Carrier Grade Resilience in Geographically Distributed Software Defined Networks

John Arundel Lewis

Supervisor:

Neco Ventura

Dissertation submitted to the Department of Electrical Engineering in fulfilment of the requirements for the Master of Science in Electrical Engineering degree at the University of Cape Town

December 2016

APPLICATION FORM

Please Note:
Any person planning to undertake research in the Faculty of Engineering and the Built Environment (EBE) at the University of Cape Town is required to complete this form **before** collecting or analysing data.The objective of submitting this application *prior* to embarking on research is to ensure that the highest ethical standards in research, conducted under the auspices of the EBE Faculty, are met. Please ensure that you have read, and understood the **EBE Ethics in Research Handbook**(available from the UCT EBE, Research Ethics website) prior to completing this application form: http://www.ebe.uct.ac.za/usr/ebe/research/ethics.pdf

APPLICANT'S DETAILS		
Name of principal researcher, student or external applicant		John Lewis
Department		Electrical Engineering
Preferred email address of applicant:		johnlewis0945@gmail.com
If a Student	Your Degree: e.g., MSc, PhD, etc.,	MSc
	Name of Supervisor (if supervised):	Neco Ventura
If this is a researchcontract, indicate the source of funding/sponsorship		
Project Title		Carrier Grade Resilience in Geographically Distributed Software Defined Networks

I hereby undertake to carry out my research in such a way that:

- there is no apparent legal objection to the nature or the method of research; and
- the research will not compromise staff or students or the other responsibilities of the University;
- the stated objective will be achieved, and the findings will have a high degree of validity;
- limitations and alternative interpretations will be considered;
- the findings could be subject to peer review and publicly available; and
- I will comply with the conventions of copyright and avoid any practice that would constitute plagiarism.

SIGNED BY	Full name	Signature	Date
Principal Researcher/ Student/External applicant	John Lewis	Signed	05 Dec 2016

APPLICATION APPROVED BY	Full name	Signature	Date
Supervisor (where applicable)	Neco Ventura	Signed	05 Dec 2016
HOD (or delegated nominee) Final authority for all applicants who have answered NO to all questions in Section1; and for all Undergraduateresearch (Including Honours).	E Bosse	Signed	7/12/16
Chair : Faculty EIR Committee For applicants other than undergraduate students who have answered YES to any of the above questions.			

Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

John Arundel Lewis

December 2016

Supervisor's Declaration

As the candidate's supervisor, I have approved this dissertation for submission.

Neco Ventura

December 2016

Acknowledgements

I would like to thank the following people for their help and support throughout this project:

My parents, for all their support, encouragement, and many hours of proofreading.

Neco Ventura, for his excellent supervision and guidance during the project.

Nyasha Mukudu, for the helpful technical discussions.

Niels van Adrichem, for sharing an early copy of ‘Computing backup rules in SDN’ and the associated source code.

Kim Mackey, for listening to my many ramblings about networks as I thought aloud.

Abstract

The Internet is a fundamental infrastructure in modern life, supporting many different communication services. One of the most critical properties of the Internet is its ability to recover from failures, such as link or equipment failure. The goal of network resilience heavily influenced the design of the Internet, leading to the use of distributed routing protocols.

While distributed algorithms largely solve the issue of network resilience, other concerns remain. A significant concern is network management, as it is a complex and error-prone process. In addition, network control logic is tightly integrated into the forwarding devices, making it difficult to upgrade the logic to introduce new features. Finally, the lack of a common control platform requires new network functions to provide their own solutions to common, but challenging, issues related to operating in a distributed environment.

A new network architecture, software-defined networking (SDN), aims to alleviate many of these network challenges by introducing useful abstractions into the control plane. In an SDN architecture, control functions are implemented as network applications, and run in a logically centralized network operating system (NOS). The NOS provides the applications with abstractions for common functions, such as network discovery, installation of forwarding behaviour, and state distribution. Network management can be handled programmatically instead of manually, and new features can be introduced by simply updating or adding a control application in the NOS.

Given proper design, an SDN architecture could improve the performance of reactive approaches to restoring traffic after a network failure. However, it has been shown in this dissertation that a reactive approach to traffic restoration will not meet the requirements of carrier grade networks, which require that traffic is redirected onto a back-up route less than 50 ms after the failure is detected. To achieve 50 ms recovery, a proactive approach must be used, where back-up rules are calculated and installed before a failure occurs. Several different protocols implement this proactive approach in traditional networks, and some work has also been done in the SDN space.

However, current SDN solutions for fast recovery are not necessarily suitable for a carrier grade environment. This dissertation proposes a new failure recovery strategy for SDN, based on existing protocols used in traditional carrier grade networks. The use of segment routing allows for back-up routes to be encoded into the packet header when a failure occurs, without needing to inform other switches of the failure. Back-up routes follow the post-convergence

path, meaning that they will not violate traffic engineering constraints on the network. An MPLS (multiprotocol label switching) data plane is used to ensure compatibility with current carrier networks, as MPLS is currently a common protocol in carrier networks.

The proposed solution was implemented as a network application, on top of an open-source network operating system. A geographically distributed network testbed was used to verify the suitability for a geographically distributed carrier network. Proof of concept tests showed that the proposed solution provides complete protection for any single link, link aggregate or node failure in the network. In addition, communication latencies in the network do not influence the restoration time, as they do in reactive approaches. Finally, analysis of the back-up path metrics, such as back-up path lengths and number of labels required, showed that the application installed efficient back-up paths.

Table of Contents

Abbreviations

API	Application programming interface
ARP	Address resolution protocol
BFD	Bidirectional forwarding detection
CPE	Customer premises equipment
EC2	Elastic Cloud Compute
ECMP	Equal-cost multipath
FIB	Forwarding information base
FRR	Fast reroute
GENI	Global Environment for Network Innovations
GiB	Gibibyte
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ITU	International Telecommunications Union
km	Kilometer
LACP	Link aggregation control protocol
LFA	Loop-free alternate
LOS	Loss-of-signal
LSP	Label-switched path
MPLS	Multiprotocol Label Switching
ms	Millisecond
NOS	Network operating system
OVS	Open Virtual Switch
P	Provider
PE	Provider edge
PHP	Penultimate hop popping
PLR	Point of local repair
PSTN	Public switched telephone network

QoS	Quality of service
REST	Representational state transfer
RLFA	Remote loop-free alternate
RSVP-TE	Resource Reservation Protocol – Traffic Engineering
RTT	Round trip time
SA	South Africa
SDN	Software defined networking
SLA	Service level agreement
SONET	Synchronous Optical Networking
SR	Segment routing
US	United States of America
VM	Virtual machine

Chapter 1

Introduction

The Internet is a fundamental infrastructure for modern life, with over 3.6 billion users [1]. While the Internet has disrupted many sectors such as education, communication, and research, it has created entirely new sectors as well. Many services and applications now rely on the Internet's robustness to meet business goals, and users expect a consistent level of service. Therefore, one of the most important characteristics of the Internet is its ability to maintain performance regardless of equipment failures in the network.

This ability to recover from failures, or resilience, has been a fundamental goal since the inception of the Internet, and significantly influenced its design. The distributed nature of the Internet's routing protocols contributed significantly to its resilience [2], since distribution means that there is no single point of failure. In addition, distributed systems generally scale better than centralized systems which may experience performance bottlenecks.

While distribution is an enabler of resilience, it significantly complicates management and configuration of the network. Many different devices must be individually configured, requiring an operator to use many different vendor specific and low-level languages. This makes device configuration and network management complex and error-prone [2]. As the network scales, the management becomes more complex [3]. The result is a system where innovation is slow, management and operational costs are high, and errors are difficult to trace and fix.

Distributed algorithms also hinder the flexibility of the network, since it is hard to implement new features in a large set of devices. Firmware upgrades are not always possible, and hardware upgrades are prohibitively expensive to perform regularly. In order to introduce new features into the network, such as load balancing, firewalls, and traffic engineering, operators rely heavily on specialized networking equipment, called middleboxes, to provide these functions [2]. Middleboxes are devices that transform, inspect, filter, or otherwise manipulate network traffic. This increases both the capital and operational cost of the network, as well as further complicating network management.

Network inflexibility impacts not only the network cost, but also revenue. While the cost per bit is not necessarily decreasing as the network scales [3], consumers increasingly want more

data at a lower cost. To remain profitable and attract customers, operators must therefore either make their networks more efficient, lowering the cost per bit, or they must look for additional revenue streams. A more flexible network could provide these additional revenue streams by enabling services that are currently challenging, such as dynamic bandwidth allocation and time-of-day access control.

These issues stem, in large part, from the lack of a common control platform or network-wide management abstraction [4] [5]. The control plane in traditional networks is tightly integrated into the forwarding devices, and its distribution model is fixed by the network topology. The data plane forwards traffic based on the local forwarding model calculated by the control plane. However, the process to calculate the local forwarding model relies on information from other nodes, and thus the control plane must perform non-local calculations. Finally, the management plane provides operational access and monitoring [6]. All three of these planes are contained within a traditional router, as shown in Figure 1 below.

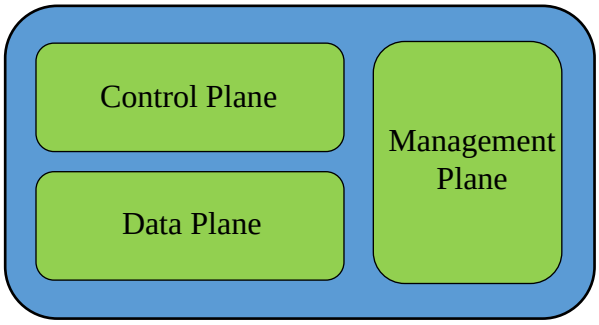

Figure 1 – Traditional operational planes of a router [6]

Due to the lack of a common control platform, new control functions must provide their own implementations of various common functions, such as network discovery and state distribution. This introduces a significant barrier to innovation, as the new functions must first solve the hard, low-level problem of designing a distributed protocol [4].

A new networking architecture, software-defined networking (SDN), has been proposed to alleviate some of these issues by introducing useful abstractions into the network. In an SDN paradigm, control functions run as applications within a logically centralized network operating system (NOS). Another common term for the NOS is ‘SDN controller’, and these terms will be used interchangeably. The NOS provides the applications with a global network view, as well as a programmatic interface to allow direct control of the network’s forwarding devices [6]. This architecture decouples the control plane from the data plane, allowing the switches to become simpler devices that only forward traffic and do not contain a control plane, and allowing the NOS to run on general purpose servers [4].

In addition, the control plane's distribution is no longer fixed by the underlying network architecture, and well-established cloud computing techniques can be used to run the NOS on a distributed set of servers, enabling reliability and scalability [4]. While the control plane will still be distributed, state distribution is now the responsibility of the NOS, and not the control functions. State distribution primitives only need to be implemented once in the NOS, and then made available to the network applications [4]. Figure 2 shows a high-level view of this architecture, as laid out in a recommendation from the International Telecommunications Union (ITU) [7].

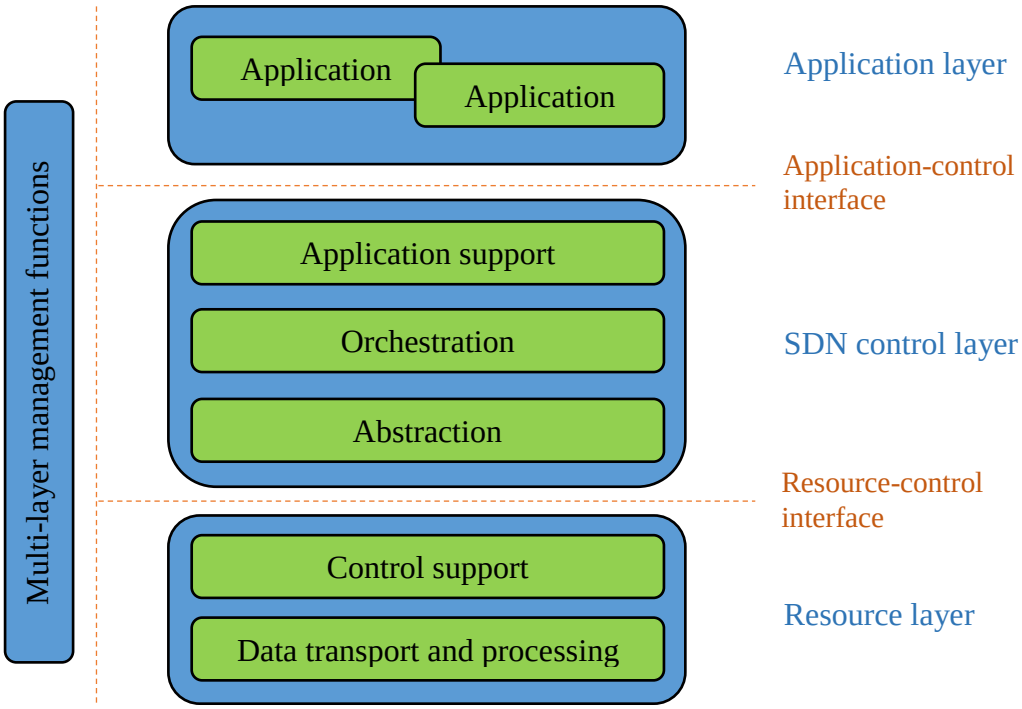

Figure 2 - High-level SDN architecture [7]

1.1. Research Motivation

The main aim of SDN is to introduce useful abstractions into the control plane, particularly by providing a global network view and a standardized, programmatic interface to the network devices. To accomplish this, the control plane is decoupled from the forwarding devices, which also allows the control plane software to be updated independently of the network hardware.

Although this approach has many well-documented benefits, and the field is maturing rapidly, there are still a number of open challenges that must be solved before adoption becomes more widespread [2]. Some of these challenges are introduced by the SDN architecture, while others are present in current networks.

One of these open areas is network resilience. As already mentioned, resilience is a key property of the Internet, and computer networks in general. With the control plane removed from the switches, they can no longer independently recalculate the forwarding information base (FIB) required to correctly forward traffic. This could limit the ability of the switches to handle failures in the data plane (e.g. link or node failure), or failures in the controller-switch communication link. Furthermore, a centralized NOS could introduce a single point of failure, that could disable the entire network if it were to fail.

Many of these concerns have been already addressed in practical SDN implementations. For instance, controller-switch link failures can be dealt with by routing control traffic through neighbouring switches that still have a connection to the NOS [8]. Reliability in the control plane can be achieved through replication or redundancy, using multiple active instances of the NOS or back up instances [9]. Switches can connect to multiple instances of the NOS (using only one as the active controller) to further mitigate against controller failure and controller-switch link failure.

Data plane failures, however, are still a challenging area of concern. This is true in traditional networks, but SDN introduces some additional concerns since the switches do not contain control logic.

Typically, traditional networks respond to failures by propagating the topology update through the network, and recalculating the FIB at each node based on the new topology. This process could take a few milliseconds, or hundreds of seconds, depending on the network and the switches [10]. During this process, traffic may be lost, and transient micro-loops often form. Since the Internet is a best-effort transport network, these failures are largely hidden from the users by fault handling mechanisms at the application level which retransmit the lost data [10].

A similar reactive approach is typically implemented in SDN, where switches notify the controller of the failure, and the controller calculates and installs a new FIB at the affected switches. This approach is less likely to form transient loops, but there are concerns that the control plane could form a bottleneck during a topology change.

There are many scenarios, however, where this reactive approach to network failures is insufficient. In carrier-grade networks, operators typically expect traffic to be redirected onto a back-up path within 50 ms of a failure being detected [11]. This requirement originates from Synchronous Optical Networking (SONET), an optical transport protocol that guarantees sub 50 ms failover times when back-up fibre links are available [10]. Since SONET is a physical

layer protocol, efforts have been made to provide similar failover times that are independent of the underlying physical layer. Various Fast Reroute (FRR) protocols have been developed to achieve this, including Internet Protocol (IP) FRR and Multiprotocol Label Switching (MPLS) FRR [10].

To implement FRR in a traditional network, a node must pre-calculate a set of back-up forwarding actions to take if a node or link fails. A simple approach is to pre-compute a back-up next hop that will not cause a loop to form. This next-hop is called a loop-free alternate (LFA) [12]. This approach does not require any co-ordination with other nodes, as it relies on the pre-failure network paths. However, the presence of a suitable LFA depends heavily on the topology, and protection cannot always be guaranteed.

