-Sequence diagrams related to the implementations on OpenDaylight, after a topology change, `topology-manager` triggers the `Dijkstra`'s bundle. After updating its internal graph it calls every bundle that is registered on AD-SAL so they can know when `Dijkstra`'s has finish its recalculation. The interface implemented by this implementation, called `IListenRoutingUpdatesWrapper`, has a method designated by `recalculateDone(List<Edge> edgesChanged)`, the one called by Dijkstra's when it has finished it is recalculation.

Once this method is called, where its activity diagram is represented on Figure 4.8, the implementation starts finding, on its cache, which paths, designated by routes[5], were affected by the list of edges modified by the topology disturbance. Afterwards it will run the method `recalculateRoutes`, explained in the following.

Then, it will get the remaining paths, also from cache, that were not directly affected by the modified edges. The reason for doing this, is to make sure in case a new path is formed between two nodes, that new path will be the primary or the backup path between those nodes.

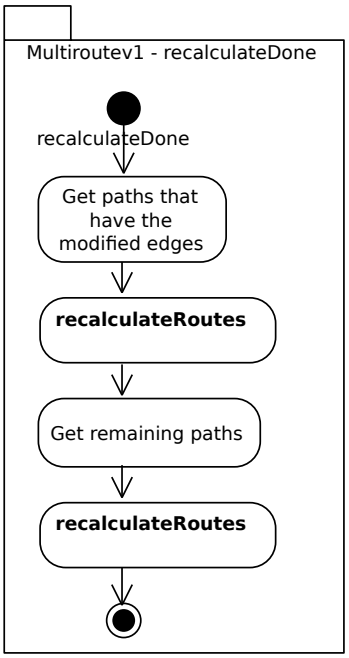


Figure 4.8: Activity diagram when a disturbance occurs in the topolgy for implementation 1.

Having a list of pairs of hosts and now that `Dijkstra`'s bundle has its graph updated, the method `recalculateRoutes`, represented on Figure 4.9, calls `getRoute` method from `Dijkstra`'s bundle to get a path between the nodes where each host is directly connected. If a path is not found, there is nothing to do for those nodes, meaning that all pairs of hosts connected to those nodes will have their communications interrupted.

[5]A route is considered a primary and, when exists, a backup path that have one or more pairs of hosts using them.

After finding a path, it is necessary to check if there is a backup path disjoint from the primary path. At this point is irrelevant if a backup path was or was not found because is more important to check if the old primary path is the same as the new backup path found. Since is relevant to prevent unnecessary disruptions and path changes for a communication, only if the new backup path is the same as the old primary path this new backup path will be the new primary path and the new primary path will be the new backup path. This means the old primary path will be the same while the new backup will not be the same as the one used before the disruption.

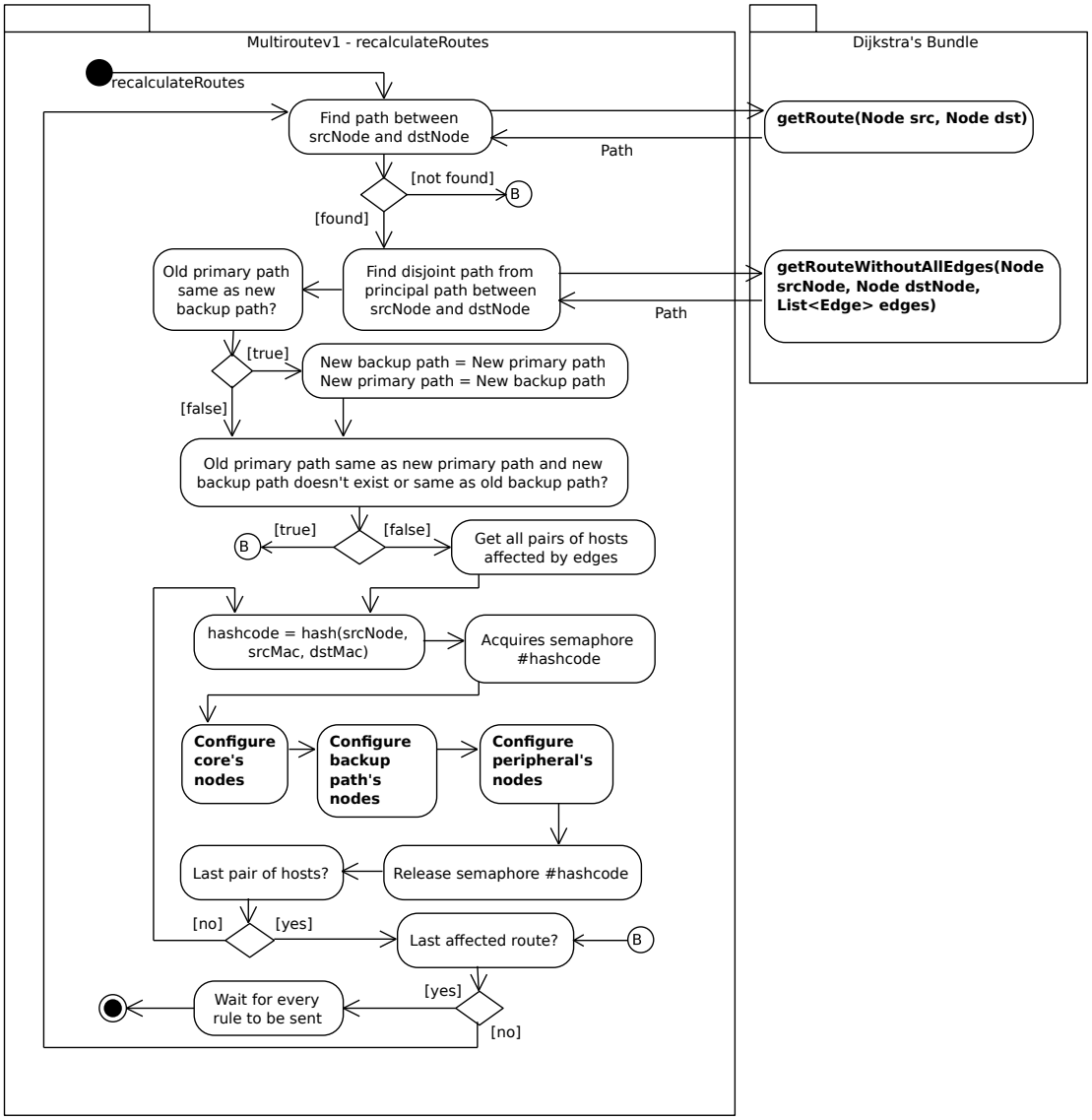


Figure 4.9: Activity diagram when a disturbance occurs in the topolgy for implementation 1 (Continuation).

Next, if the primary path has not change nor a new backup path was found or has not changed as well, is not necessary to create and resend rules for the nodes from those paths. If this is not the last route to be recalculated, the process repeats itself.

Since there could be multiple pairs of hosts using the same path, is necessary to get them from cache and, for each one of them, acquiring their respective semaphore based on the source node, the

source and destination MAC address. Is fundamental to not forget there could be multiple unmatched packets trying to acquire this semaphore as well. Because the process where a topology has suffered a disruption and is more critical than the arrival of unmatched packets, the acquisition of this semaphore does not have a timeout.

Upon acquisition of this semaphore, the rules are created in a similar way when an unmatched packet arrives with a difference in the order they were created. Supposing there was a new primary path found, if the rules created for the nodes were sent by the same order of the path, this could bring loss of packets since the old rules would be overwritten by the new ones. At first this could be meaningless but supposing the new rules would send the traffic to an unprepared node, dealing with that type of traffic could bring some chaos over the topology and consequently in the controller. Thus, the first programmed nodes are the ones from the topology's core, after the nodes from the backup path and finally the peripheral's nodes. Noticeably, while programming the core's nodes the same problem could happen but it will not since the nodes are programmed on the communication's reverse direction.

Right after the nodes are configured, the semaphore is released and if it is not the last pair of hosts from the given paths, the rules configuration process repeats itself. If it is the last pair of hosts affected, then is checked if are any more routes to reconfigure. If they exist, they will be reconfigured, if they do not exist, the method will wait until every rule is sent before terminate.

4.3 IMPLEMENTATION 2

The approach performed on this implementation was slightly different from the previous one. On implementation 1 was necessary a disjoint path to have backup. Taking in consideration not all topologies have two disjoint paths between every pair of hosts, was necessary to think on a new approach for this kind of topologies.

Thus, with the help of `getRouteWithoutSingleEdges`, is possible to have a backup path for every link failure in the primary path. With the same topology previously used and represented on Figure 4.10, continuing to assume the primary path is $\langle 1, 4, 5, 3 \rangle$, on implementation 2 there will be multiple backup paths. If link $\langle 1, 4 \rangle$ fails, the backup path will be $\langle 1, 2, 3 \rangle$, if link $\langle 4, 5 \rangle$ fails, the backup path will be $\langle 4, 2, 3 \rangle$ and if link $\langle 5, 3 \rangle$ fails the backup path will be $\langle 5, 4, 2, 3 \rangle$. This way every link, from primary path, is protected against failures and contrary to what may appear, the nodes will not suffer with countless rules because every backup path's nodes will have overlapping rules.

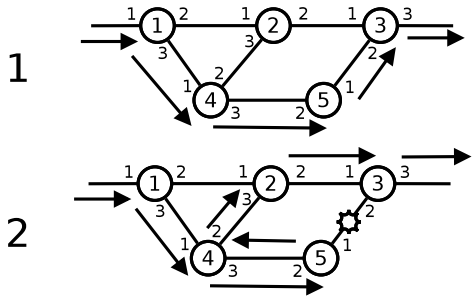


Figure 4.10: Topology behavior by using implementation 2.1 approach upon a disruption on $\langle S5, S3 \rangle$ link without contacting a controller.

4.3.1 UNMATCHED PACKET PROCESS

Similar to implementation 1, this method, with its activity diagram illustrated on Figure 4.11, starts as well by trying to acquire a semaphore based on a hash code from source node, source and destination MAC address. Unlike the previous implementation, when the semaphore is acquired,the first thing done is to get the node connector where the destination MAC address is connected. Doing this on an earlier stage prevents a possible problem, present on the previous implementation, where a host changes its node connector. Despite this prevention, there is still the problem if a destination host changes its node connector while there are traffic directed to it. For future work it would be a good idea for the new `hosttracker`'s bundle [81] to have a method that could trigger listening bundles when hosts changed their node connector. This way it would be possible for this implementation to reconfigure a new path and redirect a traffic to the new node connector.

If the MAC destination is not found on the topology, the process releases the semaphore and finishes. If it is found, the implementation will search, on cache, for rules that have the same source and destination node connector as well the source and destination MAC address. If it is not found, only the rule for the node that contacts the controller for the unmatched rule will have its `buffer_id` value applied on the rules designated to it. Both primary and backup rules are resent for every node, the methods waits for them to be sent, releasing the semaphore upon sending.

When they are not present in cache, means it is necessary to find and create the rules for that pair of hosts. Similar to implementation 1, it gets a path for the given source and destination node, where the hosts are connected, from `Dijkstra`'s bundle. Like explained before, the path from `Dijkstra`'s bundle return an empty path if both source and destination node are the same. On this case is only necessary to verify it and apply the node's `buffer_id` value for the rule that will match the unmatched packet received in the controller. Those rules are then sent for that node releasing the respective semaphore at the end.

After finding a path, the `getRouteWithoutSingleEdges` method is used to return an hash map with the node connectors as keys and a list of paths as values. That hash map as long with the primary path are processed by the `programNodes` method, explained as following. After the nodes are programmed, this method waits for them to be sent and finally releases the semaphore acquired at the

beginning.

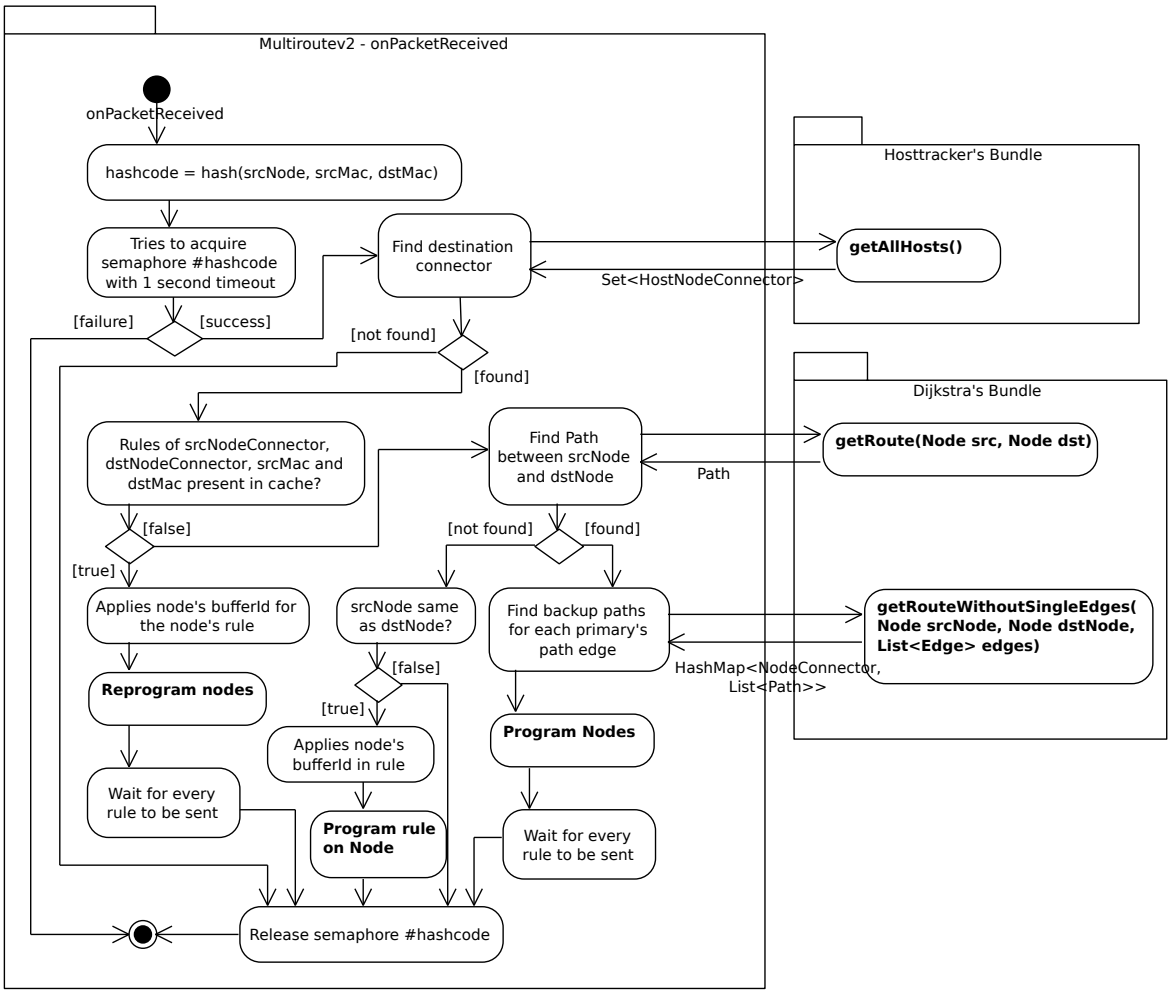


Figure 4.11: Activity diagram when an unmatched packet is received for implementation 2.

4.3.2 PRIMARY PATH RULES PROCESS

The `programNodes` method have the same effect of `sendFlowsForPrimaryPath` and `sendFlowsForBackupPath` methods from implementation 1 as well it is more simpler. This method's activity diagram is represented on Figure 4.12.

Its first task is to find out which nodes will have overlapping rules. Thanks to the hash map from the `getRouteWithoutSingleEdges` method this task is easier because there is only need to take into account which node connectors are present on the primary path and on the key set of the hash map.

For the node connectors that are on both places it means that there is possibility to create groups. The first port of a group will be the egress node connector from the primary path, the second port will be the first egress node connector from the first path of the list of paths, being that list the respective value for the ingress node connector from the primary path, used as key on the hash map.

Same as implementation 1 the groups are first sent for the nodes than the rules. Then the implementation applies the node's `buffer_id` value for the rule that will be applied on that node. Finally the rules from the primary and backup paths are written on the nodes. As well in implementation 1, the rules are sent in no particular order being only the node that contact the controller the last one to receive the rule.

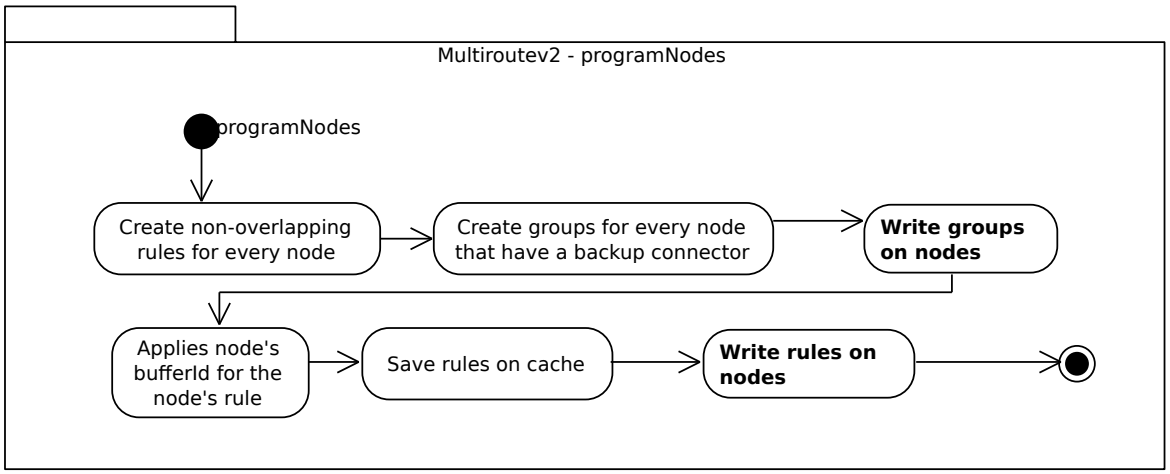


Figure 4.12: Activity diagram of rules configuration for implementation 2.

Unlike implementation 1, only the border nodes have an `idle_timeout` value being the core nodes rules permanent. Also, since the implementation has full control of the nodes as well their rules installed, only new rules are sent for them, the process to know if a rule is new or not will be explained on subsection 4.3.4 - Topology change process.

4.3.3 RULES TIMEOUT PROCESS

One of the features implemented was the functionality of cleaning all the rules from the nodes after N seconds of traffic absence, the sequence diagram for this event is on Figure 3 from Appendix A-Sequence diagrams related to the implementations on OpenDaylight. When an `idle_timeout` value from a rule reaches to 0 seconds, that rule is removed from the node and the node informs the controller about that removal. Since this implementation is listening for rule removal messages, the method `onSwitchFlowRemoved`, where its activity diagram is represented on Figure 4.13, is triggered every time the controller receives that notification.

That notification contains, among other things, a `cookie_id`. Since every `cookie_id` is saved on cache for a pairs of hosts, is easier to trace back and know which rules are being used for a given pairs of hosts.