To demonstrate the concept of LFAs, Figure 3a shows a hypothetical network topology where a suitable LFA is present. The numbers next to the links represent the weight of the link used in the routing algorithm, which chooses the path with the least combined link weight. The primary network path between the source S and the destination D uses node A, and is represented by solid green lines. To protect against a failure of link S-A, node B can be used as a backup next-hop. The pre-failure network path from B to D will use link B-D, and traffic will not loop. This backup path is represented by dashed green lines. If link S-A fails, node S can repair the path to D by rerouting traffic away from the failure, onto link S-B. This is a local repair action; hence node S is called the point of local repair (PLR).

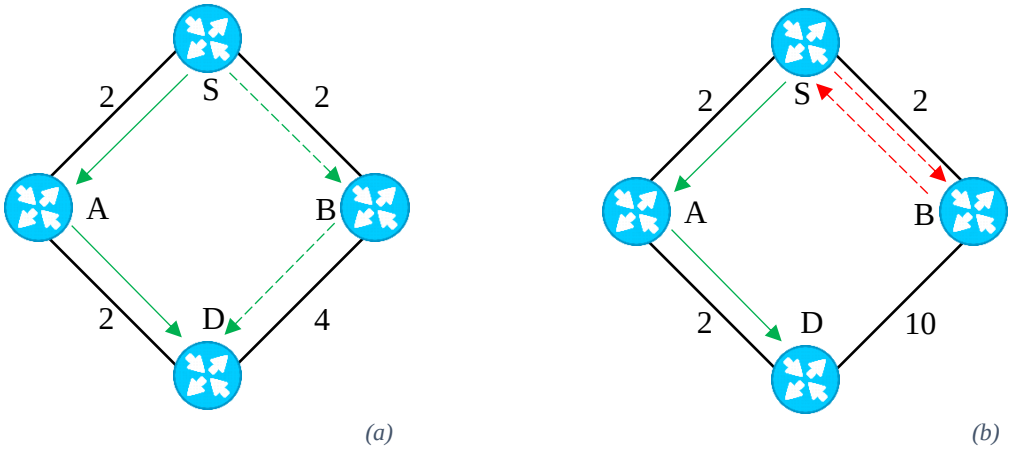

Figure 3 - Network topology with a suitable LFA (a) and without (b) [14]

However, if the link weight of B-D is increased to 10, as shown in Figure 3b, node B is no longer a suitable LFA. The primary path from B to D is via node S and node A, which means that if node S redirects traffic to B, it will be returned. Therefore, if link S-A fails and node S

attempts to repair the path by redirecting traffic to B, a loop would form, represented by dashed red lines.

To increase coverage (the percentage of failures the FRR scheme can protect against), the LFA method can be extended to use remote LFAs (RLFAs). A remote LFA is a node that will correctly forward the traffic to the destination using pre-failure network paths that do not traverse the failure, without causing any loops in the network. When a failure occurs, the PLR will forward traffic directly to an RLFA, generally using some type of tunnelling technique to ensure the traffic reaches the RLFA.

In IP networks, an IP tunnelling method such as IP-in-IP can be used [13]. In MPLS-enabled networks, tunnels can either be manually pre-provisioned, or dynamically established, but operators typically prefer to manually provision the tunnels [14]. Alternatively, a relatively new source routing protocol, Segment Routing (SR), can be used to encode the tunnel in the packet headers [15]. This removes the need to configure tunnels in the network.

SDN is particularly well-suited to implement a FRR strategy, as many of the complexities of FRR algorithms are solved by SDN's features. A FRR algorithm would no longer need to generate its own view of the network, as a global network view is provided by the NOS. The algorithm is also aware of the pre-failure forwarding rules in each switch. Control applications can programmatically configure switches with the required backup rules and tunnels, instead of operators manually configuring the routers. Finally, the control applications can run as a graph algorithm on powerful commodity servers, instead of as a distributed algorithm on resource constrained routers [16].

1.1.1. Problem Definition

To achieve SDN's primary goals of introducing control abstractions, simplifying network management, and enabling innovation, it is necessary to separate the control plane from the data plane. Removing the control plane from the forwarding devices raises several resilience concerns, particularly with regards to data plane failure handling.

One of the primary concerns is that a reactive approach to traffic recovery, where the controller is involved in calculating a new FIB, will not meet carrier-grade networks' FRR timing requirements. Thus far, SDN appears to have primarily been deployed in small to medium size networks, or in data-centres. In these environments, a reactive restoration approach has been sufficient, as the processing and communication delays involved are not

significant compared to traditional network re-convergence times. These networks typically do not have the same FRR requirements as carrier-grade networks.

As SDN moves more into carrier networks that are larger both in terms of geographical size, as well as number of nodes, a fast restoration approach will become more critical to SDN's success. SDN could also improve some of the issues with current FRR techniques, resulting in more widespread implementation of FRR.

With backup rules installed, it has been shown that the 50 ms fast failover requirement can be met, and exceeded, using SDN switches [17]. However, calculation of the backup rules is non-trivial, and requires careful consideration and design. Various metrics can be used to determine the success of the FRR algorithm, such as coverage, backup path length, or number of additional flow rules [18]. Additionally, the FRR algorithm should be suitable for use in a carrier network, which often covers a large geographical area and has traffic engineering constraints. The FRR solution should also be scalable, automatic, and provide protection against any single link or node failure. Ideally, it would require minimal or no change to existing protocols, and the algorithm should run entirely as a control plane application.

1.1.2. Research Questions

The main research questions investigated can be summarized as follows:

- Given ideal circumstances (a high-performance control plane and low latency controller-switch communication), can existing controller-based reactive approaches to failure recovery meet the 50 ms restoration requirement for carrier-grade networks?
- If a reactive restoration approach can meet carrier-grade requirements in ideal circumstances, what are the scaling limitations on the network? Of particular importance is the maximum controller-switch latency, and the number of NOS instances required for a particular topology.
- Are current SDN implementations of FRR suitable for a carrier-grade network? Important properties for carrier networks are scalability, coverage, automatic protection, and traffic engineering.
- Can new or existing FRR strategies for traditional networks be implemented in an SDN network to achieve carrier-grade resilience?

1.2. Dissertation Objectives

The success of software-defined networking in carrier networks will depend heavily on SDN's ability to meet the requirements of the operators. Resilience is an important requirement, as operators often have strict service level agreements (SLAs) with customers, and network downtime can have significant financial consequences.

The first objective of this dissertation is to present the available literature on current approaches for handling failures in the SDN data plane. To do this, a brief overview of the relevant aspects of the SDN architecture will be presented, along with a review of the reactive and proactive strategies of handling failures.

Secondly, the suitability of current methods of recovering from data plane failures will be analysed, as well as their scaling limitations. This will be done by performing experiments using a network testbed with ideal network conditions, as well as more realistic conditions.

Finally, the primary aim of this dissertation is to design and implement a fast reroute SDN control application that could be suitable for use in a carrier grade network. The focus will be on calculation of the backup rules. To analyse the suitability of the control application for carrier networks, the application will be tested using network topologies that are representative of carrier networks. The application will also be tested in a geographically distributed network testbed.

1.3. Scope and Limitations

The scope of this dissertation is limited to SDN resilience as it relates to data plane failures, specifically, either link or node failures. The literature review will, however, briefly consider other aspects of SDN resilience.

The scope is further limited to networks within a single administrative domain, that is, controlled by a single operator. Protection for single points of failure in the network will not be considered, as this cannot be performed using back-up routes but requires provisioning additional links or switches. Furthermore, only the core of the network will be considered. Carrier networks consist of three layers: the access layer, the aggregation or distribution layer, and the core layer. The core network is where fast reroute strategies are most critical, as the core is responsible for transporting large amounts of traffic.

It is assumed that if a switch has been programmed with backup rules, it will be capable of rerouting traffic within 50 ms of detecting a failure. This has been shown to be true in

numerous experiments, and can also be expected since current production switches are required to be capable of meeting this switching time. Therefore, the proposed solution is not required to meet the 50 ms timing requirement if lower-performance switches are used, but it must be capable of installing backup rules that guarantee protection against failures.

The scope of this dissertation includes fast reroute for unicast traffic, and does not include multicast traffic.

1.4. Dissertation Outline

The remainder of this document is structured as follows.

Chapter 2 presents a brief overview of the architecture of SDN, to provide context to the work, as well as some necessary background on carrier routing protocols. A more in-depth literature review of current approaches for data plane resilience in SDN is then presented. The literature on both reactive and FRR approaches is considered.

Chapter 3 presents the requirements of a FRR approach for carrier networks, as well as the design of the proposed approach. The requirements and design of a network testbed suitable for evaluating the proposed solution, as well as current data plane failure recovery techniques, is also presented. The network testbed must allow for testing of the control applications in ideal network conditions, as well as more realistic conditions.

Chapter 4 presents the implementation of the proposed FRR algorithm, as well as the network testbeds. The proposed FRR solution is implemented as a network application, built on top of an existing open-source NOS. Two network testbed platforms are used to test the existing and proposed solutions in different environments. The validation and performance testing tools are also described.

In Chapter 5 the results of the proposed FRR algorithm are detailed, focusing on coverage against failures as well as back-up path metrics. The performance results of the current reactive approach are also presented. The existing and proposed data plane resilience techniques are subjected to validation and performance tests under ideal and realistic network conditions.

Chapter 6 presents some conclusions from the work, and summarizes the contribution of the dissertation. Future work is also discussed.

Chapter 2

Background and Literature Review

The previous section introduced the basic concepts and aims of SDN, as well as some of the resilience concerns surrounding the architecture. In particular, the problem of data plane resilience was expanded in more detail. This section provides more detail on the architecture of SDN and its main components. Then, a brief description of routing protocols used in carrier networks is given, as necessary background for the fast reroute (FRR) approach proposed in later chapters. Finally, existing reactive and proactive data failure recovery strategies are described in more detail, with a focus on software-defined networks.

2.1. Software-Defined Networking Architecture

As shown in Figure 2, the ITU's reference architecture for SDN contains three main layers: the resource layer, the control layer, and the application layer. Figure 4 shows a simplified view of this same reference architecture, highlighting the main components [2]. Instead of the ITU terminology of application-control and resource-control interfaces, the terms northbound application programming interface (API) and southbound API can be used. Northbound and southbound are software directions from the reference point of the NOS. This section will follow a bottom-up approach to discuss each of the main components.

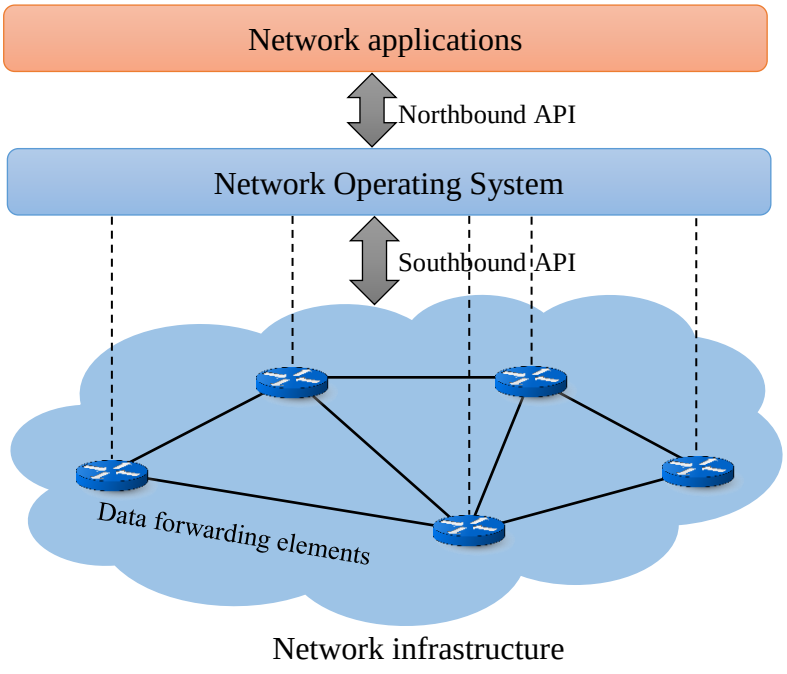

Figure 4 - A simplified view of the SDN architecture [2]

2.1.1. Data forwarding elements

On the lowest layer, the data forwarding elements, or switches, are responsible for forwarding traffic based on their forwarding tables, which are installed by the NOS. They also capture network statistics and communicate these back to the NOS. A commonly deployed SDN switch architecture is the OpenFlow architecture, specifically OpenFlow version 1.3 [19]. Later versions have been released, but are not as widely supported as version 1.3 [20].

OpenFlow 1.3 switches contain one or more flow tables, as well as a group table [21]. Received packets enter the data processing pipeline, consisting of a series of flow tables. The flow tables contain flow entries, which consist of a packet matching rule, actions to apply to matching packets, and traffic statistics. Actions include forwarding, modifying, or dropping the packet. Packets can be forwarded to a port, a flow table, a specific group in the group table, or to the controller [21]. Figure 5 shows the breakdown of a flow rule in more detail, as well as a few of the more than 40 possible matching criteria in OpenFlow 1.3 [2].

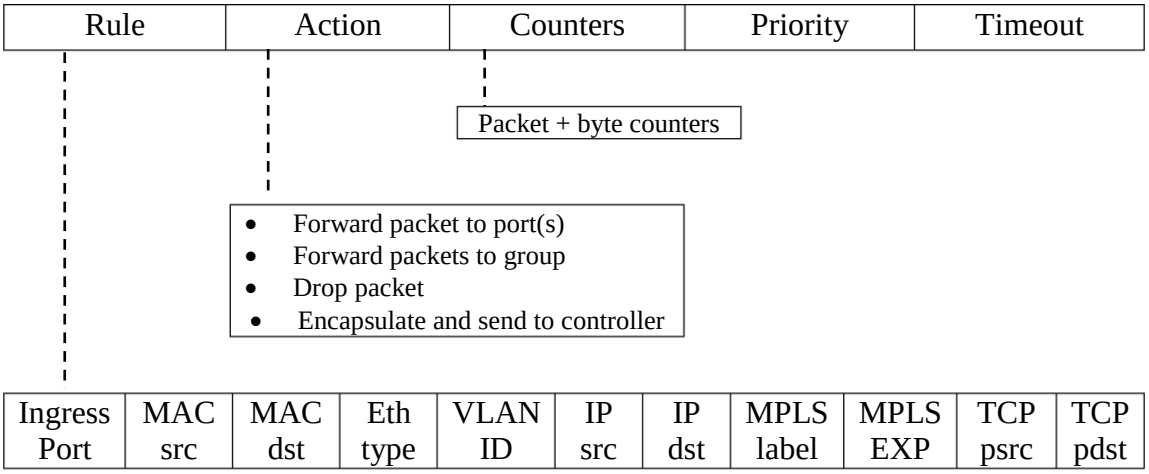

Figure 5 - Breakdown of an OpenFlow rule [2] [52]

A group allows for more complex and specialized packet operations than can easily be defined in a flow table, such as load-balancing, multicast, and fast failover [19]. The fast-failover type group is of particular relevance to this work, as it allows for a fast reroute function to be implemented in the switch.

A group contains a list of actions, with each item in the list referred to as a bucket. Fast-failover group buckets contain a watch port (or watch group) parameter, as shown in Figure 6. The status (up or down) of the associated watch port determines whether or not the bucket can be used. When a packet is forwarded to the group, the first usable (or live) bucket is selected, and its list of actions is applied to the packet [19].

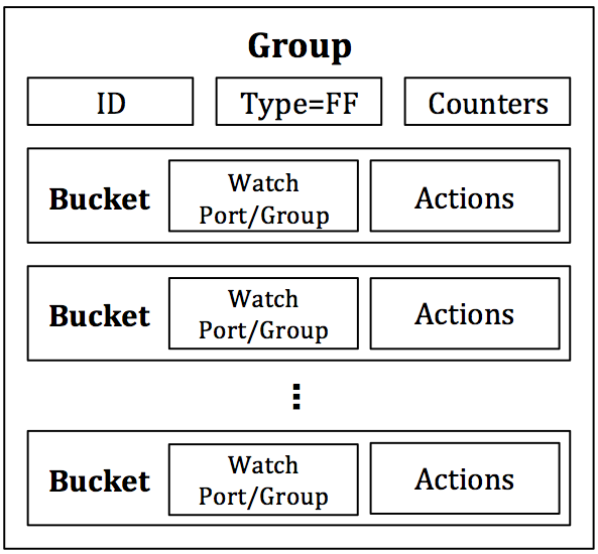

Figure 6 - The OpenFlow fast-failover group [19]

A fast reroute strategy can therefore be implemented by installing the action corresponding to the primary path into the first bucket in a fast-failover group, and the action corresponding to the backup path into the second bucket in the same group. The first bucket should monitor the liveness of the primary output port. If this port goes down due to failure of the link or neighbouring node, the primary bucket will be disabled, and the actions in the second group will be applied to the traffic.