Because there will be changes in the nodes, it is necessary to acquire the semaphore related to the source node, source and destination MAC address. After acquisition, all the rules, including the ones installed on backup nodes, are removed from the cache. Those removed rules will return their

`cookie_id`. Having that list of `cookie_id` values it will be possible to delete them from the respective nodes.

After all rules related to the pair of hosts are deleted from the respective nodes, the semaphore is released.

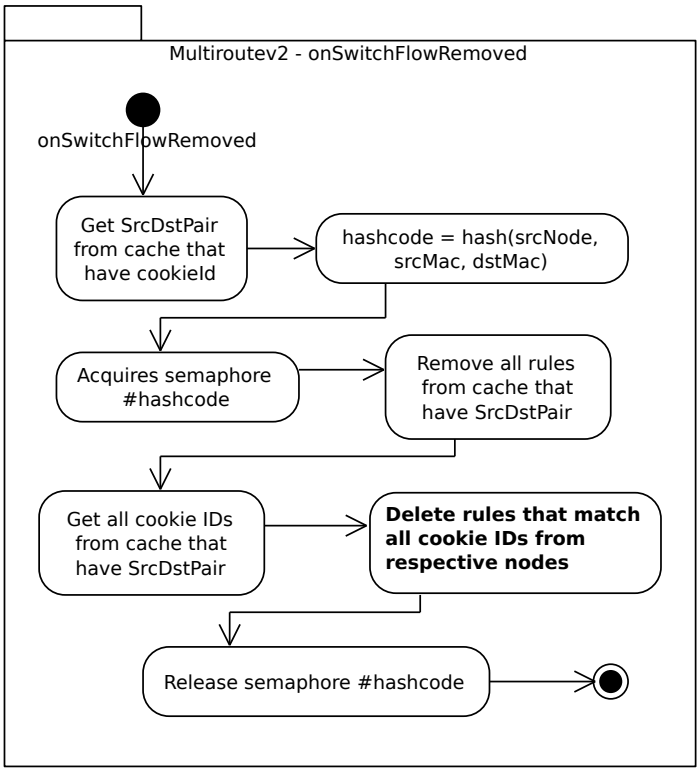


Figure 4.13: Activity diagram when a rule reaches it is timeout for implementation 2.

4.3.4 TOPOLOGY CHANGE PROCESS

Being the recalculation of new paths and respective rules when a disruption occurs in a topology the main purpose of this work, the activity diagram for this situation is represented on Figure 4.14, called `recalculateDoneTopo`. Similar as implementation 1, this only recalculates the rules when is triggered by `Dijkstra`'s bundle but, unlike implementation 1, the method is not the same. As an improvement, this implementation does not rely only on a list of edges changed, it takes in consideration as well if at least one edge from the topology was removed. The reason for checking this is quite important as a performance improvement. If an edge is removed, there is only need to reconfigure the pairs of hosts affected by that route. If an edge is added, is necessary to check every pair of hosts as that addition may have created a new path between a pair of hosts.

Having a list of hosts pairs that could have their primary path disrupted, the implementation forks itself, the new thread will deal with the list of pairs of hosts while the other will finish. The reason

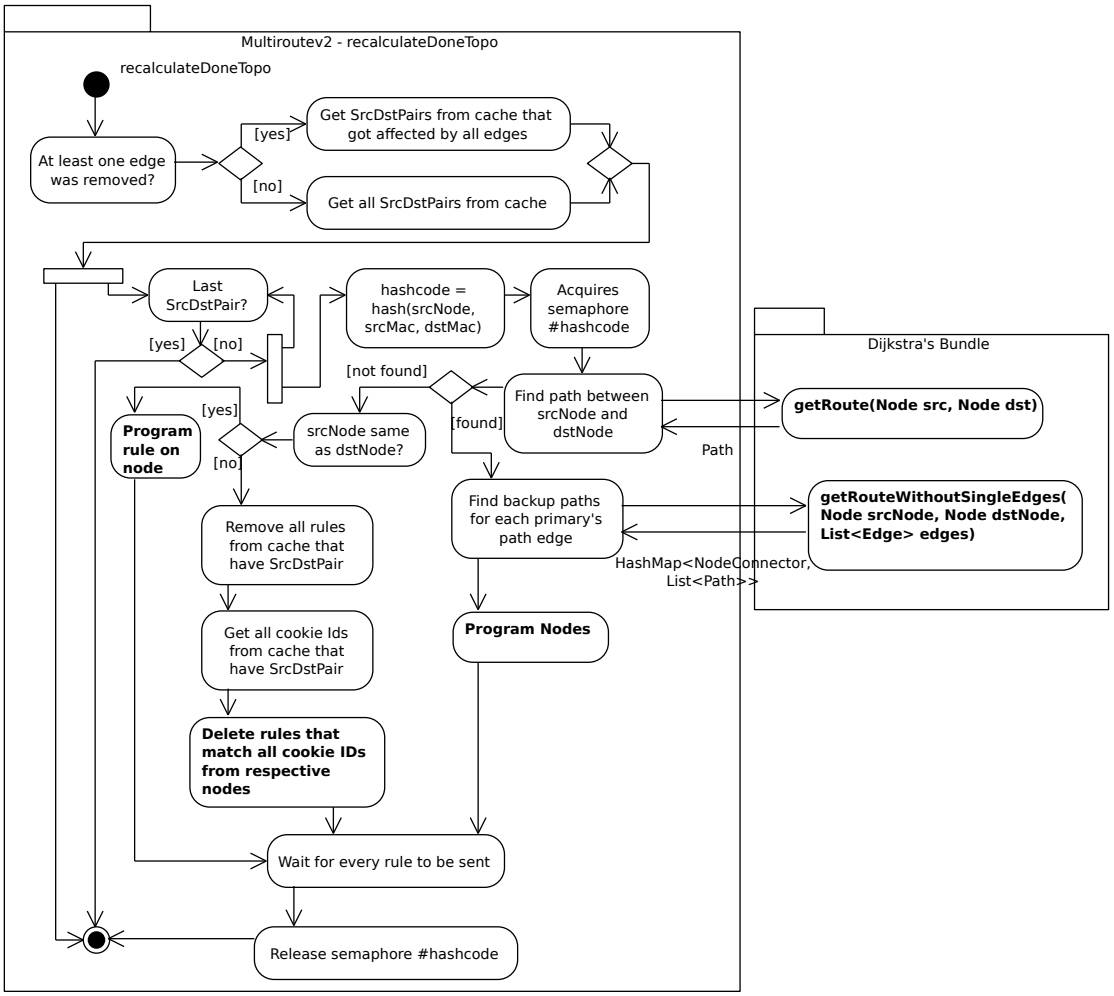


Figure 4.14: Activity diagram when a disturbance occurs in the topology for implementation 2.

for this fork was to prevent the `Dijkstra`'s bundle waiting until this method finishes, since this is a blocking method on Dijkstra's bundle.

Continuing on the new thread, it will check if there are remaining pairs of hosts needing to have their paths recalculated. While iterating over all pairs, the thread itself will fork for every pair. For this new thread, the first thing to do is to acquire the semaphore based on the source node as well the source and destination MAC address. The process to find and reconfiguring a new path for the pair of hosts is exactly the same as the one described in the reconfiguration of nodes when an unmatched packet reaches the controller.

Since the controller has total control of the nodes in the topology, it is not necessary to resend every rule when a disruption occurs in the topology. Since the `cookie_id` value of the rules is based on an hash value from the ingress node connector, source and destination MAC address as well the output action for this match, is easier to control when a rule created is different from the one present on the node. Supposing there is a rule on cache with the `cookie_id 0x12345`, based on source MAC address `00:00:00:00:00:01`, on destination MAC address `00:00:00:00:00:02`, on ingress node connector `1` and on the output action `group:2` and that same rule is installed on node 1. If a disruption occurred,

causing node 1 to only have an option to reach the machine with MAC address `00:00:00:00:00:02`, passing by port #5, the output action would be different from `group:2` to `port:5`. Changing this output action cause the value from `cookie_id` to change from `0x12345` to `0x11111`. Verifying this value is different from the one saved on cache, is necessary to send this rule to the node as well referring this `cookie_id` to the match source MAC address `00:00:00:00:00:01`, destination MAC address `00:00:00:00:00:02` and ingress port `1`.

If a primary path was not found it is necessary, as explained before, to check if the source and destination nodes are the same. If they are not, the rules are removed from cache and from the nodes. The reason for this deletion is to prevent the traffic to go through on a broken path if a new path is found afterwards.

The semaphore is released after waiting for rules to be sent for the nodes.

4.4 IMPLEMENTATION 2.1

Implementation 2.1 suffered one minimal, but relevant, modification on the source code based on implementation 2. While on implementation 1 and implementation 2 the rules had their match: the ingress node connector, the source and destination MAC address, on this implementation it was omitted the match for the source MAC address, where the traffic has its origin. Is possible to predict this will bring some performance improvements on the controller as well the decreasing number of rules installed on the switches.

4.5 CHAPTER'S SUMMARY

In this chapter was described the algorithms and the principal methods used for every implementations. It was also presented some future work for some of the OpenDaylight bundles used.

On Table 4.1 are described the main functionalities and restrictions for each one of the implementations.

Functionalities/Restrictions	**Implementation 1**	**Implementation 2**	**Implementation 2.1**
Needs a disjoint path	Yes	No	No
Multi-threading	No	Yes	Yes
Matching for rules	Port in, MAC Source and MAC Destination	Port in, MAC Source and MAC Destination	Port in and MAC Destination
Cleaning rules after *N* seconds of traffic absence	Only border nodes and from primary path	All nodes	All nodes
Sending rules after a topology change	Resend every rules	Resend new rules only	Resend new rules only

Table 4.1: Overall of the functionalities and restrictions for all the implementations developed.

CHAPTER 5

Evaluation

This section presents how the tests for the described implementations on the previous section 4 were performed. In the end, some discussion and conclusions are presented based on the obtained results from those tests.

Three different testbeds were made for the developed implementations. The first was constituted by a mesh topology where multiple computers were connected in a way where multiple paths for every pair of hosts existed. With this testbed it will be possible to determine the number of packets lost every time a link goes down and the amount of time to reestablish a communication after completely cutting it off.

Since the first implementation has the requirement of having two or more disjoint paths in order to have a proper fast-failover mechanism, the second test has a single point of failure on its topology. When one of the other redundant links were removed, it was possible to acquire enough data to compare how drastic it is an implementation requirement of having two or more disjoint paths for every communication.

The third and last test was designed to test the scalability of the solutions implemented over a large number of hosts. The number of rules every time new hosts were added to the topology was measured as well as the amount of time the controller took to reconfigured every directly affected and non-affected paths when a disturbance occurred in the topology.

As it was mentioned in the previous section 4, all three implementations were developed for OpenDaylight. This controller was running on a machine with an Intel i5-4200M CPU @ 2.50 GHz and 8 GB of RAM (only 1 GB was given to OpenDaylight), running Fedora 20 x86_64 with Java "1.7.0_51". Mininet's topologies were running on another machine with an Intel Core2 Duo CPU T6600 @ 2.20 GHz and 4 GB of RAM running Kubuntu 13.10 x86_64. Mininet was used to emulate various network topologies.

The machines involved in the tests were directly connected by an Ethernet cable. On each machine a delay of 15 ms was set for every outgoing packet. With those 30 ms delay it is a fair assumption to simulate, on a worst case scenario, for a switch to reach out a server where a controller is running. On every topology tested a 1 ms delay was set on every link, also to simulate the closest to a real environment.

The topologies presented in the figures on this section will have nodes and computers, where every node represents a switch and every computer represents a host. Those components are connected by a solid line that represents a virtual link between each other. The numbers outside the nodes represent the respective port's numbers for that link.

The OpenDaylight offers lots of bundles with vast features but not all of them were needed for this dissertation purpose due the amount of CPU usage which was increasing with an increasing number of rules, making it necessary to stop some non-essential, statistics related, OpenDaylight bundles. All those non-essential bundles stopped were statistics related, were as follow:

- `org.opendaylight.controller.statistics.northbound_0.4.2.SNAPSHOT`
- `org.opendaylight.controller.statisticsmanager_0.5.1.SNAPSHOT`
- `org.opendaylight.controller.statisticsmanager.implementation_0.4.2.SNAPSHOT`
- `org.opendaylight.controller.model.flow-statistics_1.1.0.SNAPSHOT`
- `org.opendaylight.controller.md.statistics-manager_1.1.0.SNAPSHOT`

5.1 MESH TOPOLOGY

On this first test, ICMP echo requests were sent every 100 ms, from hosts h1 to h2, h1 to h3, h1 to h4, h2 to h3, h2 to h4 and h3 to h4. The disposition of the hosts as well as the topology itself is illustrated on Figure 5.1. After all communications had stabilized, the $\langle S12, S9 \rangle$ link was removed. Then, after 15 seconds, $\langle S12, S13 \rangle$ link was removed and finally the $\langle S12, S16 \rangle$ link. When this last link was removed, all communications between h1 and h2, h1 and h4, h2 and h3, h3 and h4 ceased to exist.

To test a restoration scenario, where a new link is added or re-added, the $\langle S12, S9 \rangle$ link was attached again in the topology. The purpose of this was to see how much time it takes for a communication to restart after a breakdown.

Although all hosts are communicating with the others, it will only be analyzed the communications which suffer a major impact every time a change in the topology was made. On this case it will be h1 $\leftrightarrow$ h2 and h1 $\leftrightarrow$ h4.

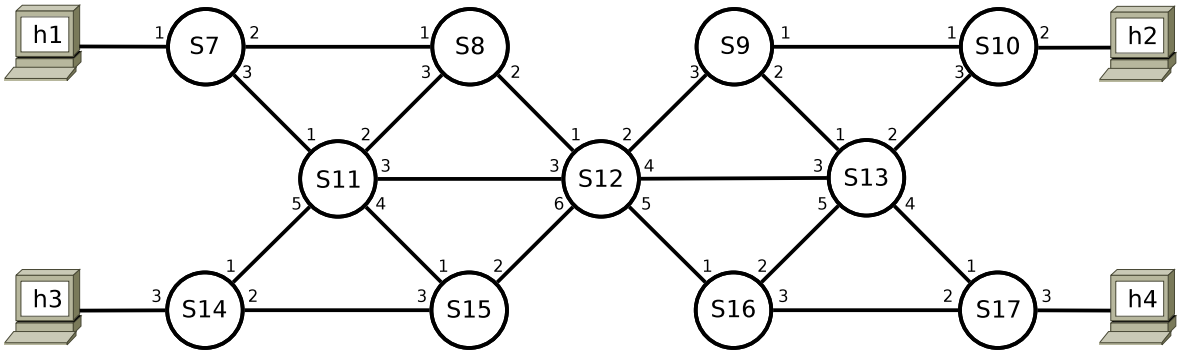


Figure 5.1: A mesh topology with hosts having multiple paths to reach the remaining hosts.

5.1.1 IMPLEMENTATION 1 - H1 ↔ H2

When the communication started, the primary path for these hosts was ⟨S7, S8, S12, S13, S10⟩ (both in request and reply) and the backup path was ⟨S7, S11, S12, S9, S10⟩ (also both in request and reply). On Figure 5.2 is represented a plot where it is marked the disturbances that were wittingly made on the topology and the end-to-end delay for every packet sent on this communication. When ⟨S12, S9⟩ link was removed this communication did not suffer a major disturbance because the link removed belonged to the backup path. After the controller received the notification stating this link was removed, the controller changed the backup path to ⟨S7, S11, S12, S16, S13, S9, S10⟩, disjoint from the primary path, for the ICMP requests and the inverse path for ICMP replies.

Around 60.027 seconds from the plot represented on Figure 5.2, after removing ⟨S12, S13⟩ link, the first ping afterwards suffered a massive delay in contrast with the previous ones. This occurred because as soon as the switch S12 detected that its port #4 was down it sent that ping to the previous switch, S8. The S8 switch did the same thing and sent it to the previous one, S7. The way the implementation was created, those 3 switches (S7, S8 and S12) reprogrammed themselves, with the `learn` action, preventing more traffic to go trough this path. After S7 received the traffic from S8, S7 started to send the traffic to the second switch from the backup path, S11 (being S7 the first one).

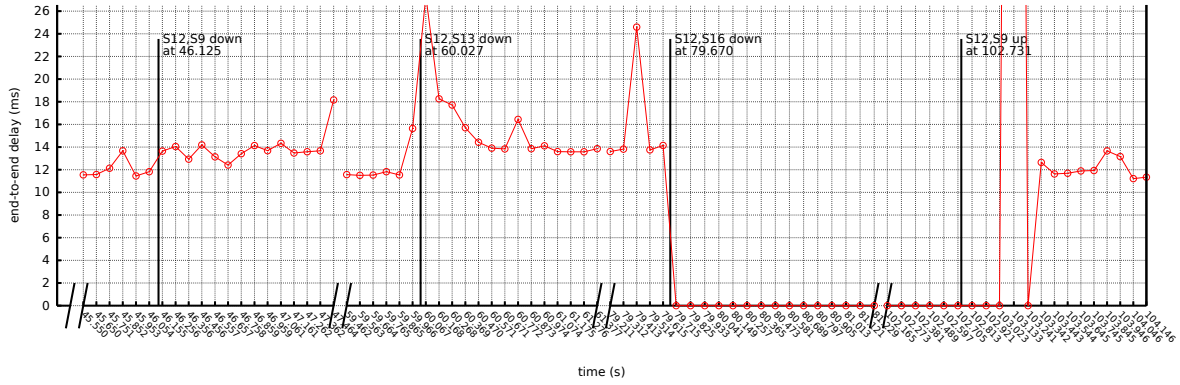


Figure 5.2: End-to-end delay for h1 and h2 communication when the different disruptions were made to the topology.

A similar thing happened on the reply with S13 and S10. After switch S10 received traffic from

S13 with the source h2 and destination h1, S10 started to send the traffic to the second switch from the backup path, S9 (being S10 the first one).

The communication stabilized after receiving a new path from the controller. Despite the lack of a disjoint path, the controller sent a different path ($\langle S7, S8, S12, S16, S13, S10\rangle$) than the one that was being used temporarily ($\langle S7, S11, S12, S16, S13, S9, S10\rangle$).

With no surprises, when the $\langle S12, S16\rangle$ link was removed the communication between h1 and h2 ceased to exist.

After the $\langle S12, S9\rangle$ link was re-added, the traffic between h1 and h2 took around 611 ms to stabilize.

5.1.2 IMPLEMENTATION 1 - H1 ↔ H4

The primary and backup paths for h1 and h4 were different than h1 and h2. Before the first disruption, the primary path went through $\langle S12, S13\rangle$ and the backup went through $\langle S12, S16\rangle$, both on ICMP request and ICMP reply. After the disruption, when $\langle S12, S9\rangle$ link was removed, both paths change, the primary path began to go through $\langle S12, S16\rangle$ and the backup began to go through $\langle S12, S13\rangle$.

After the $\langle S12, S13\rangle$ link removal, there was not any relevant disturbance on this communication because the affected path was the backup one. Since the backup path ceased to exist, switches that belong to the primary path had to be reconfigured, that reconfiguration caused the disturbance observed around second 60.027 from the plot of Figure 5.3 in the following four ICMPs.