The select group type is also relevant, as it allows for load balancing traffic across multiple paths, or multiple parallel links between two switches. A watch port can also be specified for a select group bucket, which is used to determine the liveness of the bucket. A packet entering a select group may use any of the live buckets within the group. The bucket selection algorithm is not specified by the OpenFlow protocol, and is left to the switch implementation. Load balancing can either be done on a per-packet basis, or a per-flow basis. A flow is a set of packets from a source to destination.

2.1.2. Southbound API

The southbound interface connects the switches and the NOS, allowing the NOS to install forwarding rules, collect traffic statistics, and gather the global network view. A number of southbound APIs exist, such as OpenFlow, NetConf, and ForCES [2]. The most common of these is the OpenFlow protocol [2], which is the southbound protocol used by OpenFlow switches. However, NOSes often support more than one southbound API.

The OpenFlow protocol provides an open, standard interface for SDN controllers to

communicate with switches, in a way that abstracts the implementation details of the forwarding pipeline. This is a significant shift away from the status quo of vendor specific network languages, and creates a suitable environment for active, open-source development.

This standardized, open interface is particularly useful as it allows a network operator to use switches and controllers from multiple vendors or open-source projects. It also means that the operator is not locked in to a particular NOS or switch, as either can be substituted for another OpenFlow supporting equivalent [2].

2.1.3. Network Operating System

The network operating system, or SDN controller, provides similar functions for network applications as a computer operating system provides for computer applications [22]. Its primary purpose is to abstract device-specific details, and provide common functionalities and essential services. Some of these services and functionalities include network state and topology, device discovery, distribution of network configuration, as well as installation of forwarding rules.

The network operating system is, of course, a critical component in the SDN architecture, and its design must be suited to the requirements of the network. To accommodate different networks, a wide variety of NOSes have been developed with widely varying architectures, feature sets, and performance characteristics. When choosing a NOS, a number of selection criteria are important, such as architecture, industry partners, supported northbound and southbound APIs, modularity, documentation, intended application domain, and existing applications.

One of the primary distinguishing features of a NOS is whether it is centralized or distributed. Production controllers for large, resilient networks should be distributed in order to provide fault tolerance in the control plane, as well as allowing for performance to be scaled by adding more instances of the controller. However, distributed controllers must maintain state between instances, which does introduce some additional complexities.

While many different NOSes exist, OpenDaylight and ONOS are notable as they are arguably the best candidates for production networks. The main differentiator between these two is that ONOS is primarily focused on carrier networks, while OpenDaylight originally focused on the data-centre [20]. These different focuses influence the architecture to some extent, as well as the available network applications.

A third controller, RYU, is relevant for this work as it is one of the most commonly used for research and development purposes. However, given that it has a centralized architecture, and therefore no resilience in the control plane, it is not suitable for a production carrier-grade network [20].

Table 1 presents a comparison of some of the features of these three controllers. A more exhaustive survey and comparison of most of the available controllers can be found in [20] and [2].

Table 1 - Feature based comparison of three popular SDN controllers

	Programming Language	**Modularity**	**Distributed/ Centralized**	**Southbound APIs**	**Northbound APIs**	**Application Domain**
ONOS	Java	High	Distributed	OF1.0, 1.3, NETCONF	REST API	Datacentre, WAN and transport
ODL	Java	High	Distributed	OF1.0, 1.3, 1.4, NETCONF/YANG, OVSDB, PCEP, BGP/LS, LISP, SNMP	REST API	Datacentre
RYU	Python	Fair	Centralized	OF 1.0, 1.2,1.3, 1.4, NETCONF, OFCONFIG	REST API	Campus

2.1.4. Northbound API

The northbound API is the interface between the NOS and the network applications which calculate the forwarding state of the switches, based on high level network strategies and goals. The abstractions and functionalities provided by the NOS must be made available to the applications through the northbound interface.

Unlike the southbound, where OpenFlow is widely-adopted, the northbound interface does not yet have a widely-adopted, standardized protocol. Although some standardization organizations are working on a northbound API [23], the common trend in software is likely to occur, where standards typically follow implementation [2].

Currently, most controllers have at least two northbound APIs. The first is a programmatic interface, specific to the controller. This requires the application to be written in the same language as the controller, and allows the application to read network state, be updated with new topology information, and program the switches. This approach is useful as it allows the NOS to notify the application when new information is available, and the interaction between

the application and the NOS can make use of all the features of the programming language.

The second API generally provided is a representational state transfer (REST) API. REST is a common architectural style used in web development, and allows a client to communicate with a server using a simple set of Hypertext Transfer Protocol (HTTP) methods. Since most controllers provide similar REST interfaces, network applications could be adapted to work with another controller with less work than the programmatic interface would require. However, REST APIs have a number of architectural constraints. A significant constraint of the REST interface is that the application cannot subscribe to notifications about new information, but must continuously poll the NOS. This can result in a significant processing overhead, or a significant delay in receiving new information.

2.1.5. Network applications

Network applications contain network goals and business policy, and calculate the forwarding state of the switches accordingly. Common goals are routing, isolation, security, traffic engineering, and load balancing [2]. Many NOS vendors provide a number of network applications to cover common functionalities. The preferred approach is a modular or micro-service approach [20], where each application is responsible for one specific function. This allows for these applications to be easily reused in different combinations, depending on the particular use case.

The majority of popular NOSes are open-source, allowing users to modify the internals of the controller, as well as develop new applications to meet their own requirements. This allows for a faster pace of innovation than the traditional model of relying on vendors to introduce new features.

2.2. Carrier Network Routing Protocols

To properly discuss FRR strategies in carrier networks, it is necessary to first understand the routing protocols used in these networks. Operators often make use of MPLS in the core of their network. The original benefit of MPLS was that MPLS switches could operate faster than IP switches, but as switching hardware has improved, this benefit has largely fallen away. MPLS has other benefits that are still relevant, such as traffic engineering and quality of service (QoS) capabilities [10], as well as the ability to transport a variety of traffic protocols. However, some scalability issues have been identified with the traffic engineering solutions in MPLS, and a new routing protocol, segment routing (SR), has recently been proposed. Segment routing

has a number of benefits, and can utilize the MPLS data plane without modification, possibly allowing for a phased introduction.

2.2.1. Multiprotocol Label Switching

In IP networks, each switch runs a routing algorithm to determine the next-hop for a particular traffic flow, based on the IP packet header [24]. In an MPLS network (Figure 7), however, routing algorithms are only run at the edge of the network, in the provider edge (PE) routers [25]. The PE router determines the path that the traffic should take through the network, and assigns a short, fixed-length MPLS label accordingly. Paths corresponding to the different labels are installed in the interior provider (P) routers, using a forwarding table [24]. These paths are called label-switched paths (LSPs), since the P routers switch traffic based only on the MPLS label.

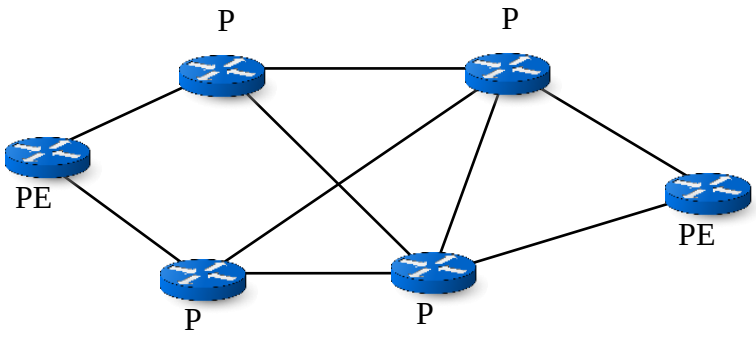

Figure 7 – An MPLS network topology showing P and PE routers [25]

MPLS enables QoS in the network by creating different LSPs for the same path, that have different priorities. Traffic engineering is enabled with the Resource Reservation Protocol – Traffic Engineering (RSVP-TE), which allows LSPs to be created that take into account network constraints and traffic requirements.

Although the interior P routers are simplified by MPLS, as they do not have to run a routing algorithm to make forwarding decisions, a large amount of network state is still required to be stored in these routers. Each LSP requires an entry in the P router, and it has proven difficult for providers to cope with the thousands of paths often needed in the network [15].

2.2.2. Segment Routing

Segment routing is similar in concept to MPLS in that it uses labels for routing, but unique labels are assigned to nodes (and links, if required), instead of being assigned to paths. At the

edge of the network, PE routers attach a stack of labels to the traffic that defines its path through the network [15]. The forwarding table of each P router therefore only contains entries for each node in the network, and not for each LSP. In this way, the network state is removed from the core of the network, and is only held in the edge routers.

This approach is very flexible, as PE routers can specify the traffic's path with various levels of granularity by either attaching a single label, specifying the final PE, or it can specify intermediate nodes as well. If too many labels are attached, issues may arise with MPLS switches that only support attaching a limited number of MPLS labels to a particular packet. However, various techniques can be used to minimize the stack depth while ensuring the optimal path is used [26]. A number of prototype SDN controllers have been developed showing that SR is a viable SDN forwarding protocol [27] [28].

To illustrate the operation of segment routing, a SR network is shown in Figure 8, where each node has a unique SR label between 9000 and 9007. Links can be labelled with an adjacency label, as shown for the two links between node 9002 and 9005. Adjacency labels are only significant to a specific node. Incoming traffic at PE1 that is destined for PE2 can be labelled with a single label, 9002. The traffic would then take the shortest path through the network, which would be determined by the routing protocol used and the link weights. With equal link weights, the traffic would be load-balanced over multiple paths if the routing protocol supports equal-cost multipath (ECMP) routing.

However, a specific path can also be enforced by attaching additional labels for intermediate nodes. For example, to force the traffic through node 9002, the edge router would simply push the label stack 9002, 9007. Traffic would still be load balanced over ECMP paths, if available. To explicitly specify the entire route, including a specific link, the edge router can push the label stack 9001, 9002, 2023, 9006, 9007.

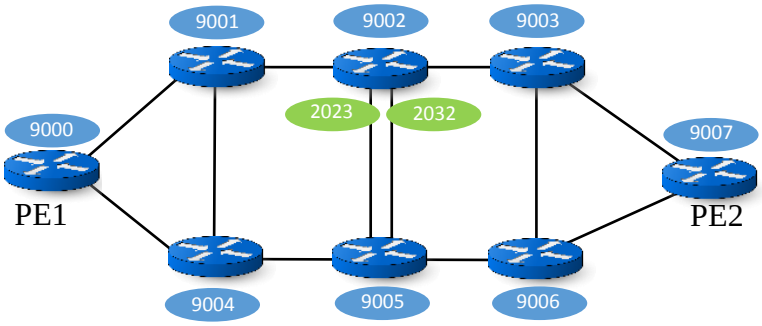

Figure 8 – A SR network showing node and adjacency labels [15]

2.3. Data Plane Failure Recovery

As discussed in Chapter 1, there are two basic approaches to responding to data plane failures. A reactive recovery approach will only recalculate the forwarding rules once a failure has occurred, and there may be traffic loss for several seconds. A proactive approach pre-emptively installs back-up forwarding rules in case of a failure, which can reduce traffic loss to below 50 ms. This section provides more detail on each of these approaches, focusing on implementations in SDN.

2.3.1. Reactive Approaches

In traditional networks with distributed control planes, data plane failures are communicated to other nodes by broadcasting the information as a topology change. Nodes then re-compute the forwarding tables accordingly. The time involved in distribution of the information and re-computation defines the minimum time for responding to the network failure [4]. This can be a lengthy process in some networks, as the information must travel through multiple hops, and the processing time of the routing algorithm is constrained by the computing power of the switches.

In a software-defined network, there are five basic steps that occur in order to recover from a failure, as shown in Figure 9 [16]. In order, they are:

1. A switch detects the failure
2. The switch notifies the controller of the failure
3. The controller computes the required updates
4. The updates are pushed to the switches
5. The switches update their forwarding table

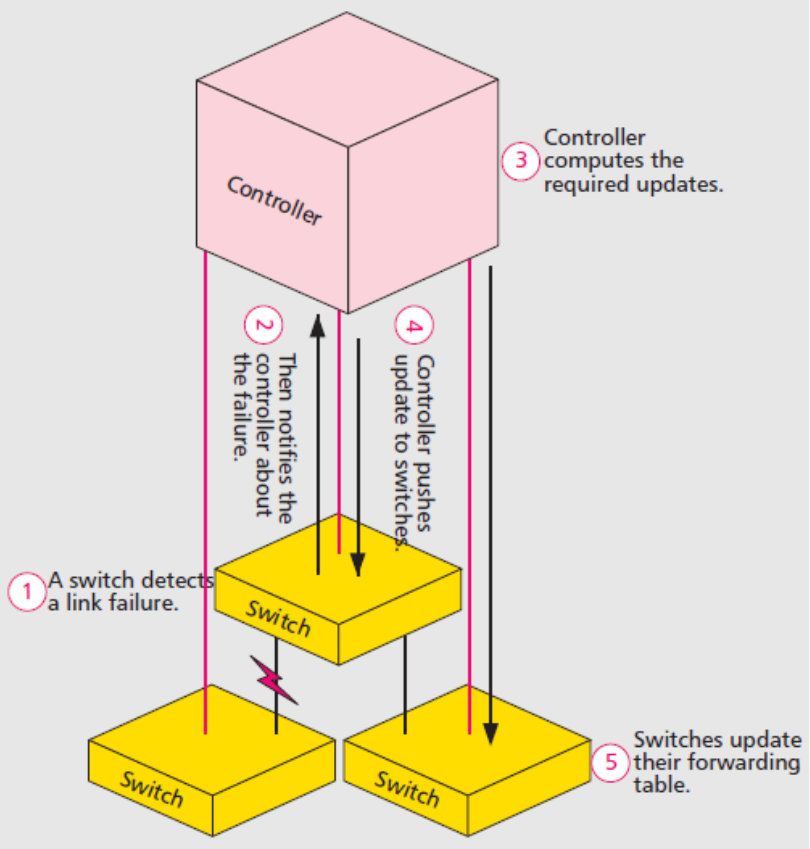

Figure 9 - The five steps that must occur when a failure occurs [16]

This process could be faster than in a traditional network, and at least will not be worse. This is because the switches communicate the change directly to controller, rather than flooding the network, and the controller can run on more powerful server hardware than the weak management CPUs in switches [16].

Therefore, the restoration time (T_r) can be broken down into four main components, as shown in Equation (1) below. These are the failure detection time (T_d), the switch-controller communication latency (T_{cs}), the computation time (T_c), and the flow installation time (T_i).

$$T_r = T_d + 2T_{cs} + T_c + T_i \quad (1)$$

The restoration requirement specifies that traffic must be redirected within 50 ms of the failure being detected [29], and therefore the 50 ms requirement does not include T_d. However, the failure detection time T_d is still important, as it is critical to minimizing traffic loss. A commonly used network protocol to detect link or node failures is Bidirectional Forwarding Detection (BFD) [11]. Two nodes establish a session over a particular path, and exchange frequent control messages to confirm that the path has not failed [17]. Alternatively, a simple loss-of-signal (LOS) approach can be used [11].

Sharma et al. used BFD sessions established between the path end-points to detect failures within 40 to 44 ms [11]. By using per-link BFD sessions instead, and increasing the frequency of the control messages, van Adrichem et al. were able to reduce the detection time to approximately 3 ms [17].

The switch-controller latency T_{cs} is largely a function of the network between the switch and controller, and will depend on the network. This factor can be reduced by geographically distributing the controller instances, so that each switch has a nearby controller. The network between the switch and controller could also possibly be improved. Both of these steps would increase the cost of the network, and the physical properties of the transmission link would still pose a restriction, since the latency of an optical fibre link is approximately 1 ms per 200 km [30]. A possible solution would be to first determine the maximum allowable value for T_{cs} by measuring the other variables in Equation (1), and then designing the network to achieve this value.

The computation time T_c is dependent on a number of factors, such as the routing application, the controller performance, and the hardware the controller is running on. The performance of the controller in terms of responses per second and response time appears to be high enough to meet carrier-grade requirements. A comparison of the most popular controllers found most of the controllers tested had a response time of less than 10 µs, and many can process upwards of a million responses per second [20]. This does not include the recalculation time, which depends on the number of flow rules that must be recalculated, and has been found to exceed 100 ms in several implementations [11] [31].

Finally, the flow installation time T_i is determined by the switch implementation. Flow installation times for a popular software switch, Open vSwitch, were found to be around 1 ms per flow installation [32]. These results are several years old, but show that flow installation could become a significant bottleneck in large-scale networks using software switches, if performance has not improved in recent years. Hardware switches are more likely to be used in a carrier network, however, and these should have a lower installation time.

Given that the four variables that contribute to the total restoration time in Equation (1) are all in the millisecond timescale, it appears unlikely that a restoration approach will be viable to meet the carrier-grade requirement of 50 ms. Furthermore, given that two of these variables, T_c and T_i, are dependent on the number of flows affected, a restoration approach will have scaling limitations.

2.3.2. Proactive Approaches

Traditional approaches to fast reroute (FRR) were described in Chapter 1.1. Similar concepts can be applied in SDN, where the global network view of the network simplifies the task of calculating the optimum back-up paths. In addition, the programmability of the network could alleviate some of the network management complexity of current FRR solutions, which has limited its deployment in production networks. In an SDN FRR solution, there are two main tasks: calculating the optimum back-up paths before a failure occurs, and configuring the data plane to use the back-up paths as soon as a failure occurs. This section will first discuss the process of configuring the data plane, and then discuss approaches to calculating the back-up paths.

Data plane configuration

In contrast to reactive approaches, proactive approaches to SDN data plane resilience preconfigure the switches to allow them to reroute traffic onto a back-up path without involving the controller. In a proactive strategy, there are two steps that the data plane must take in order to restore traffic:

1. A switch detects the failure
2. The switch selects the back-up rules

The restoration time calculation is modified from Equation (1) to reflect these two steps, and presented in Equation (2) below, where T_s represents the time taken to select the back-up flow rules.

$$T_r = T_d + T_s \tag{2}$$

The failure detection method is just as critical as in a reactive approach, and the same methods can be used. It is typically the most significant factor in the traffic restoration time, as the selection of back-up actions is very fast [17] [33]. Since the 50 ms requirement does not include the failure detection time, it is based only on T_s for a proactive approach.

Two different methods of selecting the backup rules were proposed prior to the introduction of OpenFlow fast failover groups in OpenFlow 1.1. The first project, OpenState, extends the OpenFlow traffic processing pipeline by introducing a finite state machine, allowing different sets of flow rules to be used depending on the state held in the switch [34]. When a failure is detected, the state of the switch changes, and a different set of rules are selected. This approach introduces additional complexity in the switch, which goes against some of the principles of

SDN. This, combined with the fact that it requires use of non-standard protocol, means that is unlikely to see much real-world adoption.

The second proposal was to install back-up rules into the same flow table as the primary rules, but with a lower priority. An auto-reject mechanism was implemented, which automatically deleted the primary rules when a failure occurred, causing the back-up rules to be selected [33]. Back-up actions are installed per-path, as requested, and require additional flow rules for each possible link and node failure on the protected path. This leads to a high number of flow rules needed, even in small to medium size networks. This approach also requires an extension of the OpenFlow protocol, and fast failover groups in later OpenFlow standards perform the same function, so it is also unlikely to see real world adoption.

OpenFlow fast failover groups allow the status of a port to influence the forwarding actions that are applied to a traffic flow, as explained in section 2.1.1 of this chapter. This implementation has the advantage that it does not require additional matching rules in the flow table, and it is a standard OpenFlow protocol which is already widely adopted.

Several works have implemented a protection scheme using the OpenFlow fast failover groups, and have shown that the recovery time is independent of path length and network size, since it is a local action [17].

Using per-path BFD to detect failures, Sharma et al. achieved a recovery time of 42 – 48 ms [11]. By using a per-link BFD configuration with more frequent control methods, van Adrichem et al. achieved a recovery time of 3.3 ms [17], which is substantially faster than the 50 ms requirement for carrier grade networks. This recovery time also includes the failure detection time.

However, implementing a fast reroute scheme that provides protection against all or most failures is not as simple as configuring a fast failover group to output traffic to a different port. The existence of an adjacent LFA is topology dependent, and often traffic must be forwarded to a remote LFA.

A common SDN approach to ensure that the traffic is forwarded to the RLFA is to install back-up flow rules in each switch along the back-up path. Packets are tagged at the point of local repair to notify other switches that they should be forwarded according to the back-up rules [18] [34] [35]. Once the traffic reaches the RLFA, the tag can be removed, and the primary forwarding rules used to reach the destination [35].

Back-up rule calculation

An early approach to calculate back-up rules took an end-to-end path protection approach, using disjoint paths [36]. Disjoint paths are paths that do not share any link or node, besides the end points. MPLS was used in the data plane, with a primary label switched path (LSP), as well as a disjoint back-up LSP. The end point switches monitored the connectivity of the primary path and switched over to the back-up path when the primary path failed. However, using disjoint paths is inefficient as the resulting primary path can be substantially longer than the shortest path possible [35]. Additionally, a single failure on a multi-hop path would cause the capacity of all the links to be wasted.

A later attempt, implemented on top of OpenState, used an approach where the traffic is tagged and sent back along the primary path after a failure, until a node is reached that can calculate a suitable alternate path to the destination. The tagged packets signal the failure to this repair node, and all subsequent packets are redirected down this alternate path [34]. This approach essentially selects a remote LFA on the primary path, at a point before the failure. However, a more optimal RLFA could easily exist that is not on the primary path. This approach will result in packets arriving out of order, which may be a problem for certain protocols. The initial paths for the repair traffic are relatively long, and could possibly be shortened by using the optimum RLFA. A further drawback of this approach is that it uses a custom OpenFlow extension, and adds extra state into the nodes.

Finally, a solution for IP networks was proposed by van Adrichem et al. [35]. This approach recognises that traffic can be re-routed along the primary path, to only avoid the failure, and still use the other available links. This is more efficient than a disjoint path method, which avoids all resources along the primary path. To calculate the back-up route in case of a specific failure, the element (link or node) is removed from the network graph, and the new shortest path found. This shortest-path is installed in the nodes along the path as additional forwarding rules, that match on the tag added by the node that detected the failure. The back-up action tags the traffic with information of the failure so that the correct forwarding rules are selected in following switches. This essentially creates a tunnel to the RLFA, where the tag is removed and the traffic uses normal primary forwarding rules to reach the destination. Traffic engineering was not considered in this approach, and the presence of multiple equal cost paths required significantly more rules to be installed. In addition, the solution does not support multiple parallel links, called link aggregates, between two nodes. Carriers often use traffic

engineering, link aggregates, and equal cost multipath routing (ECMP) for load balancing, so a carrier solution would require these issues to be addressed.

2.4. Chapter Discussions

This chapter first presented an overview of the SDN architecture, with a focus on fast failover groups, which are useful for a fast reroute strategy. Secondly, some background on carrier routing protocols was presented. Some of the challenges of reactive approaches to data plane failures were highlighted, as well as current efforts in this area. These challenges will make it difficult to achieve scalable carrier grade resilience with a reactive approach. Finally, the data plane support for proactive strategies was discussed, and the challenges in the control plane calculation of the back-up rules highlighted. Many current solutions have inefficient back-up paths, with the most suitable solution for IP networks being proposed by van Adrichem et al. However, a carrier solution would require some additional features, such as support for link aggregates, ECMP routing, and traffic engineering.

The following chapter will introduce the requirements of a FRR approach for carrier networks in more detail, as well as the design of the proposed solution. The requirements and design of a suitable network testbed for evaluation is also presented.

Chapter 3

Requirements and Design of a Carrier Grade SDN FRR Solution

Chapter 2 presented the current work on fast reroute solutions for SDN. The main aim of this dissertation is to develop a solution for the core of carrier grade networks, and therefore it is important to consider the requirements of these networks. This section will first look at carrier networks, specifically focusing on the core of the network. From this, the requirements of the proposed solutions will be formulated. The algorithm of the proposed solution will then be presented.

3.1. Carrier Core Network

This section details the architecture of a typical carrier network, and then considers the topology and functions of the core network. The requirements of the core network are then discussed, and used in section 3.2 to determine the requirements of the FRR solution.

3.1.1. Carrier Network Architecture

Due to the scale of national carrier networks, operators of these networks typically use a three-tier hierarchical network model to design a reliable, scalable and cost-efficient network [37] [38]. The three layers are: the access layer, the distribution layer, and the core layer.

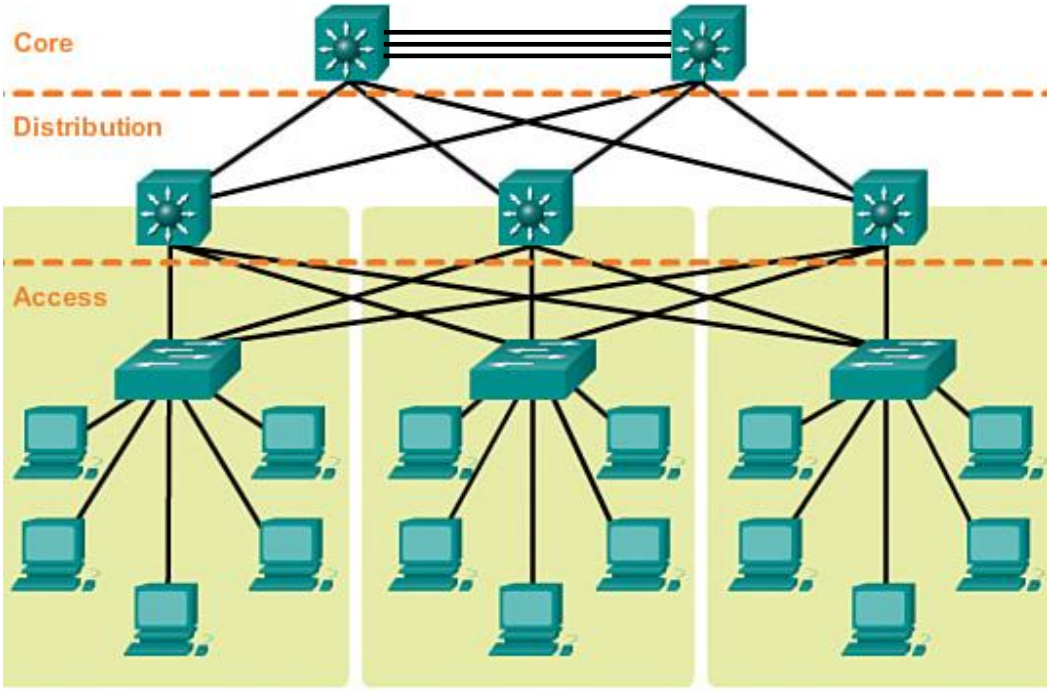

Figure 10 - Three tier hierarchical network [37]

Figure 10 shows an example network topology using these three layers. The access layer is referred to as the ‘last-mile’ part of the network, and provides connectivity and access to the customers by connecting to customer premises equipment (CPE). The distribution layer aggregates traffic from the access layer for transport on the core layer, provides policy-based connectivity, and is the boundary between the core and access layers. As it connects resources in the city or town it is located in, the distribution layer is also referred to as the metropolitan area network. Finally, the core layer connects the distribution network layers from different geographical areas, typically over long distances [38].

3.1.2. Core Network Topology

An example of a typical carrier core network topology in the United States is presented below in Figure 11. An interesting characteristic to note is the low number of links per node. A survey of 11 service provider core networks found the number of nodes ranged from 16 to 600, and the average node-to-link ratio was 1:2.3 [39].

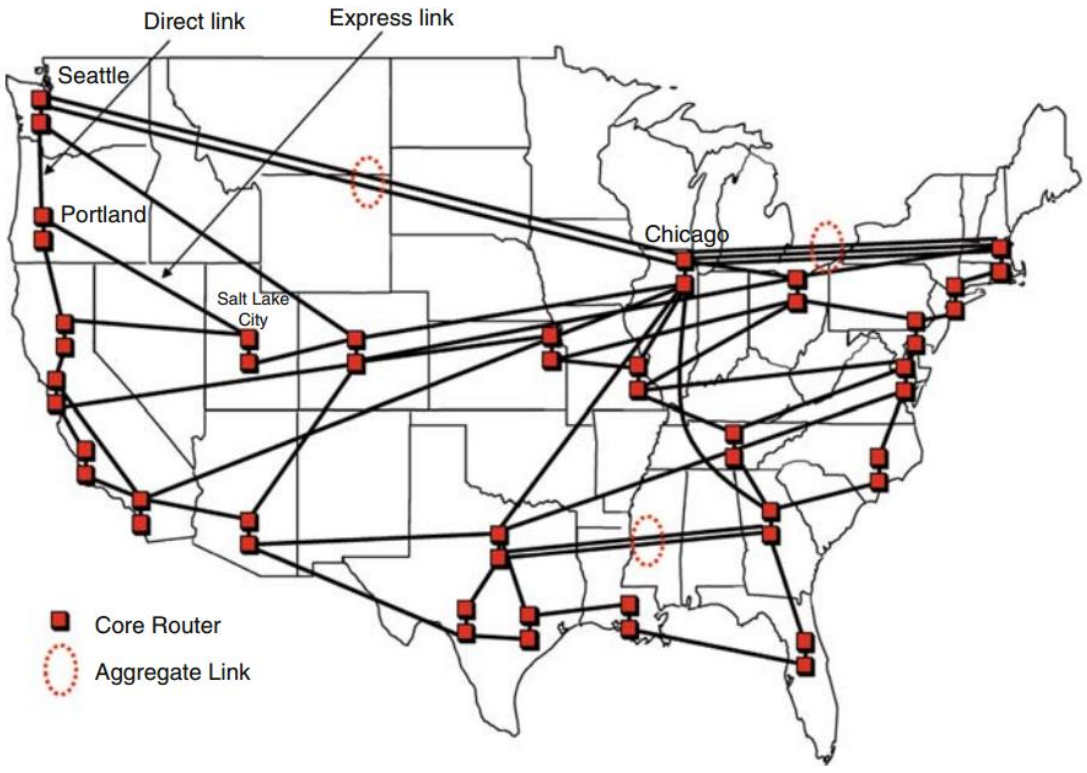

Figure 11 - Example core network for a US carrier [38]

A second important characteristic is the use of multiple links between two nodes, forming an aggregate link. In addition to single link and node failures, the fast reroute solution must take into account the various failure scenarios for these aggregate links. If a single link in the aggregate should fail, the optimum back-up path will likely use another link in the aggregate.

However, it is likely that the entire aggregate could fail at once (which is still classified as a single failure) due to a line cut or node failure. The FRR solution must take different actions for these different scenarios, and not simply avoid the entire aggregate if one link in the aggregate fails, as this would be inefficient.

3.1.3. Core Network Requirements

The function of the core network influences its design and requirements. The main requirements of the core layer are as follows:

- Provide high-speed switching [37]
- Provide reliability and fault tolerance [37]
- Provide traffic engineering functions [38]
- Support multiple types of traffic [40]

High-speed switching is required to switch large amounts of traffic quickly, while minimising the cost of the networking hardware. While modern routers can implement fast IP header lookups and path calculations in hardware, these devices are expensive. A label switching protocol like MPLS provides a more efficient solution by performing the path calculation once, at the edge of the network, and assigning a short label accordingly. Cheaper switches, as opposed to routers, then switch the traffic in the core of the network based on this label.

As mentioned previously, for the core network to be considered carrier-grade, the fault tolerance requirement is that traffic must be redirected away from a failure within 50 ms of the failure being detected [11]. A specific example of this specification can be found in the standard for MPLS Tranport Profile [29].

Carrier networks require traffic engineering functions in order to utilise the network resources efficiently and cost-effectively. For complex networks, this requires the use of a routing protocol that can allocate resources to specific traffic flows, as well as define explicit routes for traffic flows [38].

Finally, the network should support multiple types of traffic, such as voice, data, and multimedia. This allows consolidation of different networks, like the public switched telephone network (PSTN) and the IP network, into one network, reducing costs [40].

3.2. Fast Reroute Requirements and Design Considerations

Many fast reroute solutions consider a failure of a single individual link or node. However, given that core network topologies will likely contain multiple links between nodes, which could fail independently or collectively, a carrier FRR solution should also provide support for these link aggregates. The three types of failures that must be protected against are therefore:

- Failure of a single link within an aggregate
- Failure of all direct links between two nodes
- Failure of a node

Furthermore, due to the specific requirements of the core network, a fast reroute solution aimed at the core network has different requirements to a restoration approach used for a general network. Working directly from the four main requirements of carrier networks, the following requirements of the fast restoration approach can be identified:

- Support a label-switching routing protocol
- Redirect traffic within 50 ms of a failure being detected
- Use back-up paths that are aware of the traffic engineering constraints on the network
- Protect multiple types of traffic (e.g. IP and non-IP)

In addition to these requirements, there are other important design considerations. Firstly, the length of the back-up paths should be optimized, without increasing the length of the primary path, to maximise the efficiency of the network [35]. Secondly, the number of additional flow rules required should be minimised, as switch memory can be limited and quite expensive. The solution should also be automatic, as manual configuration is expensive and error-prone. Finally, the solution should not require the SDN control plane to be significantly overdesigned compared to what is required for normal operation. This is with respect to the number of control plane instances, as well as the minimum controller-switch latency.