Like the previous communication, when the $\langle S12, S16\rangle$ link was removed, this one also stopped and the packets were dropped instantaneously.

After the $\langle S12, S9\rangle$ link was re-added, the traffic between h1 and h4 took around 545 ms to stabilize as it is possible to see illustrated on the plot from Figure 5.3.

The same experience was made with the pair of hosts h2 ↔ h3 and h3 ↔ h4. Those pairs obtained similar results and their respective charts are available on Appendix B-Mesh topology charts.

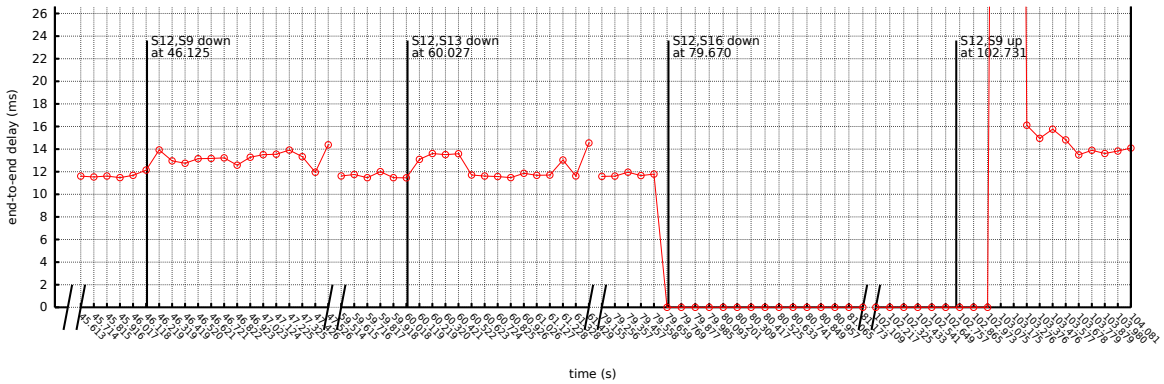


Figure 5.3: End-to-end delay for h1 and h4 communication when the different disruptions were made to the topology. The values off the chart after second 102.731 are 184 ms and 193 ms end-to-end delay.

5.1.3 IMPLEMENTATION 2.1 - H1 ↔ H2

After the communication between h1 and h2 started, the chosen path for ICMP request was $\langle S7, S8, S12, S13, S10\rangle$ and for ICMP reply was $\langle S10, S13, S12, S11, S7\rangle$. When the $\langle S12, S9\rangle$ link was removed, the communication did not suffer any significant packet delay nor packet loss. Since the primary path was not disrupted, the controller did not change it and only reconfigured a new backup path for S12 switch in a way that instead of going through S9, if the $\langle S12, S13\rangle$ link fails, the traffic goes directly to S16. On Figure 5.4 is represented a plot where are the disturbances marked that were wittingly made on the topology and the end-to-end delay for every packet sent on this communication.

As planned, when the $\langle S12, S13\rangle$ link was removed, the traffic was redirected to the backup switch S16. Although there was some visible end-to-end delay disturbance, not as different the one seen when $\langle S12, S9\rangle$ was removed, the end-to-end delay rose a little due the addition of a link in the backup path in comparison to primary path.

Once there was only one available path on this communication, after the removal of $\langle S12, S16\rangle$ link, the traffic stopped and the packets were dropped.

When the $\langle S12, S9\rangle$ link was added, the end-to-end delay increased drastically to around 360 ms gradually dropping to 270 ms, 170 ms, 70 ms and 12 ms until the communication stabilized. The reason for this to happen is implementation-defined behavior.

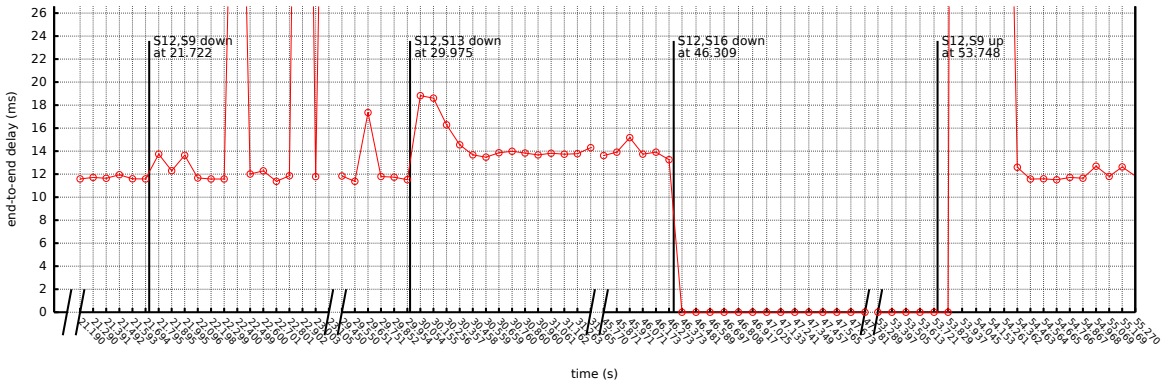


Figure 5.4: End-to-end delay for h1 and h2 communication when the different disruptions were made to the topology. The values off the chart after a re addition of $\langle S12, S9\rangle$ link are respectively: 362 ms, 274 ms, 173 ms and 69 ms.

After removing the $\langle S12, S16\rangle$ link, belonging to the last known path between h1 and h2, the controller removed the rules matching the destination h2 from all the switches belonging to that path. A similar behavior happened for rules matching the destination host 1 (from the reply's path). Once the S7 switch stopped having that rule, it started to contact the controller every time it received a packet from h1 with destination h2. As explained in the previous Section 4.1.4, when Open vSwitch contacts the controller it saves the packet received in a buffer. After the addition of $\langle S12, S9\rangle$ link, the controller knows a path between h1 and h2 and, when it receives a new unmatched packet, it configures the rules for that new path between those two hosts reconfiguring every switch belonging to that path. The rule sent from the controller to S7 switch has the buffer identification number previously sent

from the switch to the controller when the unmatched packet arrived. Since the controller took around 320 ms (160 ms for request and 160 ms for reply) to find the new path and the ICMPs were sent every 100 ms, the controller received at least 3 unmatched packets while reconfiguring the switches. After that reconfiguration, the controller started to reply for those unmatched ICMPs to the respective switches, until S7 and S10 received the matching rule.

The same experience was performed on pairs h1 ↔ h4, h2 ↔ h3 and h3 ↔ h4. Since the results obtained were the similar to the plot from Figure 5.4, they are present on Appendix B-Mesh topology charts.

5.2 SINGLE POINT OF FAILURE TOPOLOGY

Since the first implementation does not calculate a backup path in case the topology does not have a physical disjoint path of principal's path, this topology, represented on Figure 5.5 was intentionally created to have a single point of failure, the $\langle S9, S11 \rangle$ link. The purpose of this topology is to observe the recovery time when a link of the primary's path is removed as well the number of packet loss (if any) between the first and second implementation.

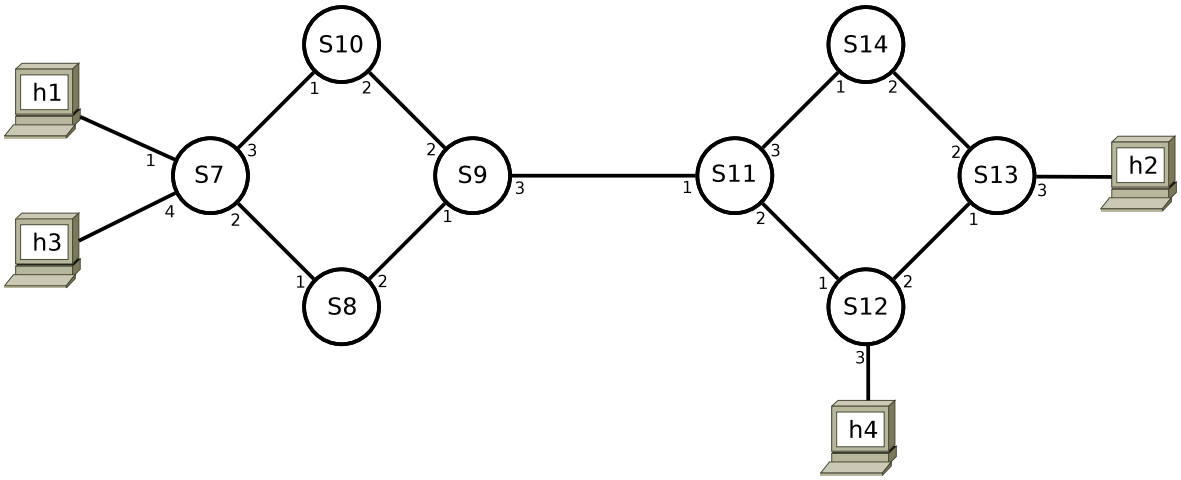


Figure 5.5: A single point of failure topology with some pair of hosts having multiple paths while the remaining have a single point-of-failure ($\langle S9, S11 \rangle$ link) on their communications.

Similar to the previous test, it was sent ICMP echo requests every 100 ms, from hosts h1 to h2, h1 to h3, h1 to h4, h2 to h3, h2 to h4 and h3 to h4. After all communications had already stabilized, the $\langle S12, S13 \rangle$ link was removed. This will provide data to analyze traffic that reaches a "dead-end". On this testbed will be all traffic from h1 and h3 with h2 as its destination. Later, this link was re-added. Afterwards it was removed the $\langle S14, S13 \rangle$ link to force all the connections, that had this link in their primary path, to be reconfigured by the controller. Finally, the $\langle S11, S12 \rangle$ link was removed to notice if the connection between h2 and h4 suffer any delay or disruption, since that link belong the backup path on both implementations for these hosts.

5.2.1 IMPLEMENTATION 1 - H1 $\leftrightarrow$ H2

As seen on Table 5.1, the primary path between host h1 and h2 was $\langle S7, S8, S9, S11, S12, S13\rangle$, the communication was interrupted, losing 2 packets, when the $\langle S12, S13\rangle$ link was removed. The controller was notified and reconfigured a new path for the hosts h1 and h2 in approximately 100 ms, accordingly with the plot on Figure 5.6. The communication continued like before after the reconfiguration for the new path.

There was not a significant change in the communication's delay when the $\langle S12, S13\rangle$ link was re-added. The minimal disturbances observed were due the reconfiguration from the controller to the switches for the new path found.

After the $\langle S12, S13\rangle$ link failure, the controller change the primary path for the ICMP reply from $\langle S13, S12, S11, S9, S8, S7\rangle$ to $\langle S13, S14, S11, S9, S10, S7\rangle$. Because of that change, the communication between hosts h2 and h1 lost 1 packet when the $\langle S14, S13\rangle$ link was removed. Like the previous disruption, the primary path was interrupted and the controller tries to find out if there is a new path between those 2 hosts, reconfiguring the switches for the new path.

The addition of $\langle S14, S13\rangle$ link had the same results observed as on $\langle S12, S13\rangle$ link.

Like the previous disruptions, the $\langle S11, S12\rangle$ link removal caused a packet loss. This was unfortunate because the link belonged to the primary path. Finally, the addition of $\langle S11, S12\rangle$ link only made the controller to calculate if there existed a new path different the one used. Since the controller found a different path it reconfigured the switches causing an insignificant variation in the communication's end-to-end delay.

Event	**Source**	**Destination**	**Primary Path**	**Backup Path**
Beginning of communication	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.
	h2	h1	$\langle S13, S12, S11, S9, S8, S7\rangle$	N.A.
link $\langle S12, S13\rangle$ down	h1	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	N.A.
	h2	h1	$\langle S13, S14, S11, S9, S10, S7\rangle$	N.A.
link $\langle S12, S13\rangle$ up	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.
link $\langle S13, S14\rangle$ down	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.
	h2	h1	$\langle S13, S12, S11, S9, S10, S7\rangle$	N.A.
link $\langle S13, S14\rangle$ up	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.
link $\langle S11, S12\rangle$ down	h1	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	N.A.
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.
link $\langle S11, S12\rangle$ up	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.

Table 5.1: Chosen paths by implementation 1 after each perturbation made in the topology of the communications between host h1 and host h2.

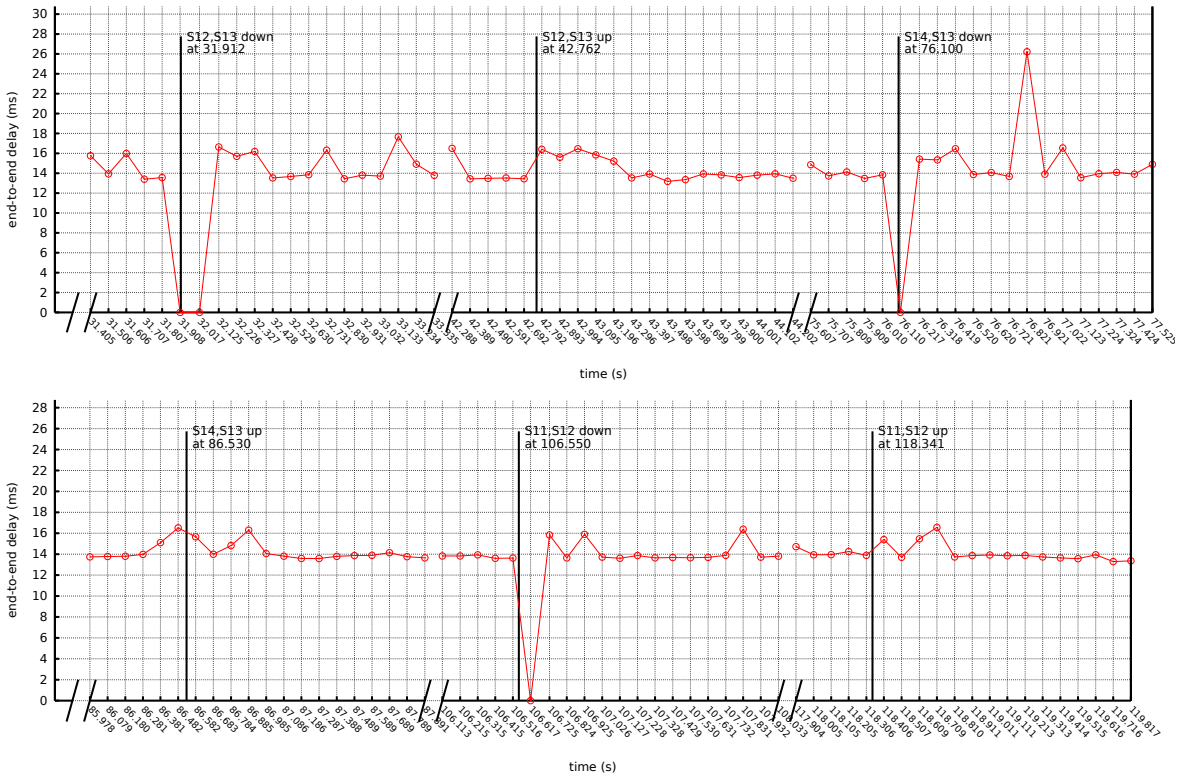


Figure 5.6: End-to-end delay for h1 and h2 communication when the different disruptions were made to the topology.

5.2.2 IMPLEMENTATION 1 - H1 ↔ H3

This pair of hosts was only to have a control group of hosts. Since the path that connect those hosts was not disturbed, as it can be seen on Figure 9 from Appendix B-Single point of failure topology charts, after every link addition and removal, the disturbances observed should be associated to the machine where all the topologies were emulated.

5.2.3 IMPLEMENTATION 1 - H1 ↔ H4

As seen in Figure 5.5, the path that connects host h1 and host h4, only has a primary path without a fully disjoint backup one. On the first, second, third and fourth disruption, it is possible to visualize on Figure 5.7 this path was not affected since those disruptions did not belong directly to the primary path. Only when the $\langle S11, S12 \rangle$ link was removed, 3 packets were lost. Like it was explained before, it was caused because of the first implementation's approach by needing a full disjoint path.

Upon reception on the controller of the notification stating that the $\langle S11, S12 \rangle$ link went down, it recalculates a new path for the pairs of hosts affected and sends the new path to the respective switches. That new path is described on Table 5.2. In this specific case, was sent a new rule for the switches belonging that new path, $\langle S7, S8, S9, S11, S14, S13, S12 \rangle$. It is possible to see the end-to-end

delay increased by 4 ms when the new path started to be used as seen on Figure 5.7 after second 106.787.

Event	Source	Destination	Primary Path	Backup Path
Beginning of communication	h1	h4	$\langle S7, S8, S9, S11, S12\rangle$	N.A.
	h4	h1	$\langle S12, S11, S9, S8, S7\rangle$	N.A.
link $\langle S12, S13\rangle$ down	h1	h4	$\langle S7, S8, S9, S11, S12\rangle$	N.A.
	h4	h1	$\langle S12, S11, S9, S10, S7\rangle$	N.A.
link $\langle S12, S13\rangle$ up	h1	h4	$\langle S7, S8, S9, S11, S12\rangle$	N.A.
	h4	h1	$\langle S12, S11, S9, S8, S7\rangle$	N.A.
link $\langle S13, S14\rangle$ down	h1	h4	$\langle S7, S8, S9, S11, S12\rangle$	N.A.
	h4	h1	$\langle S12, S11, S9, S10, S7\rangle$	N.A.
link $\langle S13, S14\rangle$ up	h1	h4	$\langle S7, S8, S9, S11, S12\rangle$	N.A.
	h4	h1	$\langle S12, S11, S9, S8, S7\rangle$	N.A.
link $\langle S11, S12\rangle$ down	h1	h4	$\langle S7, S8, S9, S11, S14, S13, S12\rangle$	N.A.
	h4	h1	$\langle S12, S13, S14, S11, S9, S8, S7\rangle$	N.A.
link $\langle S11, S12\rangle$ up	h1	h4	$\langle S7, S8, S9, S11, S12\rangle$	N.A.
	h4	h1	$\langle S12, S11, S9, S8, S7\rangle$	N.A.

Table 5.2: Chosen paths by implementation 1 after each perturbation made in the topology of the communications between host h1 and host h4.