3.3. Design of the Proposed Fast Reroute Algorithm

The first design decision that must be made is whether a reactive or proactive approach is used. Given that meeting the 50 ms recovery requirement with a reactive approach is difficult in emulated networks [8], it is unlikely to be feasible in a large-scale, geographically distributed carrier network. Therefore, a proactive approach was chosen, with back-up routes installed into the switches before a failure occurs.

The second design decision was the selection of a label-switched routing protocol for the back-up paths. MPLS appears to be the most common choice for carrier networks, but MPLS FRR is not ideal, as it requires the provisioning of back-up tunnels. Segment routing was selected instead, as it can efficiently create back-up tunnels on demand by encoding path information in the packet header. Segment routing keeps many of the advantages of MPLS, as it uses the MPLS data plane without modification. This also means that primary paths could be routed with IP, MPLS, or another protocol, and segment routing only applied to the back-up paths.

Management of link aggregates and failures within the bundles is a common problem, and can be performed by routers, using the link aggregation control protocol (LACP). However, to provide a more general, OpenFlow-only solution that will work with switches that do not support LACP, this function was implemented in the FRR application. The aim was to group links within the aggregate together, and present the aggregate to the rest of the application as a single link. This can be performed by adding each link to an OpenFlow select group, and using the port monitoring feature to implement fast failover between links within the aggregate. A failure of a single link within the aggregate will not be visible to the rest of the application, while a failure of the entire aggregate will present as a single link failure.

Finally, the calculation of the link and node protecting back-up routes is critical to the solution's ability to efficiently protect against any single link or node failure. Instead of using loop free alternates, the post-convergence path was chosen as the back-up path. Segment routing makes this easier to accomplish than in earlier protocols like IP or MPLS, as it allows the path to be explicitly routed without pre-provisioning tunnels or signalling other nodes [41]. Using the post-convergence path allows for the traffic engineering constraints on the network to be factored into the back-up path calculation, which helps ensure that the back-up paths meet the network requirements.

The algorithm to calculate the post-convergence path therefore must take in to account two different failures: single link failures, and single node failures. However, the switch can only monitor local interfaces to determine which rules should be used, so it is not able to independently determine if only a single link has failed, or if the node has failed.

The algorithm therefore assumes the failure is a single link failure, and acts accordingly. The node that detects the link failure cannot independently determine if the node has failed, as this requires information from two or more nodes. If the post-convergence path uses the node that

has possibly failed, the back-up rule attaches a tag to the packets indicating the link failure. Switches along the repair path use the presence of this tag to select the node protecting rules as the back-up rules, with the same primary rules. If another link has failed, it is assumed that a node failure has occurred, and the node protecting back-up path will be used.

To calculate the forwarding matrix of primary and back-up rules, the basic framework of the algorithm proposed by van Adrichem et al. [35] was used, with several modifications. The main modification is that the entire back-up path is returned, rather than just an alternative next hop. Also, the link protecting and node protecting paths are calculated in the same iteration through the network, rather than two consecutive iterations.

The proposed algorithm to calculate the forwarding matrix *fw*, as well as the matrices for the back-up rules, is presented below. The network topology information is stored in a directed graph *G* with a set *N* of |*N*| nodes and a set *L* of |*L*| links. Step 1 describes the calculation of the primary rules, while the rest of the steps describe the back-up rule calculation. Steps 5, 9, and 11 relate specifically to protection against single link failures. Steps 6, 10, and 12 relate to node protecting rules.

In addition to the primary forwarding matrix *fw*, the algorithm outputs include the back-up forwarding matrices *link_fw*, and *node_fw*. The generation of Open Flow forwarding rules from these forwarding matrices is described in the next chapter.

Algorithm 1: Link and node protection

Input: Network graph $G(N, L)$

Outputs: Primary forwarding rule matrix *fw*
Link-protecting backup rule matrix *link_fw*
Node-protecting backup rule matrix *node_fw*

1. set fw to all-to-all shortest paths matrix
2. set G_{copy_l} and G_{copy_n} to copies of G
3. for each node $n \in N$:
4. for each neighbouring node m:
5. remove link (n, m) from G_{copy_l}
6. remove node m from G_{copy_n}
7. select the set of destinations $\{d\}$ from N where $nextNode = m$
8. for each node d in $\{d\}$:
9. compute shortest path from n to d using G_{copy_l} and add to $link\_fw$

10. compute shortest path from n to d using G_{copy_n} and add to $node\_fw$
11. restore link (n, m) to G_{copy_l}
12. restore node m to G_{copy_n}

The compilation process of calculating the OpenFlow forwarding rules and groups from this forwarding matrix was also modified from [35] to specify the back-up path in the header of the affected traffic, and not by installing additional rules in switches along the back-up path.

The back-up path is installed in the header of the packet by attaching the minimal stack of labels that will ensure other switches route the traffic along the back-up path, using the primary rules, as they are not aware of the failure.

In IP FRR solutions, the P-space of a switch is the set of nodes that are reachable from the switch using the pre-failure forwarding rules, without traversing the failed link or node. The Q-space of the destination switch is similar, but refers to the switches that can reach the destination switch without traversing the failure.

For this work, the concept of P and Q nodes is narrowed to nodes that are reachable from, or can reach, a particular node by using the same path as the back-up path. In other words, it is not sufficient for the node to be reachable using pre-failure rules, but the post-convergence back-up path must be followed, in order to respect the traffic engineering constraints of the network.

Therefore, to find the label stack for the back-up path, the closest Q node and furthest P node must be found. In cases where these are the same node, only one label will be required. In cases where this single node is a neighbour of the repair node, no labels will be required. Otherwise, the nodes from the P node up to, and including, the Q node will make up the label stack.

3.4. Functional System Architecture

With the design of the fast reroute algorithm described, it is now possible to describe the rest of the system architecture used to implement and validate the proposed algorithm. The system architecture is comprised of the three planes in SDN: the network layer, the network operating system, and the applications layer. This functional architecture is shown in Figure 12. The function of each layer will now be described in a bottom-up approach.

Firstly, the network infrastructure consists of hosts to generate traffic, forwarding devices to switch traffic, and links to connect hosts and switches. The main functional requirement is that

the switches are OpenFlow 1.3 switches.

Secondly, a network operating system is required to provide network abstractions and network topology information to the application. The NOS must communicate with the switches, discover the links in the network, and install the flows and groups calculated by the application.

Finally, the application layer consists of the forwarding logic and protection mechanism. The topology must be retrieved from the NOS, and stored in a suitable graph format for the forwarding matrix algorithm. During this process, link aggregates must be abstracted and represented by a single link. Switches form the nodes in the graph, and links form the edges.

Next, the forwarding matrix algorithm finds all-to-all primary, link protecting, and node protecting paths for the topology. The output of this algorithm is a set of paths. The back-up path optimization function then finds the minimal set of labels that must be attached to a packet to route it along the post-convergence path, given the primary forwarding rules in other switches.

The flow compiler then uses the primary forwarding rule set, along with the optimized label stack for back-up routes, to calculate the flow rules and groups that will implement this behaviour. The flow compiler must be aware of link aggregates, and install flow rules that efficiently protect against failures within the aggregate. Using the northbound API, it must then instruct the NOS to install these into the switches.

Finally, the IP learning function is responsible for learning the location of the IP hosts, as well as installing the corresponding ingress and egress rules.

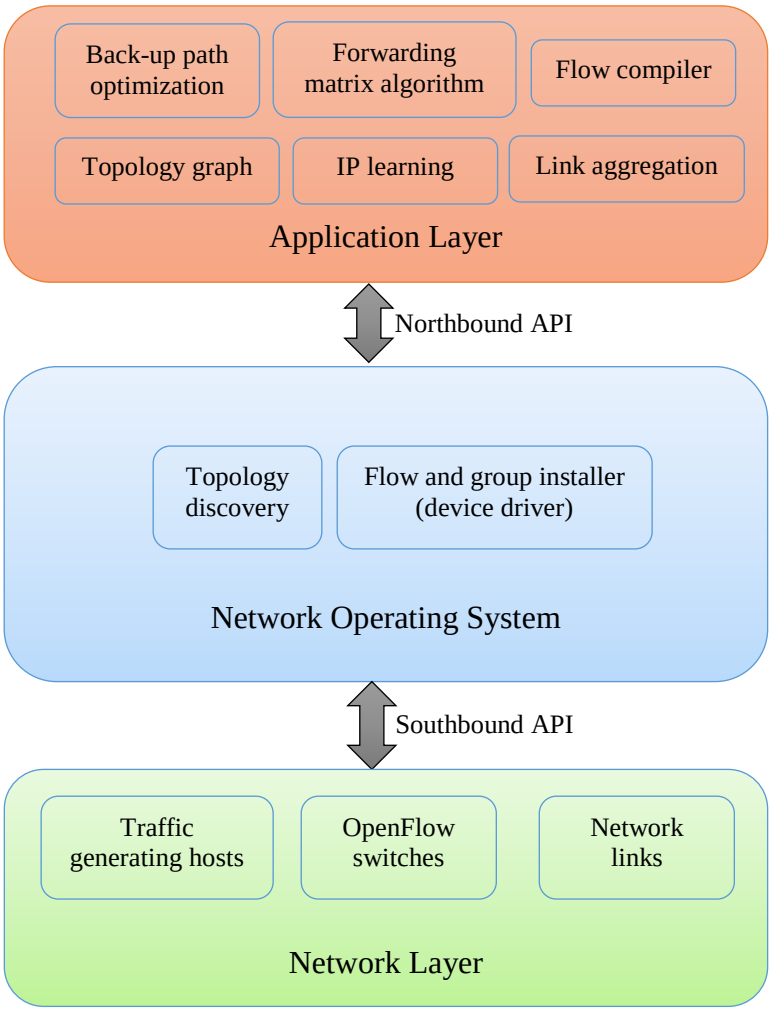

Figure 12 – Functional architecture of the proposed system

3.5. Chapter Discussions

This chapter first presented the general architecture of a carrier network, and then focused more specifically on the topology and requirements of the core of the network. From this discussion, the requirements of the proposed FRR solution were established. Finally, the design of a suitable FRR strategy to meet these requirements was presented. The algorithm used to calculate the back-up paths and label stack required was presented and explained.

The following chapter will provide more specific implementation details on the FRR solution. In particular, the implementation of the algorithm within a network application is presented, as well as the network testbed used to verify the solution.

Chapter 4

Implementation of a Carrier Grade SDN FRR Solution and Evaluation Platform

The previous chapter presented the requirements of carrier core networks, and then presented the design of the algorithm to calculate the back-up paths, as well as the algorithm to calculate the label stack for the back-up path. This section describes the implementation of the proposed FRR solution, as well as the implementation of a current reactive solution, which is used to evaluate the performance of current reactive approaches. A bottom-up approach is followed, describing the implementation of each SDN layer and the functions in the layer. The process of network setup and configuration will then be described, and finally, some of the evaluation tools used will be presented.

4.1. Network Layer

As shown in Figure 12, the three main components of the network layer are the OpenFlow switches, the network links, and the hosts used to generate network traffic. The OpenFlow switches will be discussed first, followed by the overall network environment, which will also cover the hosts and links used.

4.1.1. OpenFlow Switch

Two categories of switches exist: software switches and hardware switches. Software switches were suitable for use in this work, as the main focus was on the functionality of the controller, and not on switch performance. Software switches allow rapid, comprehensive testing of a control algorithm, and since the southbound interface to the controller (OpenFlow) is standardized, the functionality for both software and hardware switches should be the same.

Open vSwitch (OVS) [42] was selected as the OpenFlow software switch, as it is widely used, has high performance, and supports the protocols needed, such as OpenFlow 1.3, BFD, and MPLS.

Per-link BFD is used to monitor the liveness of ports, and report link failures. This liveness information is then used in the selection of the forwarding actions.

OVS by default only supports pushing a maximum of 3 MPLS labels onto a packet. This limitation would likely not cause issues in a normal MPLS deployment, but does limit a segment routing deployment. In particular, it limited the proposed FRR application, as more

than three labels were sometimes required. Since OVS is open-source, a patch was found to support more MPLS labels and documented by the community for a previous version of OVS [43]. However, it has not been incorporated into the main codebase. Therefore, it was necessary to port the patch to the latest version of OVS, and re-compile the switch. Further details of the modification can be found in Appendix A.

4.1.2. Network Environment

Two network environments were used in this project. The first, Mininet, consisted of a number of virtual switches and virtual links within a single Linux virtual machine (VM). The second environment, GENI [44], consisted of geographically distributed infrastructure and was used to test the proposed FRR application using distributed virtual switches and physical links.

Mininet

Mininet is a network emulator that uses lightweight process-based virtualization to establish a network of virtual switches and virtual Ethernet links within a single Linux VM [45]. The main design goals of Mininet were to provide a network prototyping platform that is flexible, scalable, and realistic.

Mininet can be viewed as a network orchestrator. The topology is programmatically defined in a Python script, using the Mininet API. Mininet then launches OVS instances for each requested switch, and connects them with virtual Ethernet links. Each OVS instance establishes its own connection to the controller, which is separate from Mininet. Process-based virtualization is then used to launch virtual hosts in isolated network namespaces.

The Mininet API also provides useful functions for debugging and testing, such as administratively bringing links or interfaces down, or testing all-to-all connectivity. These functions can be used to automate testing a link and node failures within the network.

These properties make Mininet ideal for development purposes, as well as verifying the functionality of the control algorithm using many different network topologies, with many nodes.

The Amazon Elastic Cloud Compute (EC2) platform was used to host these virtual machines, as it allowed for thorough testing of the reactive application's performance, without being restricted by computational resources. This was important to ensure that the reactive application's response time was not being limited by the available resources.

GENI

GENI (Global Environment for Network Innovations) provides computational and network resources which are geographically distributed across the United States [44]. Different sites can be connected with layer 2 (i.e. Ethernet) protocols, which allows for testing of non-IP protocols, like MPLS, between sites.

The same OVS switches were used as in the Mininet environment, but each switch was in a separate Linux VM. Hosts were also implemented as separate Linux VMs.

The GENI testbed allows for verifying that the application can also configure a physical network, as it uses physical network interfaces and physical links. The GENI infrastructure is shared with other researches, and well utilized, which limits the amount of available resources. In particular, the number of links that could be provisioned between different sites was limited. This limited the number of nodes that could be provisioned, and restricted testing to simple topologies.

4.2. Network Operating System

As mentioned in section 2.1.3, there are many different NOSes, with varying levels of performance and functionality. ONOS is a particularly notable controller, as it was specifically developed for the WAN, whereas many other controllers are aimed at the data centre. It has many important features necessary for core networks, such as high performance, control plane resilience, and scalability.

Given the many benefits of ONOS, it was initially chosen for this work, for both the reactive and proactive solutions. The performance and scalability of the ONOS control plane make it particularly well-suited for a reactive approach, which has already been implemented by the ONOS team.

However, it was found that the device drivers in ONOS do not yet support failover groups. This makes it unsuitable for a proactive FRR control strategy. Implementing this functionality was outside the scope of this work, so an alternative NOS, RYU was chosen to implement the proposed proactive strategy, as it does support fast-failover groups.

While RYU is not distributed, and does have some drawbacks in terms of performance and scalability, this is less critical for the proactive approach, since the forwarding rules do not have to be recalculated within 50 ms. RYU is a popular controller in network research literature, possibly due to it allowing for rapid development, making it ideal for developing a prototype

application.

4.3. Network Applications

This section describes the network applications in more detail, which contain the network intelligence used to calculate the forwarding rules. The first application, which is an off-the-shelf application developed for ONOS, implements a reactive approach to data plane failures. The second application implements the proposed proactive FRR solution.

4.3.1. Reactive Application

The intent framework within ONOS allows applications to specify the desired network behaviour in terms of policy, rather than mechanism [46]. This is a powerful framework, as it allows the NOS to handle flow compilation and recompilation in the event of a failure, rather than the application needing to implement these functions.

The intent framework documentation states the following with regards to data plane failures: “A loss of throughput or connectivity may impact the viability of a successfully compiled and installed intent. In this case, the framework will attempt to recompile the intent, and if an alternate approach is available, its installation will be attempted” [46]. This indicates that a reactive approach is used.

ONOS has published extensive performance measurements regarding the intent framework [31]. The most interesting result is the reroute latency, which is heavily influenced by the number of controller instances. For tests with more than one controller instance, the reroute latency is several orders of magnitude larger than the reroute latency with a single controller instance. This is likely due to the extra steps of controller synchronization, and shows that adding controller instances does not always improve performance. These results are shown in Figure 13, where the batch size refers to the number of installed intents, and the scale refers to the number of controller instances. All tests were run on a Dual XeonE5-2670 v2 2.5GHz server, with 64GB of RAM and 512 GB of solid state storage. The tested network consisted of a simple network with the primary path consisting of 6 hops, and the back-up path consisting of 7 hops [47].

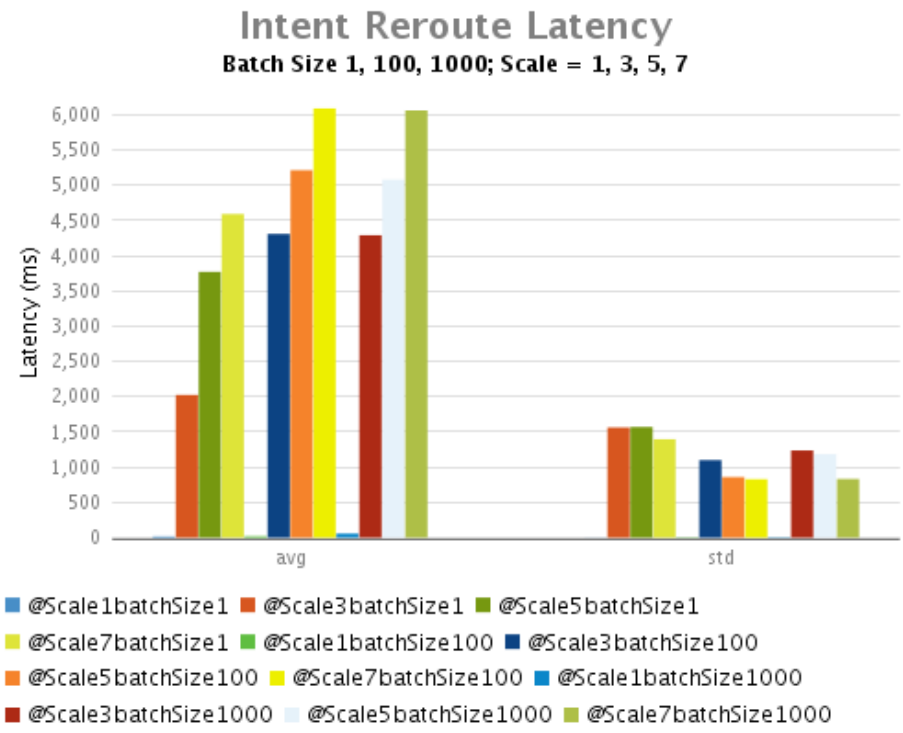

Figure 13 - Intent reroute latencies for different configurations [31]

These results clearly indicate that a sub 50 ms restoration time will only be possible with a single ONOS instance. To achieve redundancy in the control plane, the synchronization overhead would need to be eliminated, possibly by using one active controller, and several standby controllers, instead of several active controllers. For these tests, a single instance was used, to determine the best-case reroute latency with larger networks than the one tested by the ONOS performance tests.

This intent framework was used to install host-to-host connectivity intents, using the REST API of ONOS. The intent framework does accept constraints, which means that it could support traffic engineering. However, this was out of scope for this work, and the focus was on the performance of the flow recompilation in the event of a failure.

A Python application was written to request all known hosts in the network, and request host-to-host intents to be created between each host pair. This can be summarised by Algorithm 2 as shown below. The REST northbound API was used to communicate with ONOS.

Algorithm 2: Host-to-host intent request

1. Request a list of hosts
2. Calculate all possible pairs of hosts
3. Request host-to-host intent for each pair of hosts

4.3.2. Proactive FRR Application

The proactive FRR application was implemented as a RYU network application written in Python, using the RYU northbound API. The application developed by Niels et al. in [35] was used as a framework for the application, but all major components were rewritten to implement the proposed solution. The application will first be described by providing an overview of its operation, as shown in Figure 14, and then providing more specific implementation details on the components of the application.

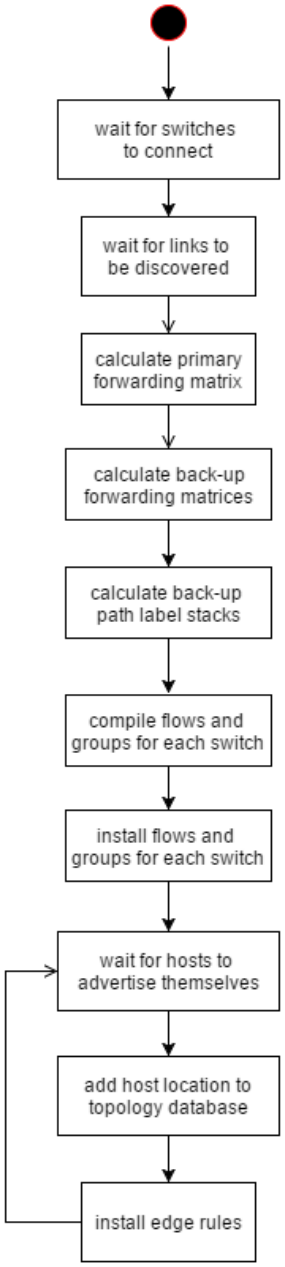

Figure 14 – Process diagram of the proactive FRR network application

When the application starts, it subscribes to notifications about topology changes and incoming packets. These notifications are generated by the NOS. Once the application has been notified of switches that have connected to the controller, and the links that have been discovered by the controller, it then waits for the topology to settle.

Once the topology has settled, and a complete view of the network has been obtained, the primary and back-up forwarding matrices are calculated. The back-up paths are converted into the necessary label stacks, and OpenFlow flow rules and fast failover groups are compiled. These are then installed on the switches, and the network is ready for hosts to begin advertising their location.

Hosts advertise their location by broadcasting address resolution protocol (ARP) messages in an attempt to discover another host. These ARP messages are received by the application, which captures the host location, and installs the necessary ingress and egress flow rules in edge switches. These flow rules match on the IP of the host, and attach the label corresponding to the edge switch the host is connected to.

Once all the hosts have been discovered, and the corresponding edge rules installed, traffic is able to flow between any two hosts without any controller involvement. Additionally, if a link or node failure occurs, the switches automatically select the back-up rules and redirect the traffic around the failure without any controller involvement.

To describe the implementation of the application in more detail, each major process in Figure 14 will now be described in more detail.

Link and Node Discovery

Link and node discovery is handled by the NOS, but the application must store this information in a meaningful way. The topology is stored as a graph, with each switch in the network as a node, and the connecting links as edges. Link aggregates are abstracted in this graph by only adding a single link when multiple parallel links connect a pair of switches. This was necessary as the graph library does not support link aggregates. The switches are also stored in a separate database, which contains full information about the link aggregates.

Primary forwarding matrix

To calculate the shortest paths in the network, the graph library NetworkX was used [48]. Although NetworkX provides a function to compute the all-to-all shortest path matrix, this function was not ideal. Due to the presence of multiple equal cost paths in the tested networks,

the calculated path from node A to node B often traversed different nodes as the path from node B to node A.

This made it difficult to efficiently calculate the label stack for the back-up path, so the forwarding module was designed to guarantee that these two paths were the reverse of each other. This also avoids unnecessary computations, as the path between two nodes is only calculated once, instead of once for each direction. The algorithm was implemented as shown below in Algorithm 3, where the shortest path from n to m was calculated using the NetworkX library.

Algorithm 3: Primary forwarding matrix

Input: Network graph $G(N, L)$

Outputs: Primary forwarding rule matrix *fw*

1. calculate set of all node pairs in G
2. for each pair (n, m) of nodes:
3. calculate shortest path p from n to m
4. set $fw[n][m] = p$
5. set q equal to the reverse of p
6. set $fw[m][n] = q$

Back-up forwarding matrix

The calculation of the two back-up forwarding matrices was implemented as described in Algorithm 1. The output of this function was two matrices, containing the paths for link protection and node protection. Again, the NetworkX graph library was used to calculate the shortest paths between a node and all other nodes in the network.

However, these paths are not useful unless installed into the switches in a functional and efficient manner, which is the purpose of the following two modules.

Back-up path label stack

Using the segment routing protocol, it is possible to encode an explicit path into the traffic header, forcing it to follow a path different to the shortest path. This functionality was exploited to encode the repair path into the packet headers, rather than installing additional flow rules in switches along the route.

To calculate the shortest label stack to direct the traffic along the repair path, the P and Q

nodes along the path must be found, as described in section 3.3. The entire set of P and Q node refers to the nodes that are reachable from the repair node and the destination node, respectively, using the pre-failure forwarding rules, and without traversing the failed resource. Given that the post-convergence path is used in this instance, to take traffic engineering constraints into account, the only P and Q nodes of relevance are the ones along the repair path that would forward along the repair path.

To find the closest Q node, the destination node's forwarding matrix was compared with the repair path. Due to the way in which the primary forwarding rules were calculated, it was guaranteed that the node on the repair path would use the reverse of the route the destination node would.

Starting with the node closest to the destination node, each node in the repair path was checked to determine if it was a Q node. As soon as a non-Q node was found, i.e. one that would by default take a different route to the destination than the repair path specified, the search was stopped and the closest Q node stored.

Next, the furthest P node was found by checking each node in the repair path against the default forwarding rules of the repair node's neighbour, since the repair node could directly forward the packet to this neighbour. The search starts from the nearest node, and stops either when a non-P node is found, or the closest Q node has been reached. If the closest Q node has been reached, and is also a P node, this means that the P and Q spaces intersect.

Finally, the label stack was generated using the furthest P node and closest Q node. If the closest Q node is a neighbour of the repair node, no label is required. If the P and Q nodes are the same node, but not a neighbour of the repair node, the label stack consists of a single label. Otherwise, the label stack consists of labels corresponding to each node in between the last P node and first Q node.

Algorithm 4: Label stack

Inputs: Back-up path bp, primary forwarding matrices *fw_source* and *fw_destination*

Outputs: Label stack l for back-up path bp

1. set s to index of source node
2. set d to index of destination node
3. set n to index of source node's neighbor in bp
4. for i in range $[1, length(bp)]$:
5. set bp_{test} to $bp[d - i : d]$

6. reverse bp_{test}
7. if bp_{test} is in $fw_{destination}$:
8. set $q_{closest}$ to $d - i$
9. else:
10. Non-Q node found, stop search
11. for i in range $[1, q_{closest}]$:
12. set bp_{test} to $bp[n: n + i]$
13. if bp_{test} is in fw_{source}:
14. set $p_{furthest}$ to $n + i$
15. else:
16. Non-P node found, stop search
17. if $p_{furthest}$ and $q_{closest}$ are equal, and equal to n:
18. return an empty label stack l
19. else if $p_{furthest}$ and $q_{closest}$ are equal, and not equal to n:
20. return l = $bp[p_{furthest}]$
21. else:
22. for i in range $[p_{furthest}, q_{closest}]$:
23. add $bp[i]$ to label stack l
24. return l

Forwarding flow rules and groups

The installation of the flow rules and groups takes place once the topology has been discovered, and the primary and back-up forwarding matrices calculated. These flow rules and groups fully define the behaviour of the switches, and must therefore implement a segment routing forwarding protocol, as well as the fast-failover actions to redirect traffic onto back-up paths.

The installation process is completed by iterating through each switch in the network, and then iterating through each entry in the forwarding matrix for that switch. Each entry in the forwarding matrix corresponds to another switch in the network, and contains the primary path, the link protecting path, and the node protecting path.

To implement the segment routing protocol, globally significant MPLS labels were assigned to each node. Locally significant adjacency labels were not assigned to specific links, since

link aggregates were viewed as a single link. Therefore, a node label corresponding to a neighbouring node, and an adjacency label corresponding to the link (or link aggregate) joining the two nodes, will be functionally equivalent.

Figure 15 below describes the process of calculating the flow rules and groups for a given switch, corresponding to a given destination. As mentioned, this process is repeated for each switch and destination combination.

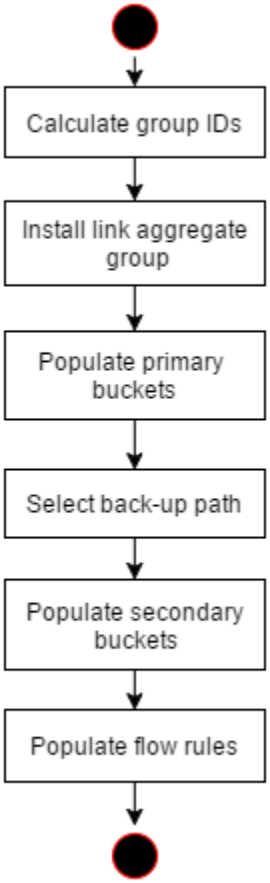

Figure 15 – Process diagram of the flow rule and group installation for a single destination

Calculating group IDs

OpenFlow group IDs can be assigned arbitrarily, but were calculated using the current switch's ID, as well as the destination node ID, so that the group can be easily referenced by other functions in the application. Two groups must be created: the pre-failure group, and the post-failure group. The pre-failure group handles all traffic that has not yet been routed around a failure, while the post-failure group handles traffic that has already been re-routed.

Installing link aggregate groups

If the next hop for any of the output actions was connected by a link aggregate, a select group

was used to implement fast failover among the links, as well as load balancing. This group was created by adding an action bucket for each port, with a watch parameter to disable the bucket if the port was disabled.

This group is only created once for each link aggregate, and re-used by any group needing to output traffic to the aggregate. The select group essentially functions as a virtual port, and will become disabled if all of its buckets become disabled.

Populating primary buckets

Within each group, the primary bucket should forward traffic along the primary path. To implement this, the port (or aggregate group) corresponding to the next hop along the primary path is set as the output action of the primary bucket within each group. The same port (or group) is set as the watch port (or group), and its liveness monitored to determine the availability of the bucket.

A special condition applies, however, if the destination is the next hop. In this case, the bucket pops this label off of the packet before forwarding it, to implement penultimate hop popping (PHP). PHP is used to avoid the destination switch having to remove its own label, and recirculate the packet back to the match table to forward the packet further (either to an attached host, or further switch).

Back-up path selection

The normal selection of back-up paths uses the link protecting back-up path for the pre-failure group, and the node protecting path for the post-failure group. If no failure has yet occurred, the link protecting path often provides sufficient protection, but if a failure has already occurred on the path, it is necessary to assume that the node has failed.

Two special conditions exist, where the node protecting path is used in the pre-failure group as well as the post-failure group. If the neighbour opposite to the failed link is contained in the label stack, the link protecting path would attach a label explicitly routing the traffic through this node. If the node has failed, this traffic would be dropped by further switches, as no protection can be applied if the top-of-stack label points to a failed node. In this case, the node protecting path is used instead, as it will work for both link and node failures.

The second special condition applies if the link protecting path passes through the potentially failed node, has a non-empty label stack, and if the node protecting path has an equal path cost. If the link protecting path attaches one or more labels, which route the traffic through a node

that has potentially failed, a node failure can cause a sup-optimal path to be followed. This is because the next node, which detects the node failure, can only calculate the shortest node protecting path to the switch indicated by the top-of-stack MPLS label, and not the shortest path to the final destination.

Since link failures are more likely than node failures [35], a shorter link protecting path is a worthwhile trade-off for this suboptimal node protecting path, since the node protecting path will less commonly be used. However, if selecting the node protecting path immediately can shorten the actual path taken in the case of a node failure, and does not lengthen the path taken in the case of a link failure, then it should be chosen. These two selection criteria are summarised in Algorithm 5 below.

Algorithm 5: Back-up path selection for pre-failure group

1. if next hop n is in link protecting label stack l_{link}:
2. use node protecting label stack l_{node}
3. else if $cost_{link} = cost_{node}$ and next hop n is in bp_{link}:
4. use node protecting label stack l_{node}
5. else:
6. use link protecting label stack l_{link}

Populating secondary buckets

The secondary bucket is used if the primary bucket is unavailable, which occurs when the primary output port (or link aggregate group) fails. The secondary bucket must redirect the traffic in such a way that it follows the back-up path selected in the previous step. In the case of an empty label stack, this only requires forwarding the traffic to a different port. In the case of a non-empty label stack, additional MPLS labels are attached to the traffic to explicitly route it. Packets are routed based on their top-of-stack MPLS, which is the last label attached. Therefore, the last label in the label stack is attached first, with the first label attached last.

Finally, the MPLS traffic class value is set to 2, to indicate to further switches that a failure has occurred along the primary path, and that the post-failure rules should be used.