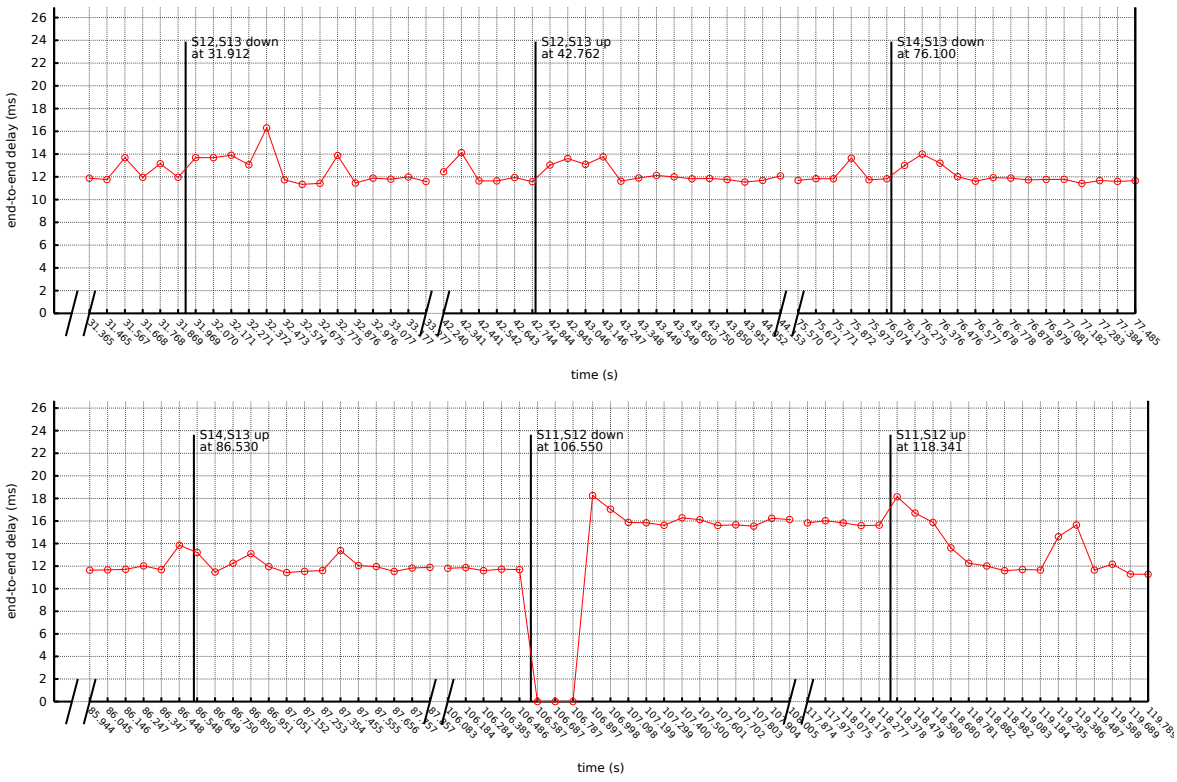


Figure 5.7: End-to-end delay for h1 and h4 communication when the different disruptions were made to the topology.

After the $\langle S11, S12\rangle$ link was re-added, the controller recalculates a new path for this pair of hosts. Since the new path is different from the previous one, the controller re-configures again the affected

switches. The new path found, $\langle S7, S8, S9, S11, S12\rangle$, made the end-to-end delay to become shorter, approaching the original's 10 ms end-to-end delay.

A similar behavior was observed on hosts h3 $\leftrightarrow$ h4 as they both use the same path from this pair of hosts. The results obtained are illustrated on Figure 10 from Appendix B-Single point of failure topology charts.

5.2.4 IMPLEMENTATION 1 - H2 $\leftrightarrow$ H3

The chosen path from the controller for hosts h2 and h3 was $\langle S7, S8, S9, S11, S14, S13\rangle$ as it can be seen on Table 5.3. When the first and second disruption on $\langle S12, S13\rangle$ link were caused, there was not a change nor packets lost between those hosts. Only when the $\langle S14, S13\rangle$ link, belonging to the primary path, was removed there was one packet lost. Only after receiving a new path from the controller that the communication started to stabilize.

When $\langle S14, S13\rangle$ link was re-added, the controller found other path between h2 and h3 different from the previous on. As seen on Figure 5.8, that reconfiguration caused a small variation in the communication's end-to-end delay.

Like the previous link removals, when $\langle S11, S12\rangle$ link was removed this caused a packet loss in this communication and only stabilized when the switches were reconfigured by the controller. The re-addition of $\langle S11, S12\rangle$ link also made the controller finding a different path from the one being used, causing a small disturbance in the communication's end-to-end delay.

Event	Source	Destination	Primary Path	Backup Path
Beginning of communication	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.
	h3	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	N.A.
link $\langle S12, S13\rangle$ down	h2	h3	$\langle S13, S14, S11, S9, S10, S7\rangle$	N.A.
	h3	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	N.A.
link $\langle S12, S13\rangle$ up	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.
link $\langle S13, S14\rangle$ down	h2	h3	$\langle S13, S12, S11, S9, S10, S7\rangle$	N.A.
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.
link $\langle S13, S14\rangle$ up	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.
link $\langle S11, S12\rangle$ down	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.
	h3	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	N.A.
link $\langle S11, S12\rangle$ up	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	N.A.
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	N.A.

Table 5.3: Chosen paths by implementation 1 after each perturbation made in the topology of the communications between host h2 and host h3.

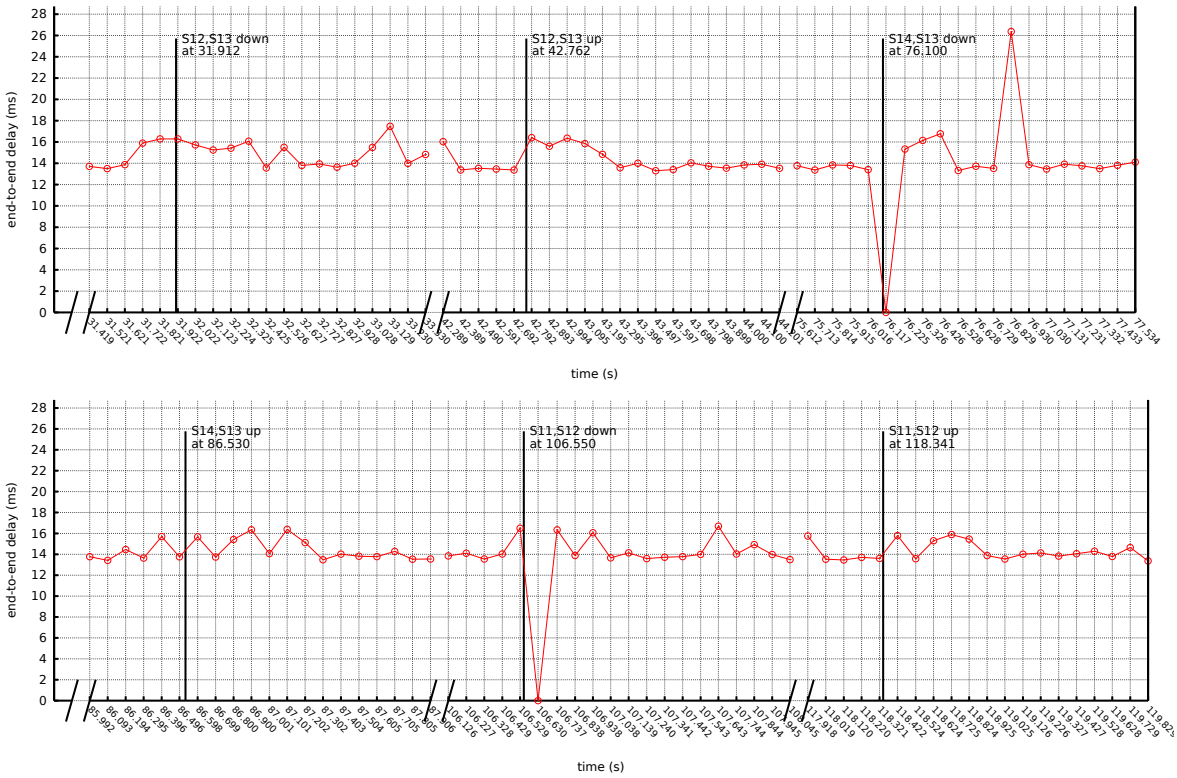


Figure 5.8: End-to-end delay for h2 and h3 communication with the disruptions.

5.2.5 IMPLEMENTATION 1 - H2 ↔ H4

Fortunately, for the communication between h2 and h4 there are two full disjoint paths, $\langle S13, S12\rangle$ and $\langle S13, S14, S11, S12\rangle$. When the controller received the first packet of that connection, it configured the shortest path as the primary one and the other path as backup as seen on Table 5.4. Unlike the other communications it is possible to see, as illustrated on Figure 5.9, that this communication did not lost any packet when the $\langle S12, S13\rangle$ link was removed. Instead the packets from this communication were directly redirected to the backup path.

The most interesting part on this test might be when the $\langle S12, S13\rangle$ link was re-added. Despite the emergence of a shorter path than the currently used, this implementation did not change it to the primary path. Instead, the controller reconfigured it as the backup path in case one of the links from the path $\langle S13, S14, S11, S12\rangle$ failed.

When the $\langle S14, S13\rangle$ link intentionally failed, the communication started to go by the backup path $\langle S13, S12\rangle$, decreasing the end-to-end delay of the communication.

Since the last three disruptions only occurred in the backup path and not on the primary one, there was not a major disturbance in this communication's end-to-end delay, after second 76.2 from the plot illustrated on Figure 5.9.

Event	Source	Destination	Primary Path	Backup Path
Beginning of communication	h2	h4	⟨S13, S12⟩	⟨S13, S14, S11, S12⟩
	h4	h2	⟨S12, S13⟩	⟨S12, S11, S14, S13⟩
link ⟨S12, S13⟩ down	h2	h4	⟨S13, S14, S11, S12⟩	N.A.
	h4	h2	⟨S12, S11, S14, S13⟩	N.A.
link ⟨S12, S13⟩ up	h2	h4	⟨S13, S14, S11, S12⟩	⟨S13, S12⟩
	h4	h2	⟨S12, S11, S14, S13⟩	⟨S12, S13⟩
link ⟨S13, S14⟩ down	h2	h4	⟨S13, S12⟩	N.A.
	h4	h2	⟨S12, S13⟩	N.A.
link ⟨S13, S14⟩ up	h2	h4	⟨S13, S12⟩	⟨S13, S14, S11, S12⟩
	h4	h2	⟨S12, S13⟩	⟨S12, S11, S14, S13⟩
link ⟨S11, S12⟩ down	h2	h4	⟨S13, S12⟩	N.A.
	h4	h2	⟨S12, S13⟩	N.A.
link ⟨S11, S12⟩ up	h2	h4	⟨S13, S12⟩	⟨S13, S14, S11, S12⟩
	h4	h2	⟨S12, S13⟩	⟨S12, S11, S14, S13⟩

Table 5.4: Chosen paths by implementation 1 after each perturbation made in the topology of the communications between host h2 and host h4.

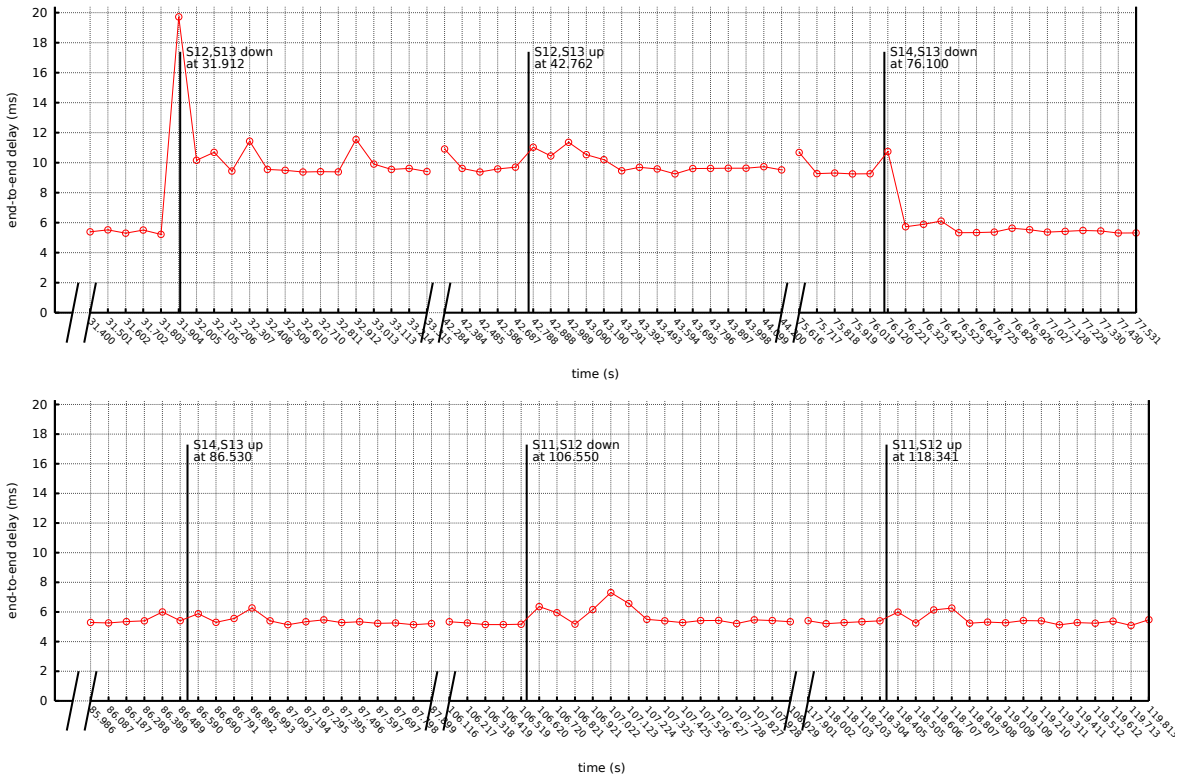


Figure 5.9: End-to-end delay for h2 and h4 communication with the disruptions.

5.2.6 IMPLEMENTATION 2.1 - H1 ↔ H2

While on the first implementation this communication did not had a backup path, this second implementation produced two backup paths at the beginning of the communication as it can be seen

on Table 5.5. On ICMP echo request, the primary chosen path was $\langle S7, S8, S9, S11, S12, S13\rangle$ and on ICMP echo reply was $\langle S13, S14, S11, S9, S8, S7\rangle$. Also, the controller determine the backup path the other possible paths, for example, in case $\langle S7, S8\rangle$ link or $\langle S8, S9\rangle$ link fails, the backup path will be $\langle S7, S10, S9, S11, S12, S13\rangle$. A similar thing will happen if the $\langle S12, S13\rangle$ link is disrupted, the traffic goes instantaneously through $\langle S7, S8, S9, S11, S14, S13\rangle$.

To prove it, $\langle S12, S13\rangle$ link was intentionally removed and even though the traffic from h1 reaches a dead-end, by reaching S12 switch, not a single packet was lost as seen on Figure 5.10. This was possible due the knowledge of the controller regarding the topology. In order to reach h2, S12 was programmed to send the traffic back to the port that it came, port #1.

Like explained in the previous section 4, and unlike implementation 1, the switches did not reprogram themselves when the traffic reaches a dead-end. Only when new rules are received from the controller that the traffic goes from S11 directly to S14. While those new rules are not received, the traffic goes from S11 to S12, coming back to S11 and then going to S14 causing the delay observed in the five following packets, around second 20.386 from the plot illustrated on Figure 5.10.

When $\langle S12, S13\rangle$ link was re-added, the controller changed the primary path, passing the traffic again through $\langle S12, S13\rangle$ instead of going through S14.

Since the ICMP echo reply goes through $\langle S13, S14\rangle$ link, when it was removed caused a variation in the end-to-end delay but nothing major. When the link was re-added, the controller change the ICMP echo reply's path to its original one.

The removal of $\langle S11, S12\rangle$ link did not cause the same disturbance as seen when $\langle S12, S13\rangle$ link was removed since, on this case, the traffic did not reach a dead-end, going directly to S14. Only when it was re-added, that was possible to see a disturbance due the reconfiguration of S11 to start sending the traffic again to S12.

Event	**Source**	**Destination**	**Primary Path**	**# of Backup Paths**
Beginning of communication	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	2
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	2
link $\langle S12, S13\rangle$ down	h1	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	1
	h2	h1	$\langle S13, S14, S11, S9, S10, S7\rangle$	1
link $\langle S12, S13\rangle$ up	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	2
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	2
link $\langle S13, S14\rangle$ down	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	1
	h2	h1	$\langle S13, S12, S11, S9, S10, S7\rangle$	1
link $\langle S13, S14\rangle$ up	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	2
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	2
link $\langle S11, S12\rangle$ down	h1	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	1
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	1
link $\langle S11, S12\rangle$ up	h1	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	2
	h2	h1	$\langle S13, S14, S11, S9, S8, S7\rangle$	2

Table 5.5: Chosen paths by implementation 2.1 after each perturbation made in the topology of the communications and the number of backup paths that existed between host h1 and host h2.

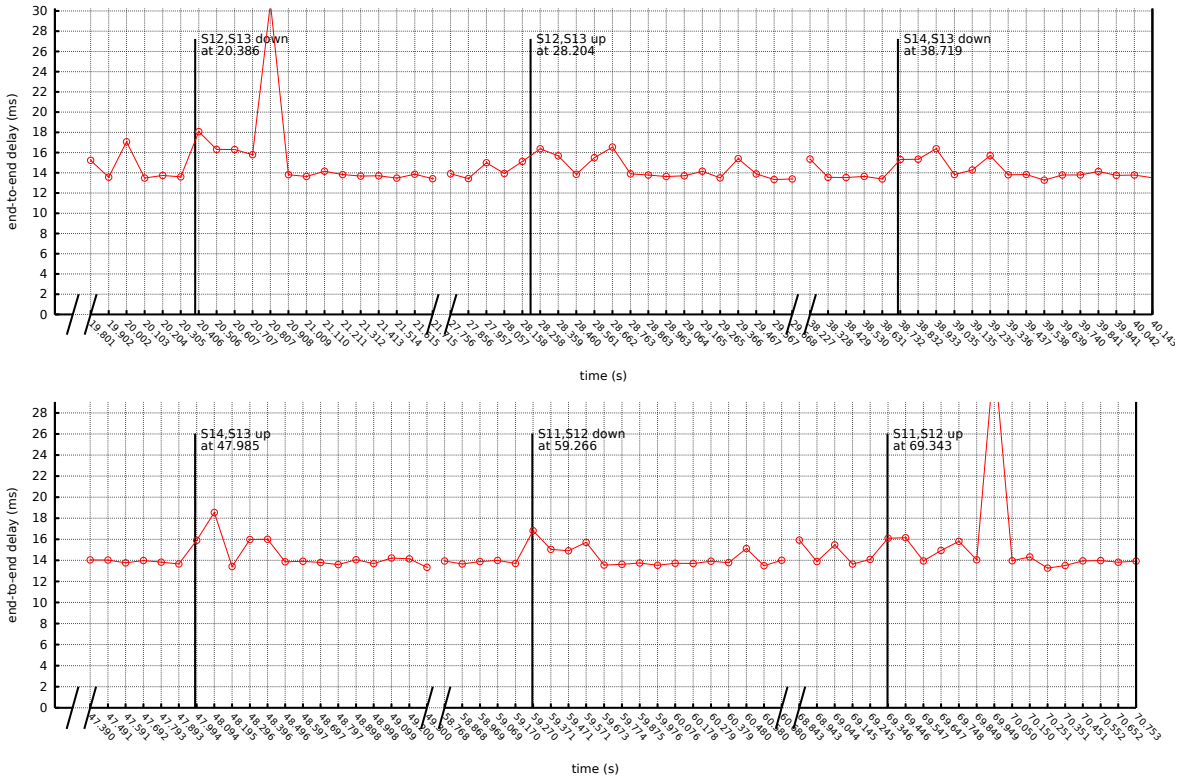


Figure 5.10: End-to-end delay for h1 and h2 communication with the disruptions.