Populating flow rules

Finally, the flow rules to forward incoming traffic to the correct group must be installed. Two sets of flow rules are needed: one for the pre-failure traffic, and one for the post-failure traffic. Both of these match on the MPLS label corresponding to the destination switch, and forward

the traffic to the correct group. The only difference, besides the output group, is that the pre-failure flow matches on an MPLS traffic class value of 1 (which is the default value used when attaching MPLS labels), and the post-failure flow matches on a traffic class value of 2.

Host learning and edge rule installation

Once the flow rules and groups from the previous step are installed on each switch, allowing all-to-all connectivity between switches, edge rules are needed to map incoming traffic to the correct egress switch, and label the traffic accordingly.

Since the network must support multiple protocol types, and the underlying protocol must be known when removing the last label, host-specific MPLS labels are needed in addition to the labels for the egress node. In a multi-protocol scenario, the host-specific label can indicate to the egress node the destination as well as the protocol. A more complete explanation can be found in Appendix B.

For this application, IP hosts were used to generate traffic, and were connected directly to core switches. Therefore, an IP learning function was implemented to map incoming traffic to the egress port. Two things must be known for this mapping process: the location of each IP host, and which switches are edge switches.

IP hosts broadcast ARP messages, which contain the source and destination IPs addresses, in order to discover a particular destination host. The switches are configured to encapsulate ARP packets and forward them to the controller, along with the input port. The controller uses this information to learn the location of the host, and also mark the switch as an edge switch.

Ingress flow rules match on the destination IP address, and apply both the host specific label (determined by the IP address), and the egress switch label, before forwarding to the group corresponding to the egress switch. Egress rules match on the host label, remove the label, and output the traffic to the host.

When a new host is learnt, its IP address is added to the database of known IP addresses. If the switch was not already an edge switch, it is marked as an edge switch and ingress rules for all known IP addresses are installed. Ingress rules are then installed for the new IP address on each switch in the network, except for the switch that it is connected to, which receives an egress rule. This process can be summarised by Algorithm 6 below.

Algorithm 6: IP host learning and edge rule installation

1. if incoming packet is an ARP message:
2. if IP_{src} is not in IP_{hosts}:
3. set $IP_{hosts}[IP_{src}] = (s_{src},\ s_{port})$
4. if s_{src} is not in s_{edges}:
5. add s_{src} to s_{edges}
6. for each IP in IP_{hosts}:
7. install ingress flow rule on s_{src} for IP
8. for each s in s_{edges}:
9. if s_{src} = s:
10. install egress flow rule on s for IP_{src}
11. else:
12. install ingress flow rule on s for IP_{src}

4.4. Network Topologies and Set-up

Five different network topologies were tested using the Mininet platform. To test the performance of the reactive approach implemented by ONOS, a simple 4 node network was used, as well as a larger 24 node topology more representative of a carrier core network in the United States [17]. To test the proposed proactive application, a topology with 7 switches was used to verify that the application was functioning correctly. Next, the US network with link aggregates was tested. Finally, a topology modelled after a possible carrier core network in South Africa was used. Public information from several different providers was combined to generate this model, which includes aggregate links [40] [49]

Using the GENI testbed, a geographically distributed network was established, using four sites. Each site contained a single switch, all connected to a single controller in one of the sites. This network was used to measure a realistic controller-switch latency, as well as verify that the proposed application could configure a physical network.

4.4.1. Mininet Topologies

Each of the Mininet topologies were defined in Python script, using the Mininet API to request switches, links, and hosts in a particular configuration. The modified version of OVS 2.6.0 was used for each switch. To obtain a graphical image of the topology, the graphical

topology viewer in ONOS was used.

The simple 4 node ring topology is shown in Figure 16 below. This was used to test the response time of the ONOS application, as each link failure requires a similar response, and the response time should be similar.

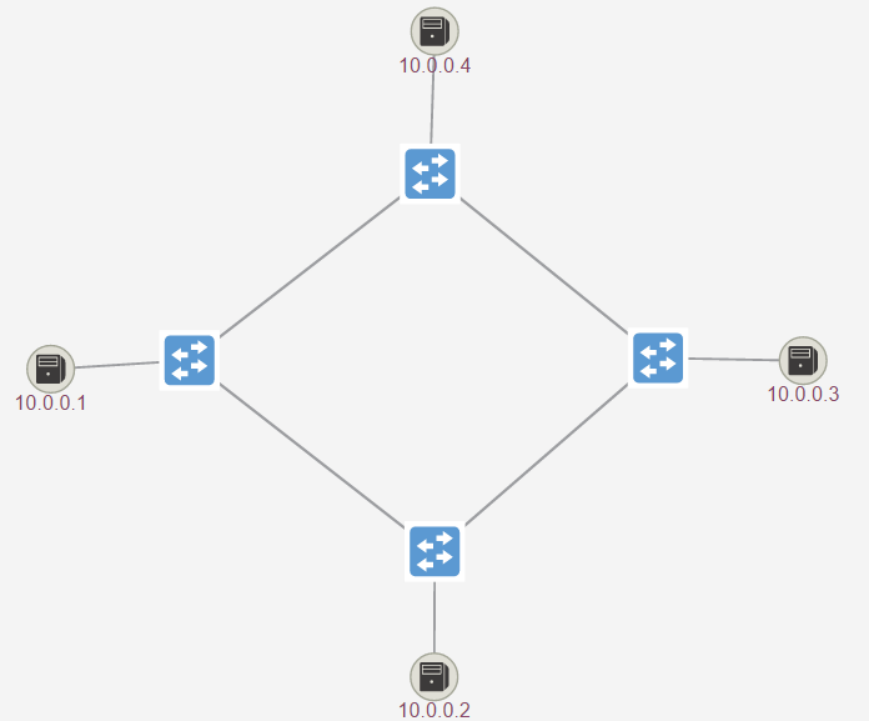

Figure 16 – Ring topology

The US carrier core topology is shown in Figure 17, and was used to test the response time of ONOS with a larger network, with more complex repair paths. Multiple host-to-host intents were also installed to test the scaling limitations of the controller.

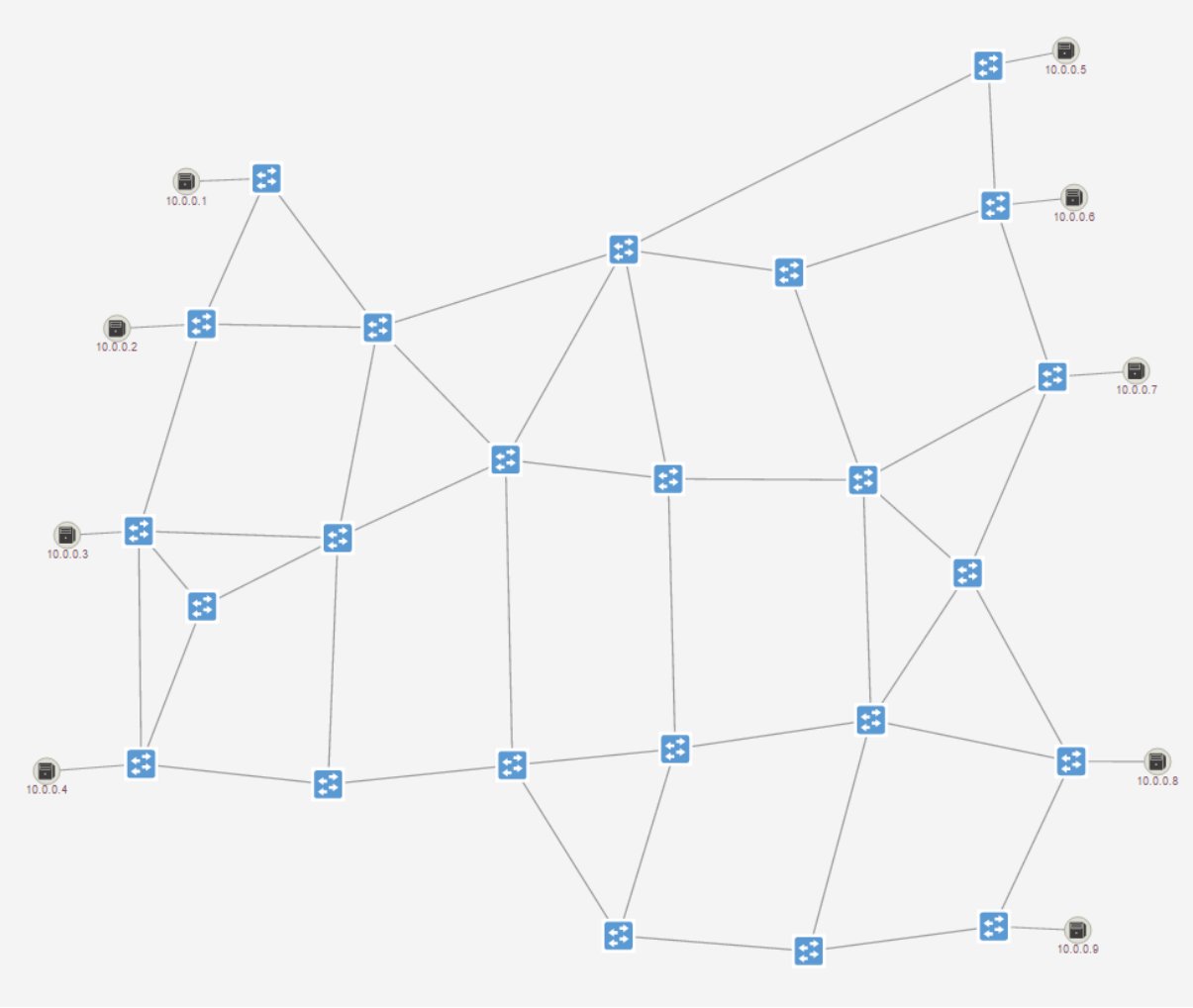

Figure 17 – US core topology

The 7 node topology that was used to verify the correct operation of the application is shown in Figure 18 below. This network was designed to test the application's response to a single failure within an aggregate, a failure of the entire aggregate, and a node failure. The correct back-up path is clearly observable in each scenario, and so the correct operation of the application is easy to verify.

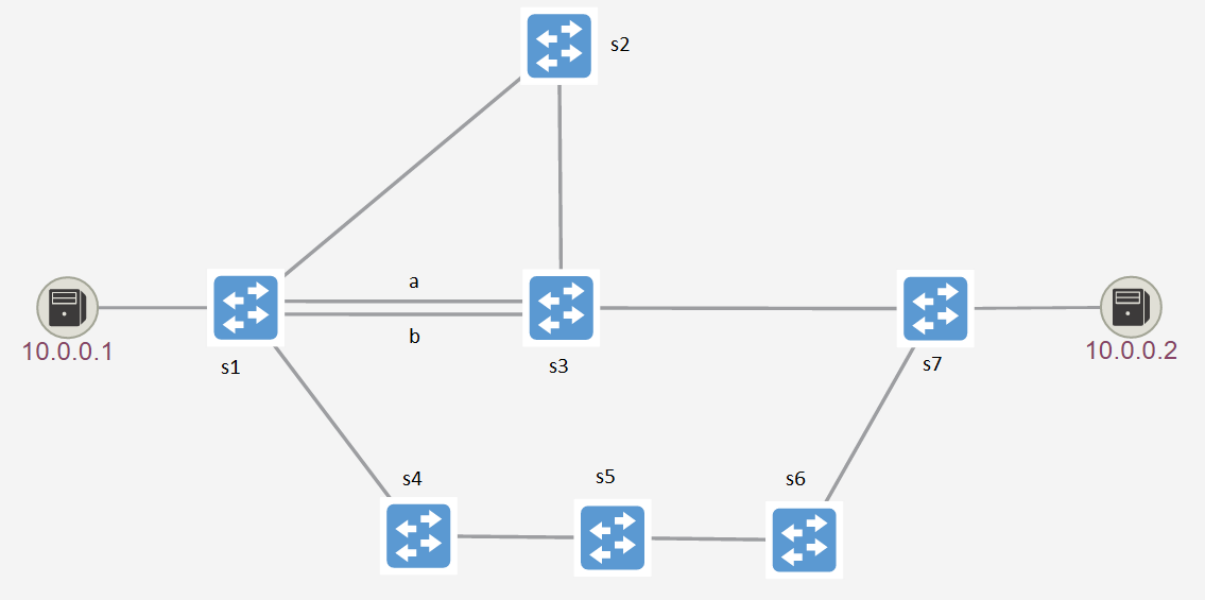

Figure 18 – 7 node network used for verification testing

The US carrier core topology with link aggregates is shown in Figure 19 below. This topology is used to test the various performance metrics in a large network with link aggregates. The SA carrier core topology is shown in Figure 20, and was used to measure the performance metrics in another possible scenario. To simplify the figure, the traffic generating hosts connected to each switch are not shown in Figure 20.

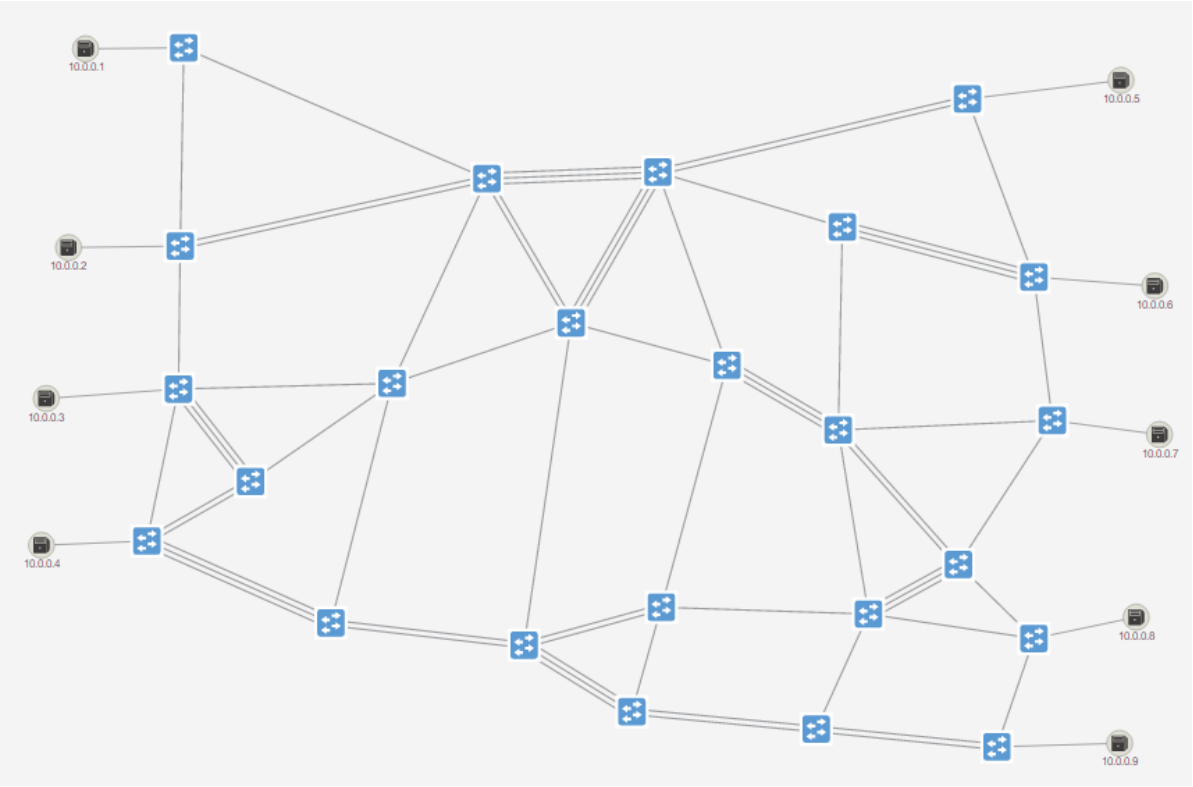

Figure 19 – US core topology with link aggregates

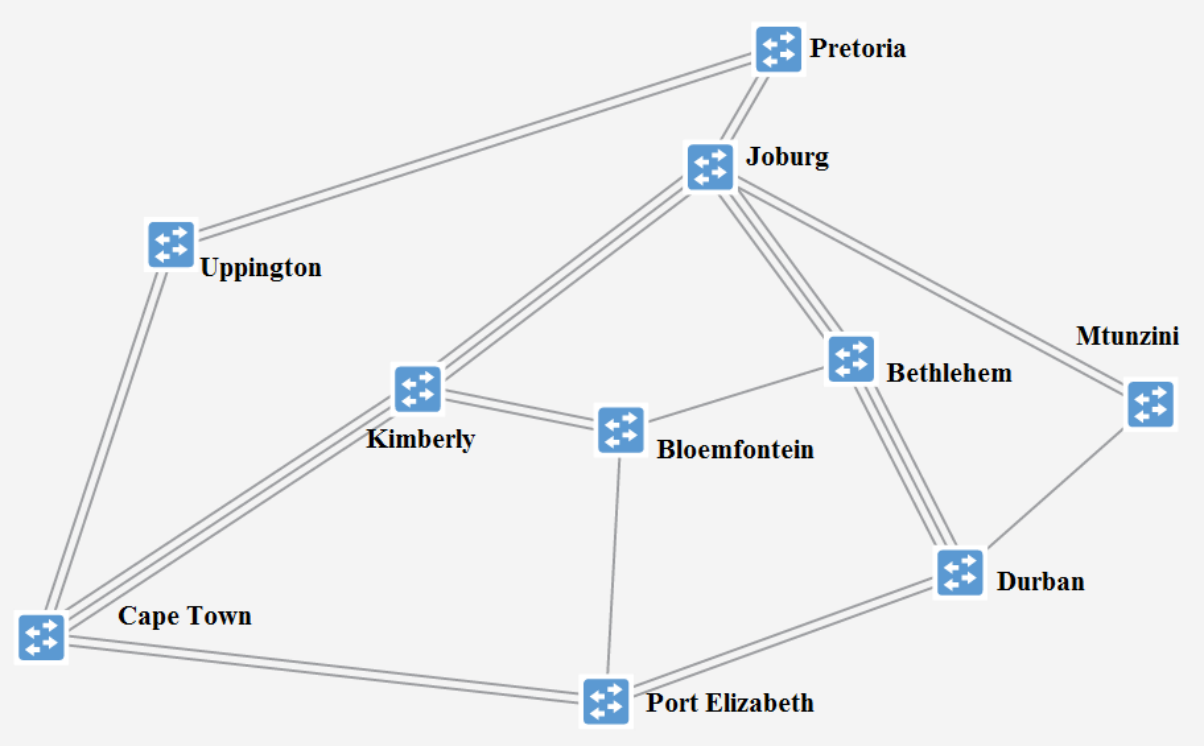

Figure 20 – South African core topology

4.4.2. GENI Topologies

GENI topologies were defined using the graphical interface provided by the GENI project. Using this interface, the user is able to request a specific configuration of Linux VMs, OVS instances and links. Geographical placement of the nodes can also be specified by placing nodes in different sites, and specifying the location of each site. There are GENI sites in many locations across the US. However, the available links between these sites are limited, placing some constraints on the geographical distribution of the network.