5.2.7 IMPLEMENTATION 2.1 - H1 ↔ H3

This pair of hosts was only to have a control group of hosts. Since the path that connect those hosts was not disturbed, as it can be seen on Figure 11 from Appendix B-Single point of failure topology charts, after every link addition and removal, the disturbances observed should be associated to the machine where all the topologies were simulated.

5.2.8 IMPLEMENTATION 2.1 - H1 ↔ H4

Like the first implementation, the first, second, third and fourth disruption did not cause any major variations in the end-to-end delay's communication as seen on Figure 5.11.

Without losing a single packet, only when the $\langle S11, S12 \rangle$ link was removed it is possible to see, accordingly with Table 5.6, the communication started to go through the backup path $\langle S7, S8, S9, S11, S14, S13, S12 \rangle$. When the $\langle S11, S12 \rangle$ link was re-added the communication returned to the original's path, also without loosing a packet.

A similar behavior was observed on hosts h3 ↔ h4 as they both use the same path from this pair of hosts. The results obtained are illustrated on Figure 12 from Appendix B-Single point of failure

topology charts.

Event	Source	Destination	Primary Path	# of Backup Paths
Beginning of communication	h1	h4	⟨S7, S8, S9, S11, S12⟩	2
	h4	h1	⟨S12, S11, S9, S8, S7⟩	2
link ⟨S12, S13⟩ down	h1	h4	⟨S7, S8, S9, S11, S12⟩	1
	h4	h1	⟨S12, S11, S9, S10, S7⟩	1
link ⟨S12, S13⟩ up	h1	h4	⟨S7, S8, S9, S11, S12⟩	2
	h4	h1	⟨S12, S11, S9, S8, S7⟩	2
link ⟨S13, S14⟩ down	h1	h4	⟨S7, S8, S9, S11, S12⟩	1
	h4	h1	⟨S12, S11, S9, S10, S7⟩	1
link ⟨S13, S14⟩ up	h1	h4	⟨S7, S8, S9, S11, S12⟩	2
	h4	h1	⟨S12, S11, S9, S8, S7⟩	2
link ⟨S11, S12⟩ down	h1	h4	⟨S7, S8, S9, S11, S14, S13, S12⟩	1
	h4	h1	⟨S12, S13, S14, S11, S9, S8, S7⟩	1
link ⟨S11, S12⟩ up	h1	h4	⟨S7, S8, S9, S11, S12⟩	2
	h4	h1	⟨S12, S11, S9, S8, S7⟩	2

Table 5.6: Chosen paths by implementation 2.1 after each perturbation made in the topology of the communications and the number of backup paths that existed between host h1 and host h4.

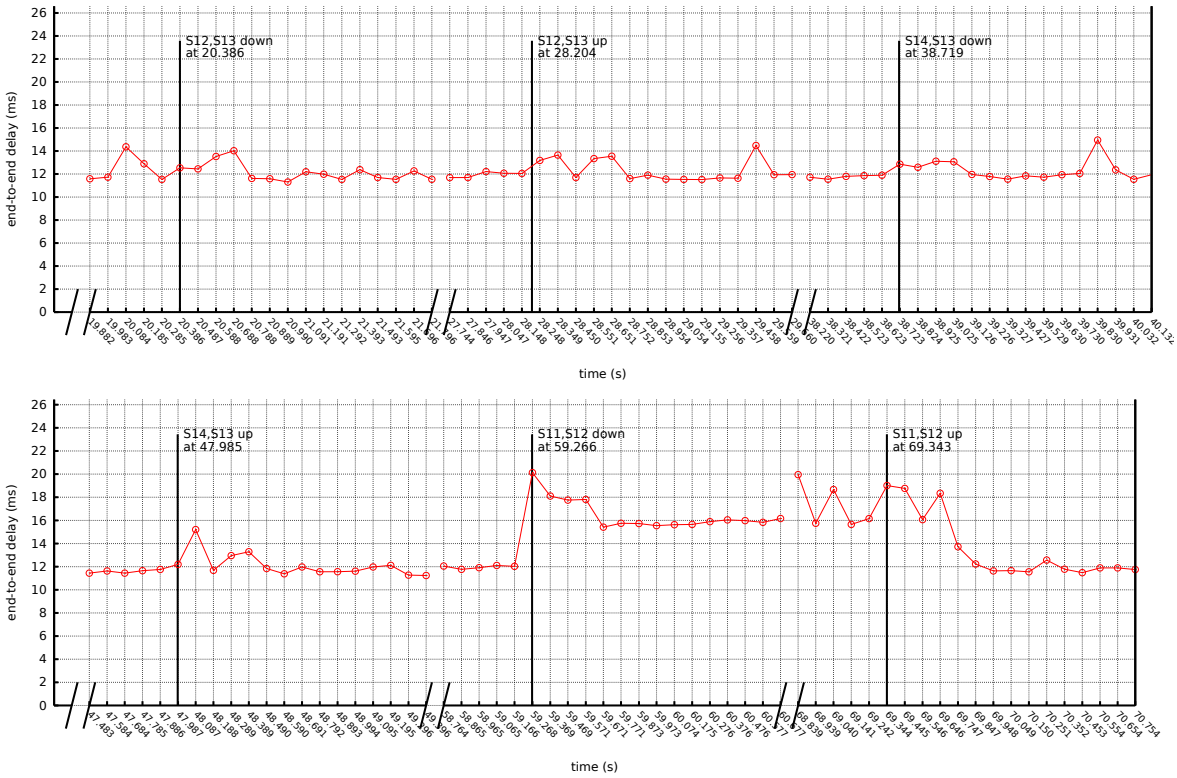


Figure 5.11: End-to-end delay for h1 and h4 communication with the disruptions.

5.2.9 IMPLEMENTATION 2.1 - H2 ↔ H3

Since this implementation had only rules' matches by MAC address destination, the chosen path for the ICMP echo reply was the same used for the ICMP echo request in the h1 to h2's connection as

seen on Table 5.5 and Table 5.7.

The first intentional disruption caused some disturbance in the communication because the ICMP echo replies reach a dead-end, like the ICMP echo requests from h1 to h2. As seen on Figure 5.12, only when the controller configured S11 the communication stabilized.

Event	Source	Destination	Primary Path	# of Backup Paths
Beginning of communication	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	2
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	2
link $\langle S12, S13\rangle$ down	h2	h3	$\langle S13, S14, S11, S9, S10, S7\rangle$	1
	h3	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	1
link $\langle S12, S13\rangle$ up	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	2
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	2
link $\langle S13, S14\rangle$ down	h2	h3	$\langle S13, S12, S11, S9, S10, S7\rangle$	1
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	1
link $\langle S13, S14\rangle$ up	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	2
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	2
link $\langle S11, S12\rangle$ down	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	1
	h3	h2	$\langle S7, S8, S9, S11, S14, S13\rangle$	1
link $\langle S11, S12\rangle$ up	h2	h3	$\langle S13, S14, S11, S9, S8, S7\rangle$	2
	h3	h2	$\langle S7, S8, S9, S11, S12, S13\rangle$	2

Table 5.7: Chosen paths by implementation 2.1 after each perturbation made in the topology of the communications and the number of backup paths that existed between host h2 and host h3.

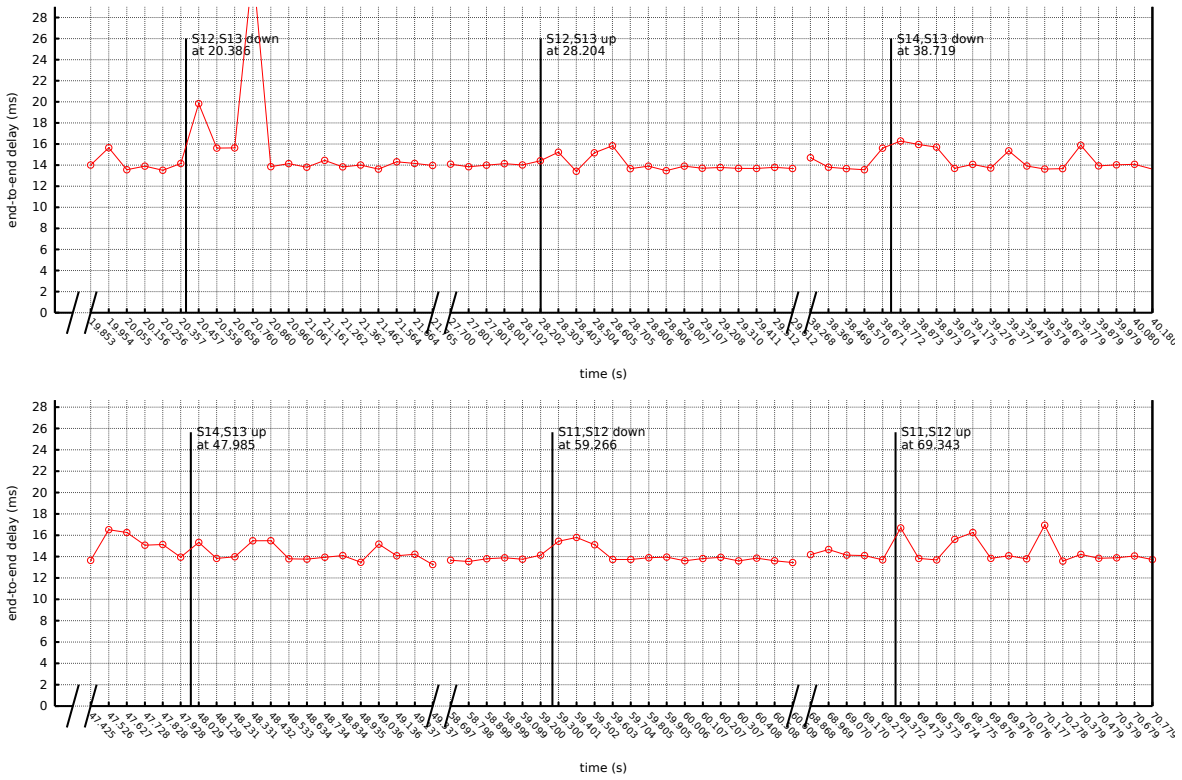


Figure 5.12: End-to-end delay for h2 and h3 communication with the disruptions.

Since the rules used only match the ingress port and MAC address destination, it is possible to see a small improvement in implementation 2.1 in contrast to implementation 2. When the controller

received a notification, stating the ⟨S12, S13⟩ link was removed, it only had to send one rule to S11 with the MAC address destination of h2 instead of two rules, one for the traffic coming from h1 with destination h2 and the other one from h3 to h2.

Because the following disruptions did not occur on a dead-end link (both on request and reply), the communication's end-to-end delay did not suffer a major variation.

5.2.10 IMPLEMENTATION 2.1 - H2 ↔ H4

This particular communication had similarities between both implementations 1 and 2.1. Since the communication had two different disjoint paths, it is possible to compare both implementations.

When the ⟨S12, S13⟩ link was removed, like observed in implementation 1, on this one there was not a packet loss as illustrated on Figure 5.13. The differences occurred when the ⟨S12, S13⟩ link was added. On implementation 1, the path did not change because there was always only two paths, the principal and backup. On this second implementation there is always one primary path and one or more backup paths, on this specific case only one backup path was found as seen on Table 5.8. Once that link was re added, the controller found a new path with less links, making it as the primary one, decreasing the communication's end-to-end delay.

Regarding the following disruptions, since they did not belong to the primary path, they did not cause any disturbance in the communication.

Event	**Source**	**Destination**	**Primary Path**	**# of Backup Paths**
Beginning of communication	h2	h4	⟨S13, S12⟩	1
	h4	h2	⟨S12, S13⟩	1
link ⟨S12, S13⟩ down	h2	h4	⟨S13, S14, S11, S12⟩	0
	h4	h2	⟨S12, S11, S14, S13⟩	0
link ⟨S12, S13⟩ up	h2	h4	⟨S13, S12⟩	1
	h4	h2	⟨S12, S13⟩	1
link ⟨S13, S14⟩ down	h2	h4	⟨S13, S12⟩	0
	h4	h2	⟨S12, S13⟩	0
link ⟨S13, S14⟩ up	h2	h4	⟨S13, S12⟩	1
	h4	h2	⟨S12, S13⟩	1
link ⟨S11, S12⟩ down	h2	h4	⟨S13, S12⟩	0
	h4	h2	⟨S12, S13⟩	0
link ⟨S11, S12⟩ up	h2	h4	⟨S13, S12⟩	1
	h4	h2	⟨S12, S13⟩	1

Table 5.8: Chosen paths by implementation 2.1 after each perturbation made in the topology of the communications and the number of backup paths that existed between host h2 and host h4.

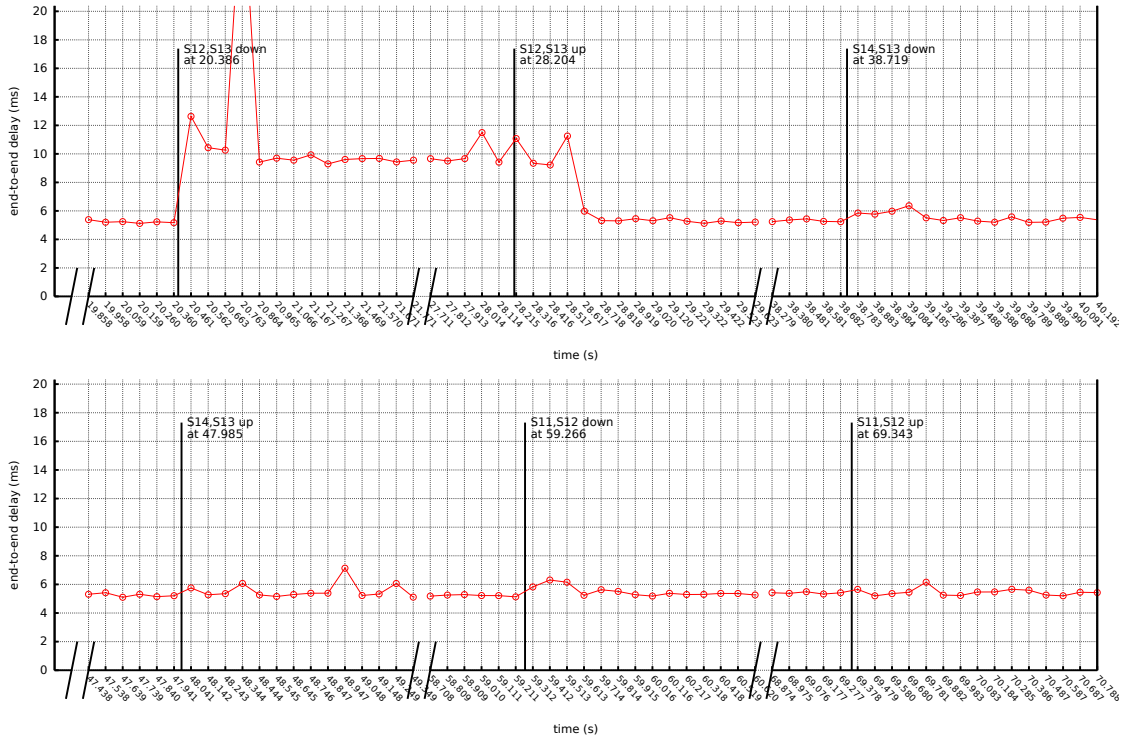


Figure 5.13: End-to-end delay for h2 and h4 communication with the disruptions.

5.3 20 HOSTS TOPOLOGY

Contrary the previous topologies, this topology, illustrated in Figure 5.14, was created to test the controller's response when the number of flows was increased. The number of rules presented in the topology when the number of flows was increased was also analyzed.

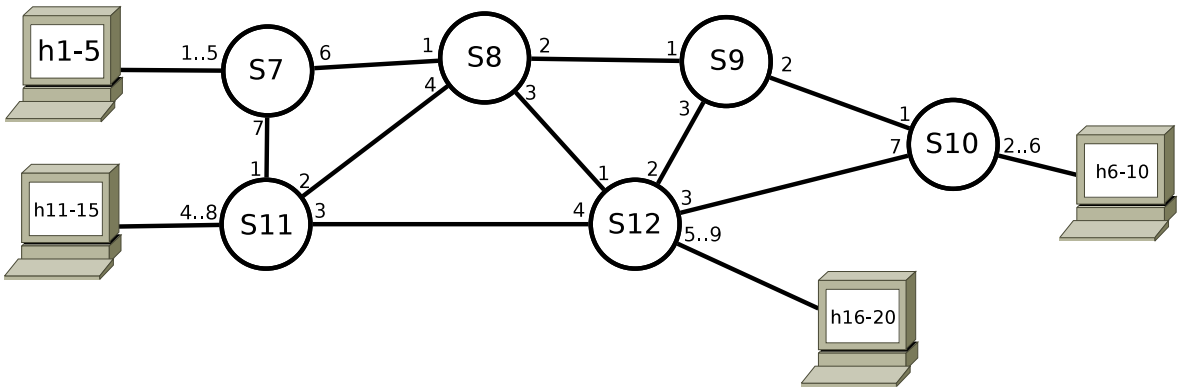


Figure 5.14: A 20 hosts topology.

First, ICMP echo request were sent from h1 to h6 through h10 and h16 through h20, having twenty different connections (ten for ICMP requests and ten for ICMP replies). The $\langle S8, S12 \rangle$ link was removed as soon the communications stabilized. Then, the time the controller took since the moment

it received the notification that the link was removed, until the last affected flow is reconfigured was measured. This procedure was repeated for hosts h2, h3, h4, h5, h11, h12, h13, h14 and finally h15 reaching a total of 200 flows.

5.3.1 CONTROLLER - TIME TO CALCULATE NEW PATHS

On every implementation the time it took to recalculate a new backup path increased when the number of affected flows increased. For 20 flows, the maximum time it took for implementation 1, 2 and 2.1 was respectively 1.65, 0.41 and 0.31 seconds and for 200 flows the maximum time it took for implementation 1, 2 and 2.1 was respectively 139, 17 and 5 seconds.

Taking these times in consideration, it can be concluded implementation 1 is not suitable for a typical layer 2 network, either by the time it took to calculate a new backup path either by the need for a full disjoint path of the primary path.

When observing those times for implementations 2 and 2.1, it is remarkable the performance gained by the absence of MAC address source matching. This absence contributed to an increased performance of around 70%. Although the implementation 2.1 took around 5 seconds to recalculate a new backup path for 200 flows, it is necessary to take into account, for a typical network with 200 flows, it is rare the occurrence of two failures on a 5 second period [82].

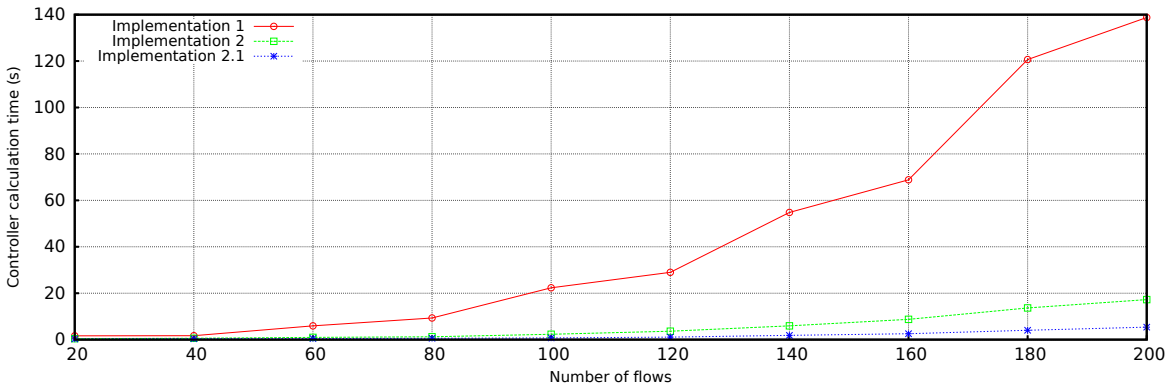


Figure 5.15: Plot with the amount of seconds the controller took to calculate new paths for the respective number of flows.

5.3.2 RULES - NUMBER OF RULES PRESENT IN THE TOPOLOGY.

While the time for calculating a backup rule is important, there is also need to take in consideration the amount of rules present on the topology when the number of flows increase. Although the amount of rules present for the implementation 1 and 2 were around 1700-1800, the lack of matching for MAC address source allowed the number of present rules to be decreased over 77%.

Also, 30 seconds (easily adjustable) after the communication between hosts stopped, for implementation 2 and 2.1 the number of rules present in the topology dropped to 0 while on the implementation 1 only dropped to 1243. The removal of those rules prevents a burst in the switches' memory when new connections appear and the old ones disappear.

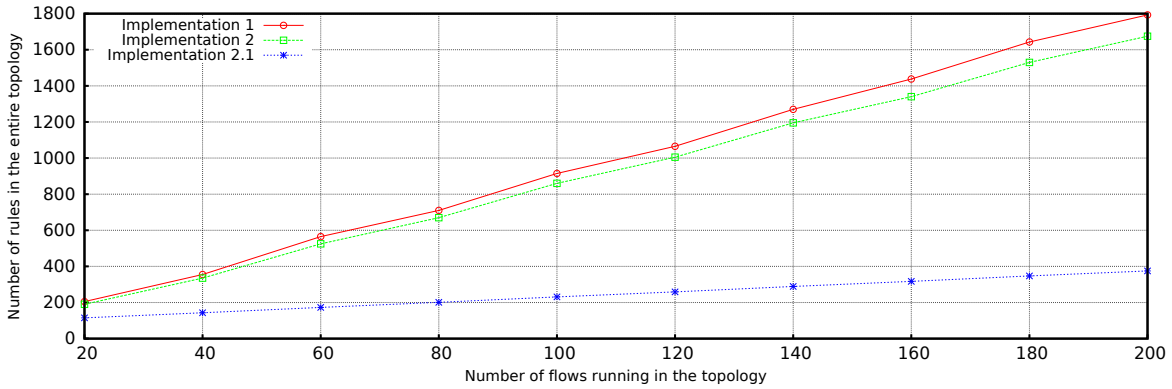


Figure 5.16: Plot with the total number of rules present in the topology for the respective number of flows.

5.4 RESULTS ANALYSIS

The different implementations produced for this dissertation were tested and further analyzed in this chapter. Since each one of them had a different approach they produced different results. First of all, implementation 1 produced the worst results, essentially because of the need of a full disjoint path from the primary one to create a backup path. In addition, as Figure 5.15 illustrates, the abrupt calculation's time when the number of flows increased is not good for a real environment. Regarding implementations 2 and 2.1, after the disruptions performed on a single and non-single point of failure topologies, it was possible to obtain a critical communication with 0 packet loss upon a topology's link disruption. On implementation 2.1, with the removal of MAC address source from the matching rules, a substantial improvement occurred on the controller side as well as a decreasing number of rules on the switches' side. Nevertheless, the approach followed on implementation 2 and 2.1 of finding multiple backup paths did not bring major differences in the amount of time taken by the controller comparing to implementation 1.

In essence, OpenDaylight controller is still a young project and considering the studied references, it is not currently being used on an enterprise network. However, on a data center environment, where there are fewer machines connected, it is possible to reach the desired 100% up time with OpenFlow. As a conclusion to the work done throughout the dissertation, on a larger network it would be essential to have parallel controllers in order to smooth the amount of data needed to be managed when topology changes occur. On the other hand, to overcome the large amount of controllers it is relevant to have a

more proactive network. This would level the amount of possible proactive scenarios and the amount of controllers a topology could have.

CHAPTER 6

Conclusion

6.1 work overview

Considering the SDN paradigm, there is still a lack of development of solutions that provide enough redundancy and explore all its functionalities. The future of the enterprises data centers is to adopt virtualized networks in general due to the saving of resources. It is relevant, then, that these enterprises make use of improved solutions that guarantee 100% up time. However, the protocols in existence such as STP, MSTP, RSTP are not optimized for SDNs. In addition, in spite of TRILL and SPB protocols being emerging, only one of them has guaranteed proof of concept, the SPB. SDNs are evolving quickly and in order to create solutions that provide the required redundancy levels for data centers and network infrastructures, there is already hardware and software to help implement such solutions. Therefore, the main objective of this dissertation was to implement a robust enough solution that uses SDN technologies, for critical communications.

Taking SDN paradigm in consideration, it was firstly chosen the NOX controller because it is the most basic and considered as the first controller to ever being made for OpenFlow. However, since NOX is considered deprecated it was dismissed. The next step was choosing another controller, POX. With this controller a small tutorial was followed in order to create rules for the switches and was possible to have a STP implementation using OpenFlow. The Mininet emulator was used for creating the network topologies, as recommended by the tutorials. Later on, it was found that the Mininet includes a virtual forwarding device, the Open vSwitch. Unfortunately, it was a very old version of it. This motivated the search for another forwarding device and the *ofsoftswitch13* was the next choice. In order to adapt to *ofsoftswitch13*, it was followed the same tutorial used on POX. While testing this forwarding device it was found that it did not recognized when a link was re added after its removal. This was essential to future tests when links would be removed and added multiple times. This implementation issue led to a new search for another forwarding device. The Open vSwitch had a

non stable version, 2.1.0, which was still on development. This version was chosen because in spite of not being stable yet, it showed to be promising. Later, this version transited from non-stable to stable and no issues were added to the implementations already performed.

A deeper study on the OpenFlow protocol directed this dissertation to the fact that there was still the need for a more stable and professional controller in order to fulfill the main objective of implementing a fail safe solution for critical communications. Some knowledge was taken about OpenDaylight and it was after considered as the ideal controller since it was a open-source project, has a large number of contributors and some essentially core functionalities needed for the implementations developed.

Following, the first implementation could be developed since all the requirements for the devices were met. The first solution consisted on finding a primary and a backup path between the two machines that were trying to communicate. In order to have a backup path this solution had the requirement of needing a full disjoint path between the two switches that directly connected the machines. This solution did not include, hence, innovation as it could not guarantee that all topologies had two completely disjoint paths for each pair of machines.

As a result, the second implementation took place. This implementation had a totally different approach than the one followed on the first implementation: it was not necessary to have disjunctive paths. This happened due to the fact that it was possible to have one primary path and various sub backup paths, where these sub backup paths could even overlap among them. This solution, however, implied that an overload of installed rules and memory issues could happen on each forwarding device. Each device had to know which backup and primary paths that the packets could take in order to reach the destiny. This approach, due to memory issues that could be involved, was not completely followed as the first approach.

The testing stage was the next phase of the dissertation. The developed tests were created specifically to observe the existing failures on both implementations and in order to verify if any optimizations could occur. The first test was designed in order to observe the behavior of both implementations towards a re addition of a link, whenever a connection between two hosts had been totally broken. The second test was designed to verify the degree of the complications which were present on the first implementation, since its approach considered a topology where some of the machine pairs did not have disjoint paths between them. The last test was created to observe the controller scalability and the number of installed rules in all forwarding devices on the topology. While performing this test, it was verified that the second implementation could still be optimized if the matching of the source MAC addresses was removed, maintaining only the matching between the destiny MAC address and source port. This led to version 2.1. The same tests were applied to the new implementation and it showed impressive results on the scalability test because both the number of installed rules as well as the time for the controller to calculate new backup paths, for a fair number of connections, immensely dropped.

With these tests it was possible to verify that not a single packet was lost and, within 1 second,

the packets were redirected to their destination whenever link in the topology was removed. Besides, it could be concluded that for 200 up connections at the same time, a domestic computer can calculate new backup paths within 5 seconds.

6.2 FUTURE WORK

Although the overall objectives of this dissertation were achieved, there is still some relevant work that can be implemented in outside the data centers environments. Some of the most relevant work and new functionalities that can be implemented in the future are:

- At a more technical level, this work was implemented between OpenDaylight releases. It would be interesting to have the implementations developed for the next release of OpenDaylight, including the implementation dependencies;
- The OpenDaylight supports multiple controllers for redundancy purposes [83]. Although the OpenFlow have procedures to deal in case the communication between the switch and the controller breaks, it would be interesting to see how the topology behaved after the switches lose communication with the controller;
- As described in this work, one of the dependencies used was the Dijkstra algorithm to find the shortest path between two nodes in the topology. This plugin has an interface to implement the path between two nodes that have available bandwidth in the hole topology and it can be interesting to have traffic engineering on a fast fail-safe topology;
- Although some tests were performed to determine the scalability of the implementations, they were not totally conclusive for larger networks with over 100 switches. Therefore, it would be interesting to perform these tests on future deployments;
- The solutions developed are OpenFlow based but the same principles can be implemented on Engine Control Unit (ECU). The current car manufacturers are already moving the internal protocols to Ethernet but there are still problems dealing with redundancy on those environments [84]. Thus, this work can give continuity to a fast fail-safe topology for micro controllers environment. An extra work has to be carried in order to verify SDN applicability in automotive scenarios.

References

[1] R. Perlman, "An algorithm for distributed computation of a spanningtree in an extended LAN", *ACM SIGCOMM Computer Communication Review*, pp. 44–53, 1985. [Online]. Available: http://dl.acm.org/citation.cfm?id=319004.

[2] Juniper. (2014). QFabric System - Virtualized Data Center Foundation - Juniper Networks, [Online]. Available: http://www.juniper.net/us/en/products-services/switching/qfabric-system/.

[3] Brocade, *Brocade VCS Fabric Technical Architecture*, 2012. [Online]. Available: http://www.brocade.com/downloads/documents/technical%5C_briefs/vcs-technical-architecture-tb.pdf.

[4] Cisco. (2014). Cisco FabricPath - Cisco, [Online]. Available: http://www.cisco.com/c/en/us/solutions/data-center-virtualization/fabricpath/index.html.

[5] A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management", *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 3, p. 41, Oct. 2005, ISSN: 01464833. DOI: 10.1145/1096536.1096541. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1096536.1096541.

[6] Carnegie Mellon University. (2008). Clean Slate Architectures for Network Management, [Online]. Available: http://www.cs.cmu.edu/afs/cs/Web/People/4D/ (visited on 06/02/2014).

[7] M. Caesar, D. Caldwell, A. Shaikh, N. Feamster, J. Rexford, and J. van der Merwe, "Design and Implementation of a Routing Control Platform", in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05, Berkeley, CA, USA: USENIX Association, 2005, pp. 15–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251205.

[8] M. Casado, T. Garfinkel, and A. Akella, "SANE: A protection architecture for enterprise networks", *15th USENIX Security Symposium*, pp. 137–151, 2006. [Online]. Available: https://www.usenix.org/event/sec06/tech/full%5C_papers/casado/casado%5C_html.

[9] M. Casado, M. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise", *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 1–12, Aug. 2007, ISSN: 0146-4833. DOI: 10.1145/1282427.1282382. [Online]. Available: http://dl.acm.org/citation.cfm?id=1282427.1282382.

[10] S. Shenker, M. Casado, T. Koponen, and N. McKeown, "A Gentle Introduction to Software Defined Networks", Tech. Rep., 2009, p. 39. [Online]. Available: http://tce.technion.ac.il/files/2012/06/Scott-shenker.pdf.

[11] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks", *ACM SIGCOMM Computer Communication Review*,

vol. 38, no. 3, pp. 105–110, Jul. 2008, ISSN: 0146-4833. DOI: 10.1145/1384609.1384625. [Online]. Available: http://dl.acm.org/citation.cfm?id=1384609.1384625.

[12] N. McKeown, T. Anderson, L. Peterson, J. Rexford, S. Shenker, H. Balakrishnan, G. Parulkar, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008, ISSN: 01464833. DOI: 10.1145/1355734.1355746. [Online]. Available: http://portal.acm.org/citation.cfm?id=1355746%20http://dl.acm.org/citation.cfm?id=1355746.

[13] H. J. Chao and B. Liu, *High Performance Switches and Routers*. 2006, pp. 1–613, ISBN: 0470053674. DOI: 10.1002/9780470113950.

[14] P. Gonçalves, A. Martins, D. Corujo, and R. Aguiar, "A fail-safe SDN bridging platform for Cloud Networks", in *International Telecommunications Network Strategy and Planning Symposium*, Funchal, 2014, p. 6.

[15] Cisco. (2014). Per VLAN Spanning Tree, [Online]. Available: http://www.cisco.com/en/US/tech/tk389/tk621/tk846/tsd_technology_support_sub-protocol_home.html (visited on 06/02/2014).

[16] C. J. Sher Decusatis, A. Carranza, and C. M. Decusatis, "Communication within clouds: open standards and proprietary protocols for data center networking", *IEEE Communications Magazine*, vol. 50, no. 9, pp. 26–33, Sep. 2012, ISSN: 0163-6804. DOI: 10.1109/MCOM.2012.6295708. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6295708.

[17] P. Ashwood-Smith, "The Great Debate : TRILL Versus 802.1aq (SBP) What is this all about?", in *NANOG 50*, 2010, p. 68. [Online]. Available: http://www.nanog.org/meetings/nanog50/presentations/Monday/NANOG50.Talk63.NANOG50%5C_TRILL-SPB-Debate-Roisman.pdf.

[18] R. Perlman, D. E. 3rd, D. Dutt, S. Gai, and A. Ghanwani, *Routing Bridges (RBridges): Base Protocol Specification*, RFC 6325 (Proposed Standard), Updated by RFCs 6327, 6439, 7172, 7177, 7179, 7180, Internet Engineering Task Force, Jul. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6325.txt.

[19] D. Eastlake, A. Banerjee, D. Dutt, R. Perlman, and A. Ghanwani, *Transparent Interconnection of Lots of Links (TRILL) Use of IS-IS*, RFC 6326 (Proposed Standard), Obsoleted by RFC 7176, Internet Engineering Task Force, Jul. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6326.txt.

[20] D. E. 3rd, R. Perlman, A. Ghanwani, D. Dutt, and V. Manral, *Routing Bridges (RBridges): Adjacency*, RFC 6327 (Proposed Standard), Obsoleted by RFC 7177, updated by RFC 7180, Internet Engineering Task Force, Jul. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6327.txt.

[21] J. Carlson and D. E. 3rd, *PPP Transparent Interconnection of Lots of Links (TRILL) Protocol Control Protocol*, RFC 6361 (Proposed Standard), Internet Engineering Task Force, Aug. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6361.txt.

[22] R. Perlman, D. Eastlake, Y. Li, A. Banerjee, and F. Hu, *Routing Bridges (RBridges): Appointed Forwarders*, RFC 6439 (Proposed Standard), Updated by RFC 7180, Internet Engineering Task Force, Nov. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6439.txt.

[23] D. Oran, *OSI IS-IS Intra-domain Routing Protocol*, RFC 1142 (Historic), Obsoleted by RFC 7142, Internet Engineering Task Force, Feb. 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1142.txt.

[24] R. Perlman and D. Eastlake, "Introduction to TRILL", *The Internet Protocol Journal*, vol. 14, no. 3, pp. 2–20, Jun. 2011, ISSN: 1471-4159. DOI: 10.1111/jnc.12566. [Online]. Available:

http://www.cisco.com/web/about/ac123/ac147/archived%5C_issues/ipj%5C_14-3/ipj%5C_14-3.pdf.

[25] Ronald van der Pol, *TRILL and IEEE 802.1aq Overview*, 2012. [Online]. Available: http://www.rvdp.org/publications/TRILL-SPB.pdf.

[26] IEEE. (2014). 802.1Qbp - Equal Cost Multiple Paths, [Online]. Available: http://www.ieee802.org/1/pages/802.1bp.html (visited on 06/05/2014).

[27] Avaya, "Compare and Contrast SPB and TRILL", pp. 1–10, 2011. [Online]. Available: http://www.avaya.com/uk/resource/assets/whitepapers/SPB-TRILL%5C_Compare%5C_Contrast-DN4634.pdf.

[28] ——, "Sochi 2014 Olympic Winter Games", p. 4, 2013. [Online]. Available: http://www.avaya.com/ru/resource/assets/casestudies/DN7191EN-Sochi.pdf.

[29] The OpenStack Foundation. (2014). OpenStack Open Source Cloud Computing Software, [Online]. Available: https://www.openstack.org/.

[30] The OpenNebula Project. (2014). OpenNebula | Flexible Enterprise Cloud Made Simple, [Online]. Available: http://opennebula.org/.

[31] Apache Software Foundation. (2014). Apache CloudStack: Open Source Cloud Computing, [Online]. Available: http://cloudstack.apache.org/.

[32] Eucalyptus Systems, Inc. (2014). Eucalyptus | Open Source Private Cloud Software, [Online]. Available: https://www.eucalyptus.com/.

[33] The Open Compute Project. (2014). Open Compute Project, [Online]. Available: http://www.opencompute.org/.

[34] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey", p. 49, Jun. 2014. arXiv: 1406.0440. [Online]. Available: http://arxiv.org/abs/1406.0440.

[35] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN", *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, Apr. 2014, ISSN: 01464833. DOI: 10.1145/2602204.2602219. [Online]. Available: http://dl.acm.org/citation.cfm?id=2602219.

[36] Open Networking Foundation, *OpenFlow Switch Specification 1.0.0*, 2009. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf.

[37] D. Erickson, G. Gibb, B. Heller, D. Underhill, J. Naous, G. Appenzeller, G. Parulkar, N. McKeown, M. Rosenblum, M. Lam, S. Kumar, V. Alaria, P. Monclus, F. Bonomi, J. Tourrilhes, P. Yalagandula, S. Banerjee, C. Clark, and R. McGeer, "A demonstration of virtual machine mobility in an OpenFlow network", in *ACM SIGCOMM*, 2008, p. 4, ISBN: 9781605581750. [Online]. Available: http://klamath.stanford.edu/~nickm/papers/p513-ericksonA.pdf.

[38] Open Networking Foundation, *OpenFlow Switch Specification 1.4.0*, 2009. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[39] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an OpenFlow architecture", in *Proceedings of the 23rd International Teletraffic Congress*, International Teletraffic Congress, Sep. 2011, pp. 1–7, ISBN: 978-0-9836283-0-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2043468.2043470.

[40] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks", *Proceedings of the 2Nd USENIX Conference on Hot*

Topics in Management of Internet, Cloud, and Enterprise Networks and Services, p. 10, Apr. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228283.2228297.

[41] S. Kim, J.-M. Kang, S.-s. Seo, and J. W.-K. Hong, "A cognitive model-based approach for autonomic fault management in OpenFlow networks", *International Journal of Network Management*, vol. 23, no. 6, pp. 383–401, Nov. 2013, ISSN: 10557148. DOI: 10.1002/nem.1839. [Online]. Available: http://doi.wiley.com/10.1002/nem.1839.

[42] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in OpenFlow networks", *2011 8th International Workshop on the Design of Reliable Communication Networks (DRCN)*, pp. 164–171, Oct. 2011. DOI: 10.1109/DRCN.2011.6076899. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6076899.

[43] A. Tanenbaum, *Modern Operating Systems (3rd Edition)*, 3rd ed. Prentice Hall, 2007, p. 1104, ISBN: 978-0136006633.

[44] Cisco. (2014). Cisco IOS Technologies - Cisco, [Online]. Available: http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-technologies/index.html.

[45] Juniper. (2014). Junos Network Operating System - Juniper Networks, [Online]. Available: http://www.juniper.net/us/en/products-services/nos/junos/.

[46] Z. Cai, A. Cox, and E. T. S. Ng, "Maestro: A System for Scalable OpenFlow Control", Tech. Rep., 2011, p. 10. [Online]. Available: http://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf.

[47] T. S. Eugene Ng (PI), Alan L. Cox (co-PI), Zheng Cai (code maintainer), Florin Dinu, Jie Zheng. (2014). maestro-platform - A scalable control platform writen in Java which supports OpenFlow switches, [Online]. Available: http://code.google.com/p/maestro-platform/.

[48] B. Pfaff and B. Davie, *The Open vSwitch Database Management Protocol*, RFC 7047 (Informational), Internet Engineering Task Force, Dec. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7047.txt.

[49] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, *Network Configuration Protocol (NETCONF)*, RFC 6241 (Proposed Standard), Internet Engineering Task Force, Jun. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6241.txt.

[50] J. Case, M. Fedor, M. Schoffstall, and J. Davin, *Simple Network Management Protocol (SNMP)*, RFC 1157 (Historic), Internet Engineering Task Force, May 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1157.txt.

[51] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed multi-domain SDN controllers", *arXiv preprint arXiv:1308.6138*, p. 8, 2013. arXiv: arXiv:1308.6138v2. [Online]. Available: http://arxiv.org/abs/1308.6138.

[52] ——, "DISCO: Distributed multi-domain SDN controllers", English, in *Network Operations and Management Symposium*, IEEE, 2014, pp. 1–4, ISBN: 9781479909131. DOI: 10.1109/NOMS.2014.6838330. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6838330.

[53] Project Floodlight. (2014). Floodlight OpenFlow Controller -Project Floodlight, [Online]. Available: http://www.projectfloodlight.org/floodlight/.

[54] Nicira. (2014). About NOX | NOXRepo, [Online]. Available: http://www.noxrepo.org/nox/about-nox/.

[55] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: a distributed control platform for large-scale production networks", pp. 1–6, Oct. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968.

[56] Open Networking Lab. (2014). ONOS, [Online]. Available: http://onlab.us/tools/onos.html.

[57] The OpenDaylight Project, Inc. (2014). OpenDaylight | A Linux Foundation Collaborative Project, [Online]. Available: http://www.opendaylight.org/.

[58] Nicira. (2014). About POX | NOXRepo, [Online]. Available: http://www.noxrepo.org/pox/about-pox/.

[59] Ryu. (2014). Ryu SDN Framework, [Online]. Available: http://osrg.github.io/ryu/.

[60] D. Erickson, "The beacon openflow controller", in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, New York, New York, USA: ACM Press, 2013, p. 13, ISBN: 9781450321785. DOI: 10.1145/2491185.2491189. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2491185.2491189.

[61] OpenDaylight. (2014). PacketCablePCMM:Main - Daylight Project, [Online]. Available: https://wiki.opendaylight.org/view/PacketCablePCMM:Main.

[62] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild", in *Proceedings of the 10th annual conference on Internet measurement - IMC '10*, New York, New York, USA: ACM Press, Nov. 2010, p. 267, ISBN: 9781450304832. DOI: 10.1145/1879141.1879175. [Online]. Available: http://dl.acm.org/citation.cfm?id=1879141.1879175.

[63] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple", in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM - SIGCOMM '13*, vol. 43, New York, New York, USA: ACM Press, Aug. 2013, pp. 87–98, ISBN: 9781450320566. DOI: 10.1145/2486001.2486030. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486001.2486030.

[64] Martín Casado. (2013). OpenStack and Network Virtualization | VMware Company Blog, [Online]. Available: http://blogs.vmware.com/vmware/2013/04/openstack-and-network-virtualization.html.

[65] Alan Weissberger. (2013). VMware's Network Virtualization Poses Huge Threat to Data Center Switch Fabric Vendors | The Viodi View, [Online]. Available: http://viodi.com/2013/05/06/vmwares-network-virtualization-poses-huge-threat-to-data-center-switch-fabric-vendors/.

[66] NetFPGA. (2014). NetFPGA - NetFPGA, [Online]. Available: http://netfpga.org/.

[67] NoviSwitch. (2014). NoviFlow - NoviSwitch, [Online]. Available: http://noviflow.com/products/noviswitch/.

[68] Hewlett-Packard. (2014). HP 8200 zl Switch Series - HP Networking | HP, [Online]. Available: http://h17007.www1.hp.com/us/en/networking/products/switches/HP_8200_zl_Switch_Series/.

[69] Brocade. (2014). Brocade MLX Series Overview, [Online]. Available: http://www.brocade.com/products/all/routers/product-details/netiron-mlx-series/index.page.

[70] CPqD. (2014). CPqD/ofsoftswitch13 · GitHub, [Online]. Available: https://github.com/CPqD/ofsoftswitch13.

[71] Open vSwitch. (2014). Open vSwitch, [Online]. Available: http://openvswitch.org/.

[72] Juniper Networks. (2014). Juniper/contrail-vrouter - GitHub, [Online]. Available: https://github.com/Juniper/contrail-vrouter.

[73] F. N. N. Farias, J. J. Salvatti, E. C. Cerqueira, and A. J. G. Abelem, "A proposal management of the legacy network environment using OpenFlow control plane", in *2012 IEEE Network Operations and Management Symposium*, IEEE, Ed., Ieee, Apr. 2012, pp. 1143–1150, ISBN: 978-

1-4673-0269-2. DOI: 10.1109/NOMS.2012.6212041. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6212041.

[74] Open vSwitch. (2014). Features | Open vSwitch, [Online]. Available: http://openvswitch.org/features/.

[75] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer.", in *8th ACM Workshop on Hot Topics inNetworks*, ACM, vol. VIII, New York City, 2009, p. 6. [Online]. Available: http://www.icsi.berkeley.edu/pubs/networking/extendingnetworking09.pdf.

[76] Open vSwitch. (2014). ovs-ofctl manpage, [Online]. Available: http://openvswitch.org/cgi-bin/ovsman.cgi?page=utilities/ovs-ofctl.8.

[77] OpenDaylight. (May 2014). OpenFlow Plugin:Overview Architecture, [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:Overview_Architecture (visited on 05/27/2014).

[78] ——, (2014). OpenFlow Plugin Project, [Online]. Available: https://git.opendaylight.org/gerrit/openflowplugin (visited on 06/02/2014).

[79] Open Networking Foundation, *OpenFlow Switch Specification 1.3.1*, 2009. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.

[80] OpenDaylight. (2014). OpenDaylight OpenFlow Plugin:Potential Helium Items, [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:Potential_Helium_Items (visited on 05/27/2014).

[81] ——, (2014). Dixon-Erickson Proposal:Host Tracker Plan, [Online]. Available: https://wiki.opendaylight.org/view/D-E_Proposal:Host_Tracker_Plan (visited on 05/27/2014).

[82] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers", *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, p. 350, Oct. 2011, ISSN: 01464833. DOI: 10.1145/2043164.2018477. [Online]. Available: http://dl.acm.org/citation.cfm?id=2043164.2018477.

[83] OpenDaylight. (2014). penDaylight SDN Controller Platform (OSCP):Main - Daylight Project, [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight_SDN_Controller_Platform_(OSCP):Main.

[84] M. Jochim, "General perspective on AVB 2 & Redundancy aspects relevant for backbone and control applications", IEEE, Tech. Rep., 2012, pp. 1–45. [Online]. Available: http://www.ieee802.org/1/files/public/docs2012/new-avb-jochim-redundancy-requirements-GM-perspective-AVB2-0312.pdf.

Appendix A

Sequence diagrams related to the implementations on OpenDaylight

In this appendix is presented the sequence diagrams for the OpenDaylight developed bundles in this work.

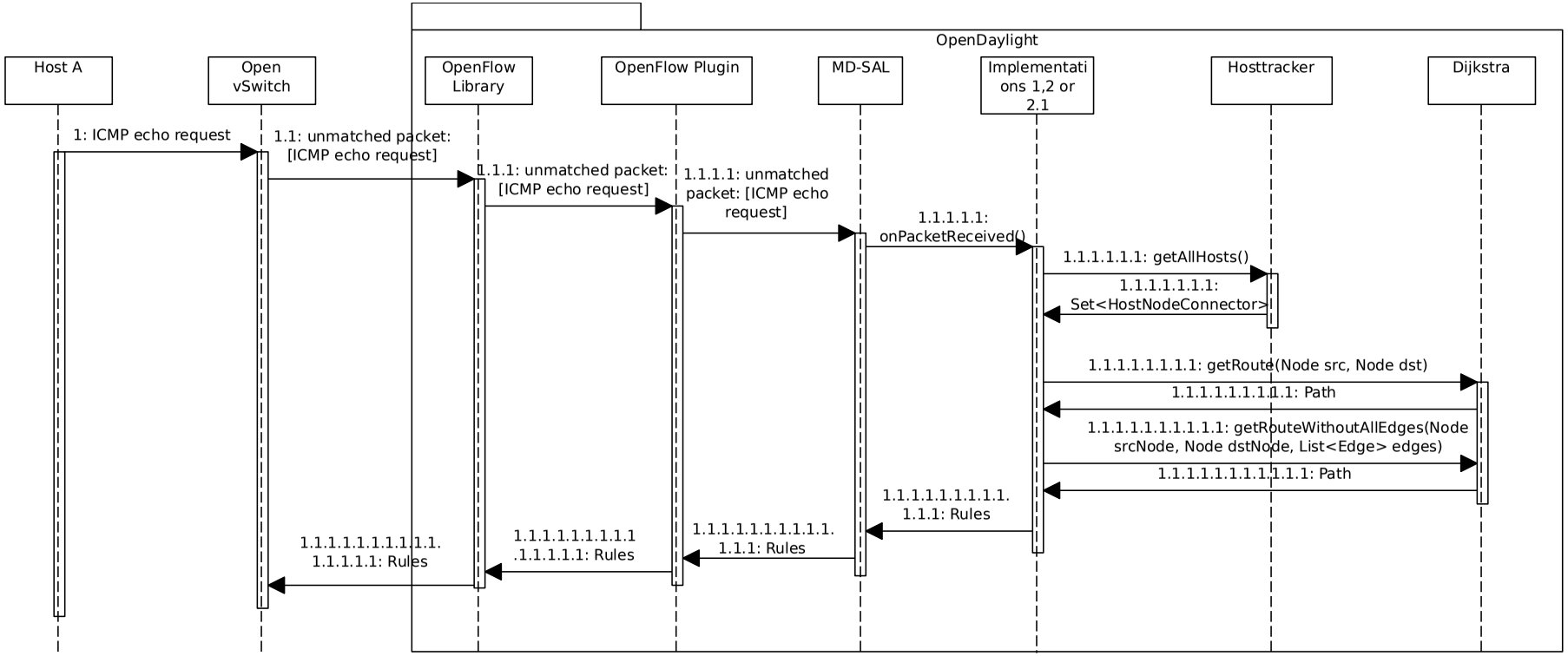


Figure 1: Sequence diagram for the OpenDaylight bundles when a unmatched packet reaches a node.

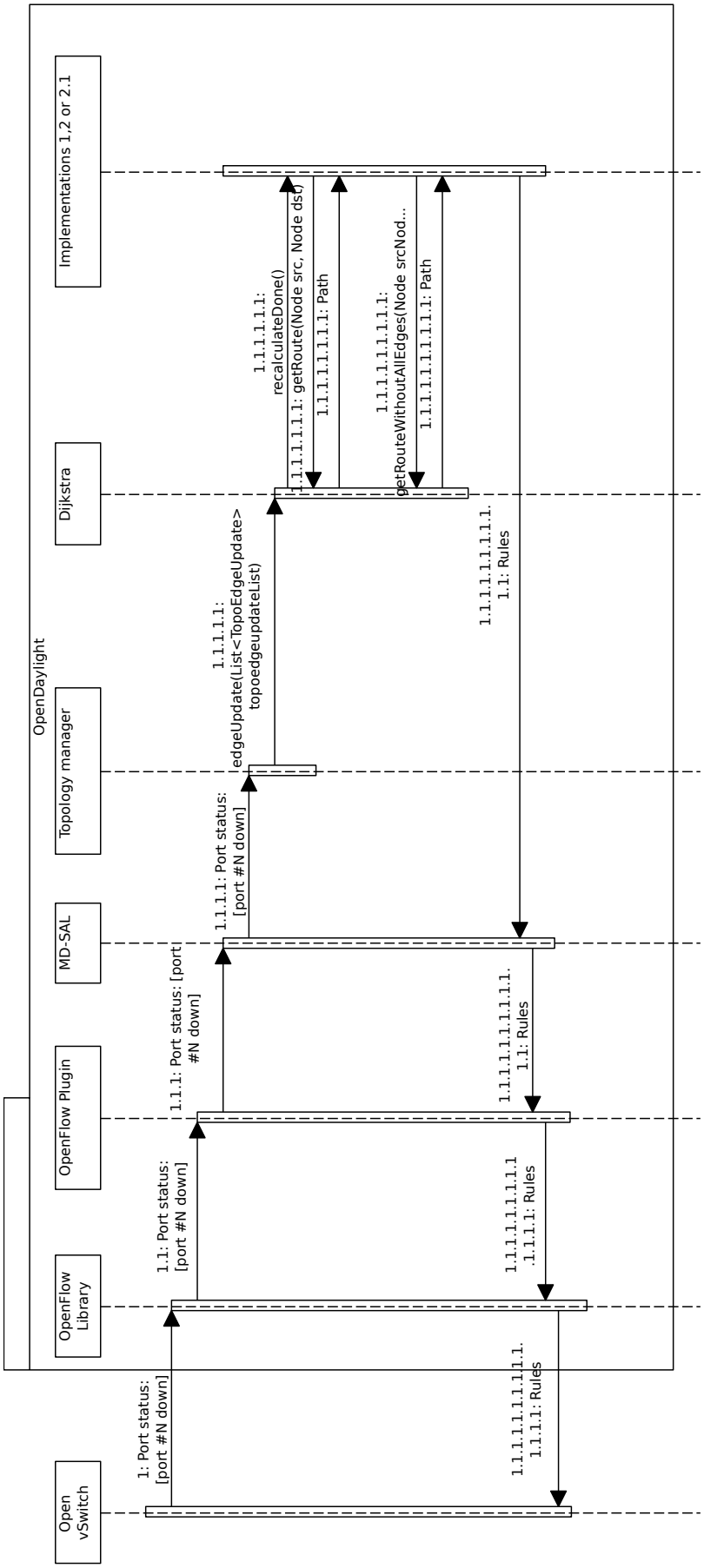


Figure 2: Sequence diagram for the OpenDaylight bundles when a link from the topology is removed.

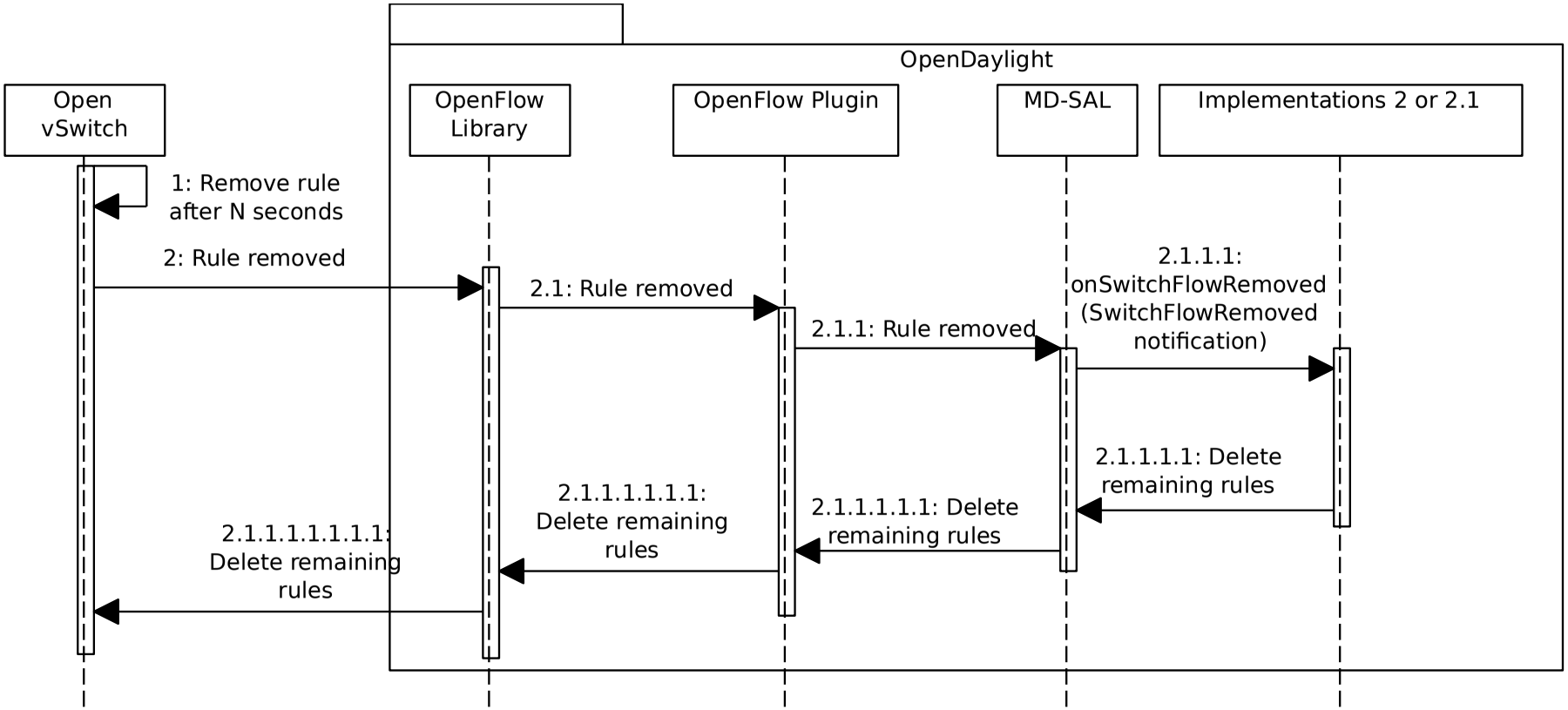


Figure 3: Sequence diagram for the OpenDaylight bundles when a link from the topology is removed.

Appendix B

Mesh topology charts

Implementation 1 - h2 ↔ h3

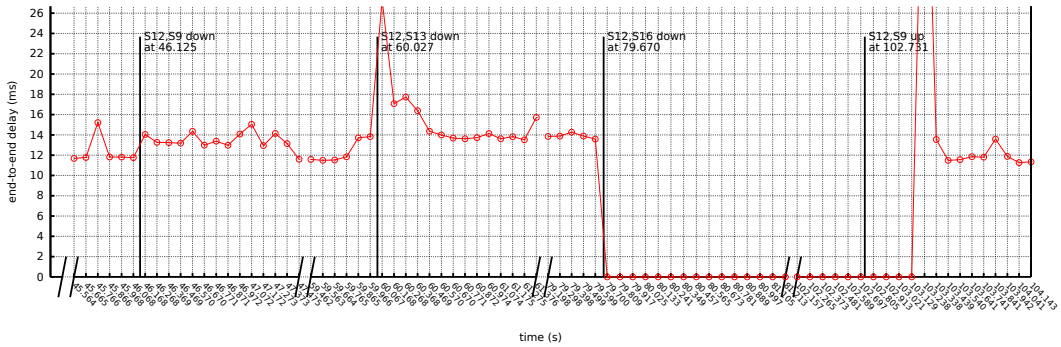


Figure 4: End-to-end delay for h2 and h3 communication with the disruptions.

Implementation 1 - h3 ↔ h4

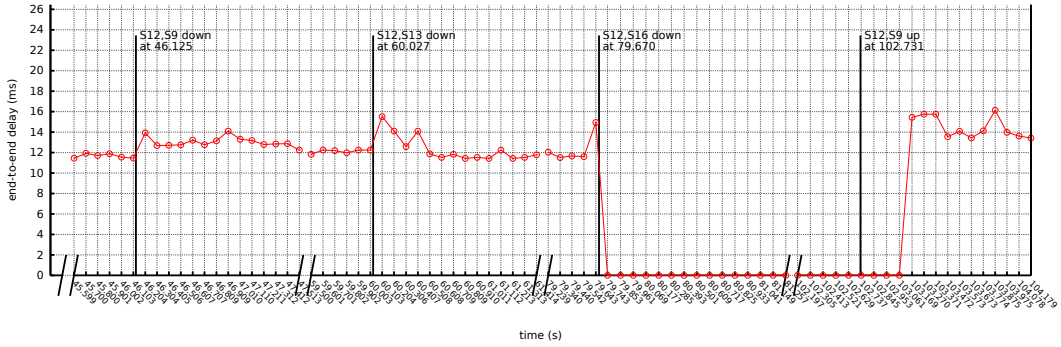


Figure 5: End-to-end delay for h3 and h4 communication with the disruptions.

IMPLEMENTATION 2.1 - H1 ↔ H4

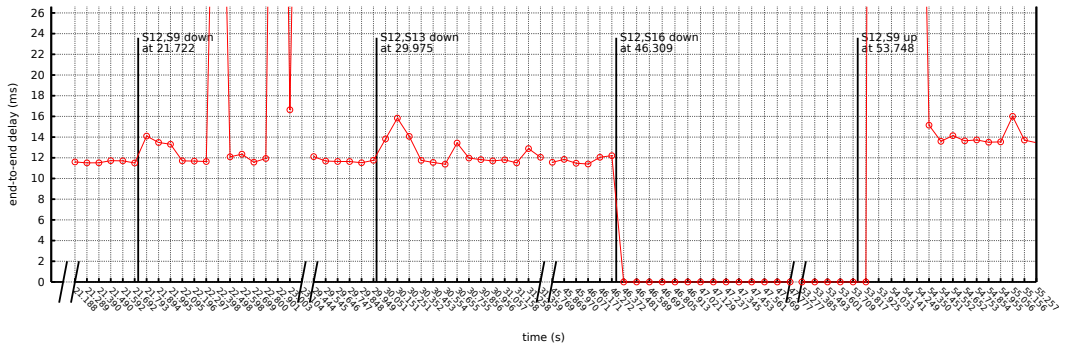


Figure 6: End-to-end delay for h1 and h4 communication with the disruptions.

IMPLEMENTATION 2.1 - H2 ↔ H3

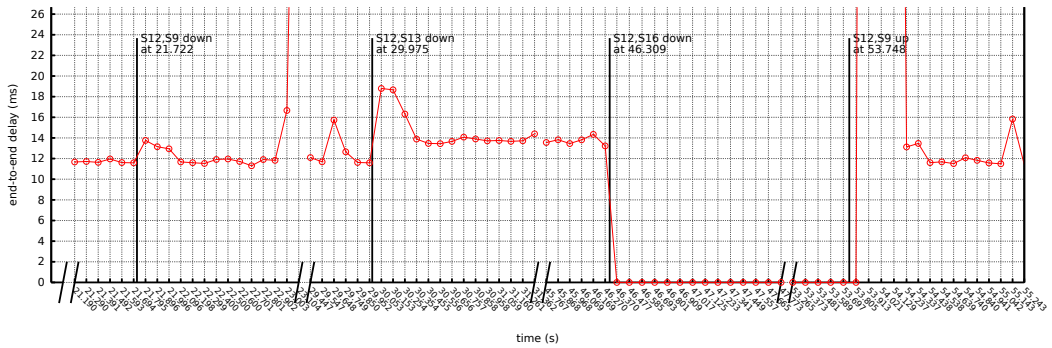


Figure 7: End-to-end delay for h2 and h3 communication with the disruptions.

IMPLEMENTATION 2.1 - H3 ↔ H4

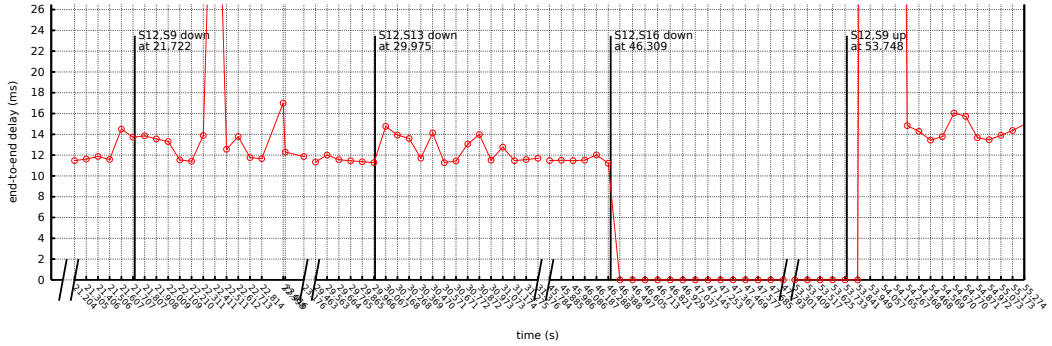


Figure 8: End-to-end delay for h3 and h4 communication with the disruptions.

SINGLE POINT OF FAILURE TOPOLOGY CHARTS

IMPLEMENTATION 1 - H1 ↔ H3

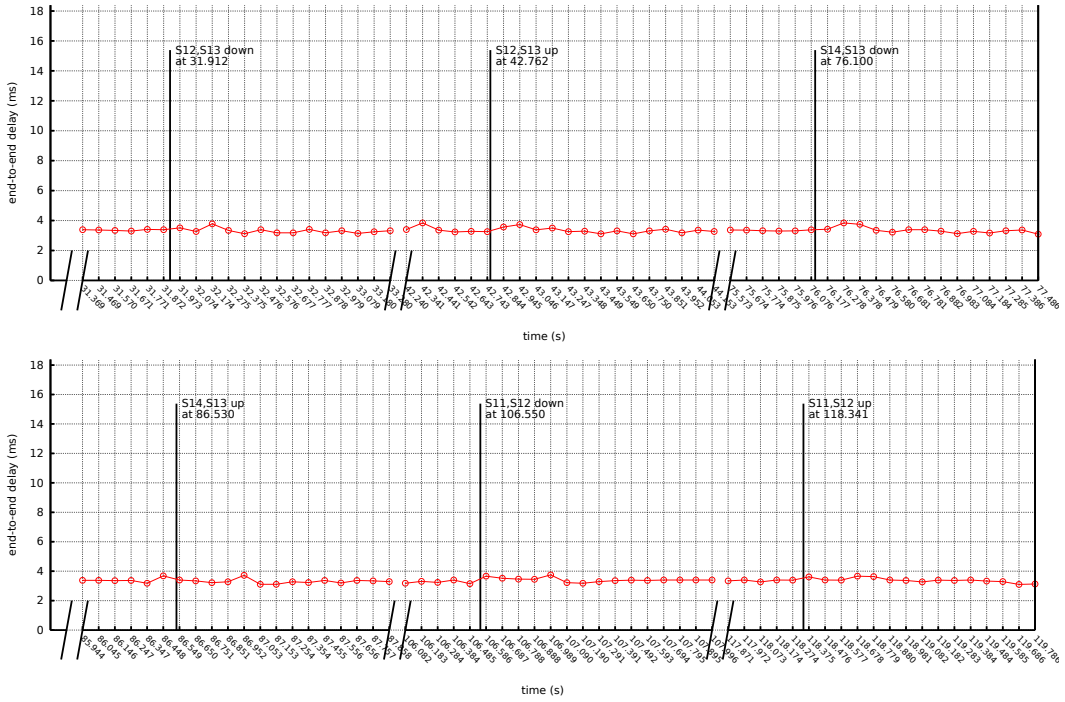


Figure 9: End-to-end delay for h1 and h3 communication when the different disruptions were made to the topology.

IMPLEMENTATION 1 - H3 ↔ H4

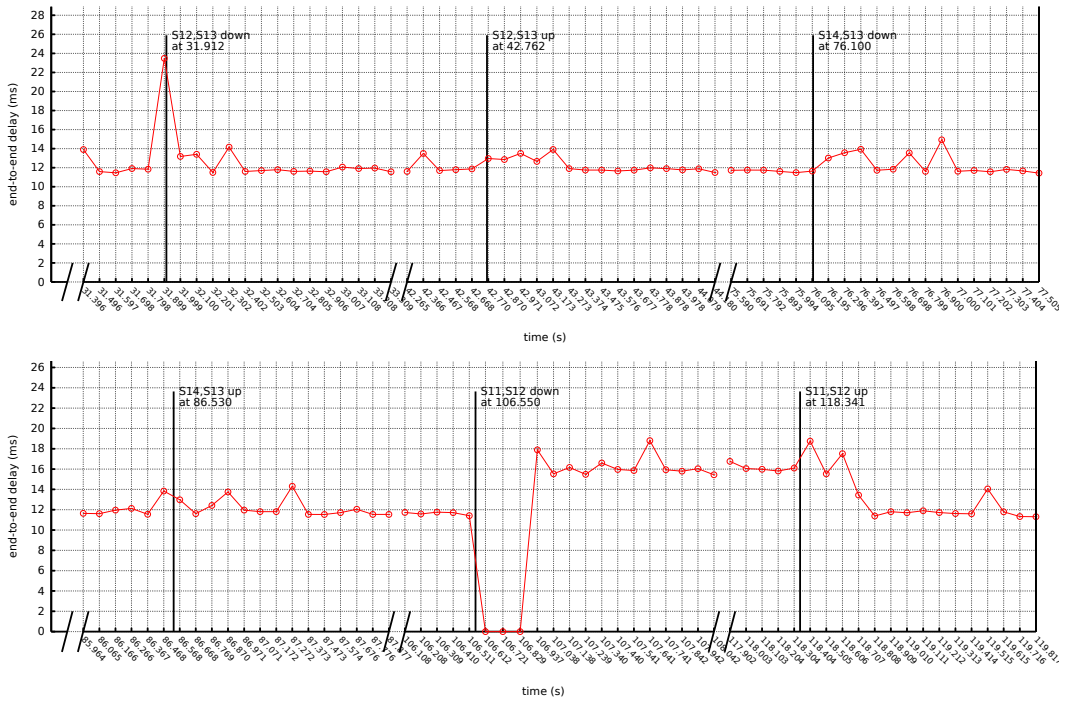


Figure 10: End-to-end delay for h3 and h4 communication with the disruptions.

IMPLEMENTATION 2.1 - H1 ↔ H3

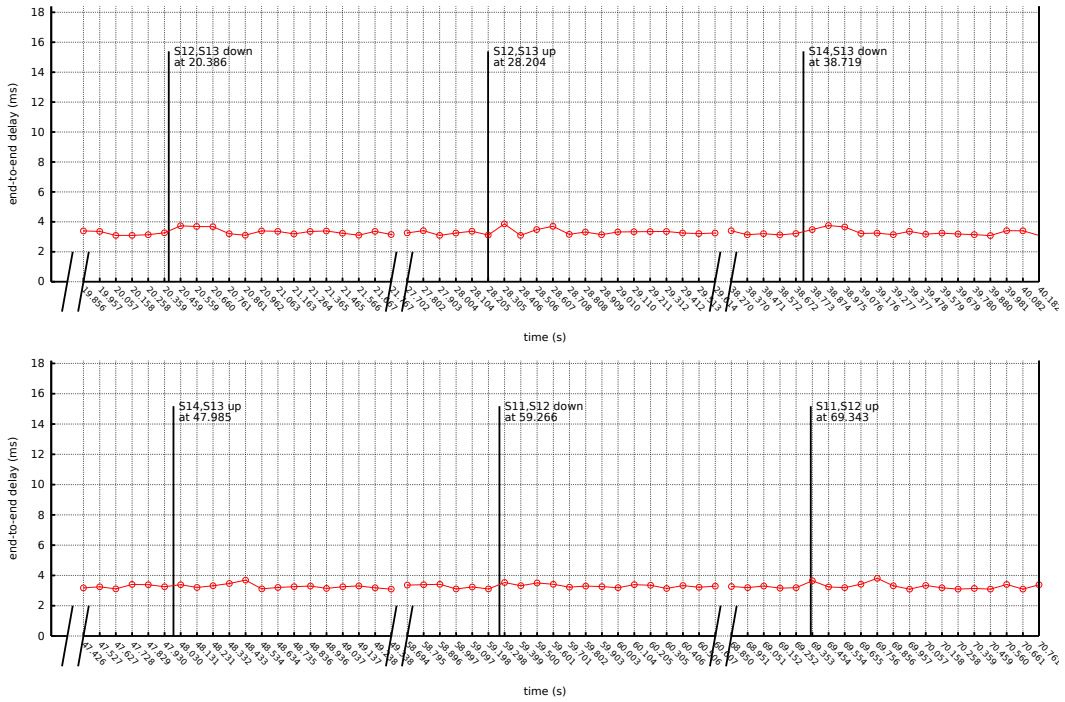


Figure 11: End-to-end delay for h1 and h3 communication with the disruptions.

IMPLEMENTATION 2.1 - H3 ↔ H4

Event	Source	Destination	Primary Path	# of Backup Paths
Beginning of communication	h3	h4	$\langle S7, S8, S9, S11, S12\rangle$	2
	h4	h3	$\langle S12, S11, S9, S8, S7\rangle$	2
link $\langle S12, S13\rangle$ down	h3	h4	$\langle S7, S8, S9, S11, S12\rangle$	1
	h4	h3	$\langle S12, S11, S9, S10, S7\rangle$	1
link $\langle S12, S13\rangle$ up	h3	h4	$\langle S7, S8, S9, S11, S12\rangle$	2
	h4	h3	$\langle S12, S11, S9, S8, S7\rangle$	2
link $\langle S13, S14\rangle$ down	h3	h4	$\langle S7, S8, S9, S11, S12\rangle$	1
	h4	h3	$\langle S12, S11, S9, S10, S7\rangle$	1
link $\langle S13, S14\rangle$ up	h3	h4	$\langle S7, S8, S9, S11, S12\rangle$	2
	h4	h3	$\langle S12, S11, S9, S8, S7\rangle$	2
link $\langle S11, S12\rangle$ down	h3	h4	$\langle S7, S8, S9, S11, S14, S13, S12\rangle$	1
	h4	h3	$\langle S12, S13, S14, S11, S9, S8, S7\rangle$	1
link $\langle S11, S12\rangle$ up	h3	h4	$\langle S7, S8, S9, S11, S12\rangle$	2
	h4	h3	$\langle S12, S11, S9, S8, S7\rangle$	2

Table 1: Chosen paths by implementation 2.1 after each perturbation made in the topology of the communications and the number of backup paths that existed between host h3 and host h4.

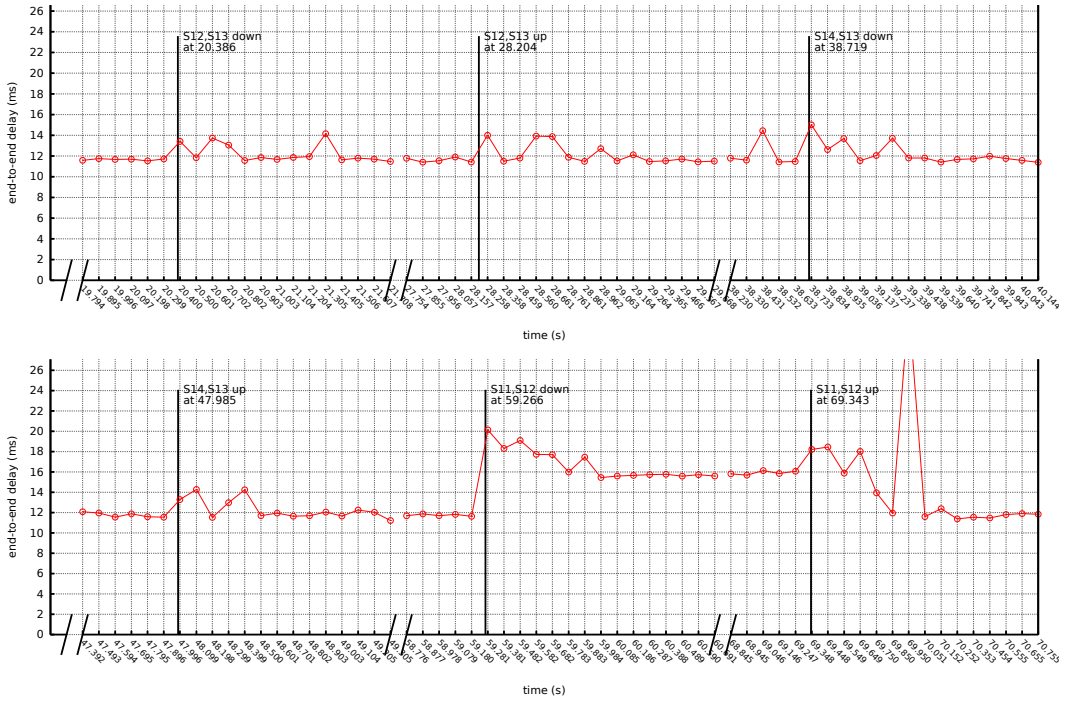


Figure 12: End-to-end delay for h3 and h4 communication with the disruptions.