The ring topology used is shown in Figure 21 below. Each switch has been placed in a different site, with stitched Ethernet links connecting them. The controller is located in the same site as switch $s2$, and has direct connections to each switch. The selection of the sites was made based on the available GENI resources at the time. This network allowed for measuring the controller-switch latency, as well as for testing the proposed application in a distributed network.

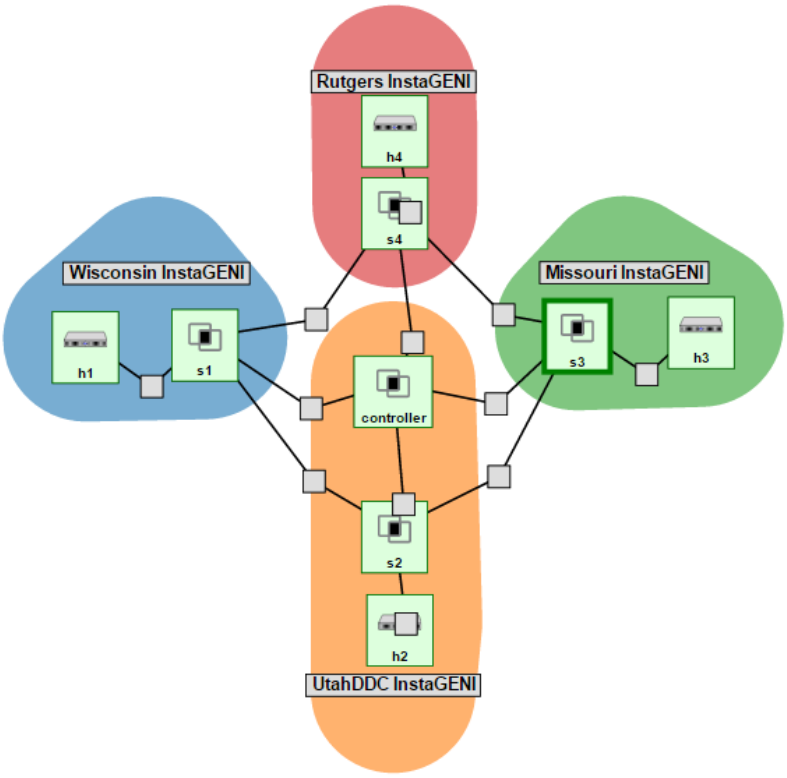

Figure 21 – Geographically distributed ring topology

The location of the four sites are marked on a map of the United States in Figure 22 below. The shortest distance by road is also shown, and the distances are recorded in Table 2. Assuming that the links will roughly follow this path, this gives an indication of the length of the path.

Figure 22 –Map of the sites used

Table 2 – Distances by road between the sites

Link	Sites	Road distance (km)
s1-s2	Wisconsin-Utah	2183
s2-s3	Utah-Missouri	1927
s3-s4	Missouri-Rutgers	1701
s4-s1	Rutgers-Wisconsin	1524
controller-s4	Utah-Rutgers	3502

4.5. Performance Metrics and Evaluation Tools

To verify the functionality and performance of the existing reactive approach, and the proposed FRR application, several different performance metrics were used. For the reactive approach, the primary metric of interest is the response time of the controller, as this determines the time between the failure being detected and the traffic being rerouted onto a new path. For the proposed proactive application, the primary metric is connectivity, and the secondary metrics are the back-up path lengths and label stack lengths. First, however, the method of producing failures will be discussed.

4.5.1. Failure Generation

Since the applications must protect against three types of failures, these three types of failures must be generated for testing purposes. Specifically, these are: failure of a single link within an aggregate; failure of a single link or entire aggregate; and failure of a single node.

For the Mininet testbed, the API provides a convenient method of administratively bringing a link down, by disabling the ports on both ends of the link. This method of administratively bringing down a link was used to remove the detection time variable from the tests.

A Python application was written to generate the three types of failures, using the Mininet API. To generate single link failures, the application iterated through each link connecting two switches in the network, failing one link at a time and then bringing it back up. This also tested failures of a single link within an aggregate.

To test failures of entire aggregates, the application found all possible pairs of switches within the network, and failed all the links between each pair of switches, if multiple direct

links existed between the pair. Each pair was tested individually, before bringing the links back up and moving on to the next pair.

To test node failures, the application iterated through each switch in the network, simultaneously bringing down every link connected to a particular switch.

This failure generating application was integrated with the following response time and connectivity tests, so that each of these metrics could be recorded for each failure.

For the GENI testbed, this API was not available, and failures had to be generated manually at each node. A link was disabled by administratively disabling the ports on both switches, and a node disabled by disabling all of its links.

4.5.2. Response Time

Measuring the traffic loss would include implementation-specific variables, such as failure detection time and switch response time. Since the response time of the controller was of primary interest, the controller-switch traffic was instead monitored to isolate the controller performance. By monitoring the control traffic at the switch, the time it takes for new flow rules to be received after a failure notification has been sent can be measured. This isolates the switch-controller latency and the controller performance from the switch performance and failure detection time.

This measurement was performed in the Mininet environment by using the failure generating application to generate failures, while capturing the traffic between the controller and all switches. The captured traffic was then analysed to determine the times between the switch notifying the controller of the failure, the arrival of the first new flow rule, and the last flow rule. A Python script was used to automate this analysis, and also record the number of flow rule modifications needed to reconfigure the network.

4.5.3. Connectivity

Connectivity is measured by verifying that each host can reach every other host in the network. This ensures that any traffic entering the network will successfully reach its destination. This is primarily a metric for the proactive approach, to ensure that the back-up paths can successfully route traffic around failures without causing loops to occur, or traffic to be dropped for another reason.

In the Mininet framework, this connectivity test is requested via the Mininet API. Mininet

sequentially instructs each host to ping every other host in network. For the GENI testbed, connectivity was tested manually by connecting to each host, and pinging every other host. If no pings fail when a particular failure has been caused in the network, then the FRR solution has successfully protected traffic against that failure.

4.5.4. Back-up Path Length

Another important metric of the proposed solution is the length of the back-up path, or the cost, relative to the primary path, as this indicates how efficient the back-up strategy is. The path costs of the link and node protecting paths is important, but the actual path followed when using both link and node protection is implemented is the most important metric. This is more complex to calculate, and not necessary for controller operation, so it was calculated offline by a separate application. The FRR application recorded the primary, link protecting and node protecting path costs, as calculated by the NetworkX library, as well as the label stacks used. These recorded values were used to calculate the costs of the back-up paths followed in different scenarios.

If a link failure occurs, the back-up path cost is simply the cost of the link protecting back-up path. If only node protection is applied, the back-up path cost is calculated similarly. However, if link and node protection is applied, the actual path followed in the case of a failure of node x is not determined solely by the back-up paths installed at the first node that detects the failure. The first node (PLR_{link}) that detects a failure will assume a link failure has occurred, and apply the link protecting action. If a node x is on the link protecting path, then a second node (PLR_{node}) will detect a failure, and determine it to be a node failure, since the traffic carries a tag indicating the first failure. This second node will then apply the node protecting action, to route the traffic around the failed node towards the final destination. In this case, the back-up path cost consists of the following two costs:

- The link protecting path for the PLR_{link}- x link, from PLR_{link} to PLR_{node}
- The node protecting path for x from PLR_{node} to l

However, the addition of segment routing labels in some link protecting actions may cause a sub-optimal node protecting route to be followed. If when the traffic reaches PLR_{node}, the top-of-stack label (l) is not the label corresponding to the final destination, the PLR_{node} can not always select the optimum node protecting route to the final destination, as it only sees the top of the stack. The actual path cost is then calculated by adding the costs of the following

paths:

- The link protecting path for the PLR_{link}- x link, from PLR_{link} to PLR_{node}
- The node protecting path for x from PLR_{node} to l
- The primary path from l to the final destination, through any other nodes in the remaining label stack

To measure the efficiency of the back-up paths in different scenarios, the ratio of primary forwarding path to back-up path was used. The path costs were calculated from the first node that detects a failure. This ratio is calculated for each PLR and final destination combination, and then the ratios averaged. Three different scenarios were considered, as summarised below. In the second, where node protecting rules are applied initially, the link protecting path is used if the next hop is the final destination, as no node protecting path exists in this case.

- Link failure assumed initially, no node protection
- Node failure assumed initially, link protecting rules used if next hop is final destination
- Link failure assumed initially, node failure assumed on detection of a second failure

4.6. Chapter Discussions

This chapter described the implementation of all three layers of the SDN paradigm: the data plane, the control plane, and the application layer. The existing reactive application was briefly described, and the proposed FRR application was described in detail. The proposed FRR application implements a segment routing forwarding protocol, as well as the proposed proactive back-up rule calculation. The network topologies were then described, discussing the purpose of each topology. Finally, the performance metrics were discussed, as well as how the measurements were made.

The next chapter will present the evaluation results of the different topologies, using the different control strategies. The effect of geographical distribution on the performance of both applications will also be tested, by distributing the switches and controller using the GENI testbed.

Chapter 5

Results and Discussion

The previous chapter presented the implementation of the evaluation platform and the network applications to implement the reactive and proactive recovery methods. The performance metrics, and testing procedures were also described. This section provides the results for each of the applications, focusing on the relevant performance metrics.

5.1. Reactive Approach

For the reactive approach, the paths used after the failure are calculated by the controller, based on the new topology, and will therefore be the optimum paths for the current topology. Therefore, the primary metric for this application is the time taken by the controller to fully reconfigure the network, since this determines the amount of traffic loss experienced.

To determine the optimal response time of the controller, the effect of the computational resources of the controller was investigated. Next, the effect of an increased number of installed intents in the network was investigated, to determine the scalability of the controller. These tests were conducted using virtual machines in the Amazon EC2 platform to provide low controller-switch latency, and sufficient computational resources. The controller-switch latency is relatively constant, and below a millisecond, so it has no significant impact on the response time. Some rare exceptions occur, where the communication latency briefly increases to a few milliseconds, but the effect of these events is limited as the measured results were the average of a large number of a samples.

Each of these tests were conducted using the failure generating application to test for every single link failure in the network, while measuring the time taken by the controller to respond with the flow rule modifications for each failure (the response time). Link failures were used since this is the simplest failure, and the best-case scenario was being investigated.

Two different topologies used, shown in Figure 16 and Figure 17, each generated using Mininet, running in a t2.micro Amazon EC2 instance, with the specifications listed in Table 3 below.

Table 3 – t2.micro specifications

Instance type	Operating System	vCPUs	Memory (GiB)
t2.micro	Ubuntu 16.04.1	1	1

5.1.1. Effect of Computational Resources on Response Time

To test the performance of the controller with different computational resources, the response time to failures was measured using the same control software running on three different EC2 instance types, using only one controller instance during each test. The specifications of the instances range from the minimum recommended requirements of ONOS [50] to eight times the minimum specifications, and are listed in Table 4.

Table 4 –Compute optimized instance specifications

Instance type	Operating System	vCPUs	Memory (GiB)
c4.large	Ubuntu 16.04.1	2	3.75
c4.2xlarge	Ubuntu 16.04.1	8	15
c4.4xlarge	Ubuntu 16.04.1	16	30

A simple ring network consisting of four switches and four hosts was used for this test, shown in Figure 16. This network was used as each link failure is similar, and require similar responses from the controller. This provides a consistent set of events for each controller to respond to.

The first instance tested, the c4.large instance, serves as a baseline test, as it slightly exceeds the recommended specifications. As mentioned previously, the application measured the times between the switch notifying the controller of the failure, and the arrival of the first and last flow rule modifications. Each test iterated through all four link failures 10 times, generating 40 data points for each instance. Since the response time is dependent on the number of modifications required [31], the average number of modifications is also recorded. The measured results are shown in Table 5 below.

Table 5 –Response times for three different instances

Instance type	First flow arrival (ms)			Last flow arrival (ms)			Average number of flow rule modifications
	Average	Max.	Min.	Average	Max.	Min.	
c4.large	28,60	45,93	23,90	33,67	52,83	27,23	20,13
c4.2xlarge	25,66	35,43	23,40	28,40	38,18	25,79	25,12
c4.4xlarge	24,79	31,11	23,86	27,52	34,06	25,48	25,3

From these results, it appears that the response time of the controller can be improved by increasing the computational resources of the controller instance. Although the average number of flow rule modifications performed by the c4.2xlarge and c4.4xlarge instances were higher than the c4.large instance, the average response time was lower. This indicates an improved response time, even with an increased load.

5.1.2. Effect of the Number of Installed Intents on Response Time

A realistic carrier network will contain more than one traffic policy between two edge switches in order to connect multiple destinations beyond the edge switches, as well as to implement traffic engineering. For example, in an MPLS network, thousands of label switched paths are often needed [15].

Using the ONOS intent framework and traffic-generating hosts, this translates to multiple host-to-host intents between a pair of hosts. To test the performance of the ONOS controller with different numbers of installed intents, multiple identical intents were installed between host pairs. Although they are identical, ONOS treats each intent as a unique intent and calculates it independently.

The example carrier network without link aggregates, shown in Figure 17, was used for this test. The number of installed intents ranged from the minimum number required to connect all hosts, to 6 times this amount. For each number of intents, every possible link failure was tested, and the response times averaged. The results are presented in Table 6 below, along with the average number of flow rule modifications required to reconfigure the network.

Table 6 –Installed intent test results

No. of installed intents	First flow mod (ms)	Last flow mod (ms)	No. of flow mods
36	23,33	32,60	72,79
72	23,03	35,60	109,23
108	23,19	38,31	142,86
216	23,01	44,88	231,65

From these results, it is apparent that with an increase in the number of installed intents, the average number of flow rule modifications required increases. The time taken for the last flow rule modification to be received also increases, and therefore the time to fully reconfigure the network is increased.

For an individual failure, the number of intents that need to be recomputed is determined by the number of paths that used that particular link. The number of modifications required is a good indication of how many intents needed to be recomputed. A plot of the response time against number of flow rule modifications is shown in Figure 23 below. This clearly shows that the response time is dependent on the number of modifications, and the relationship appears to be linear. It is also clear that if more than 300 modifications are required, the response time with regards to the final modification will violate the 50 ms restoration requirement, even if all other variables are negligible. This limit of 300 flow rule modifications is not a practical restriction for a carrier network.

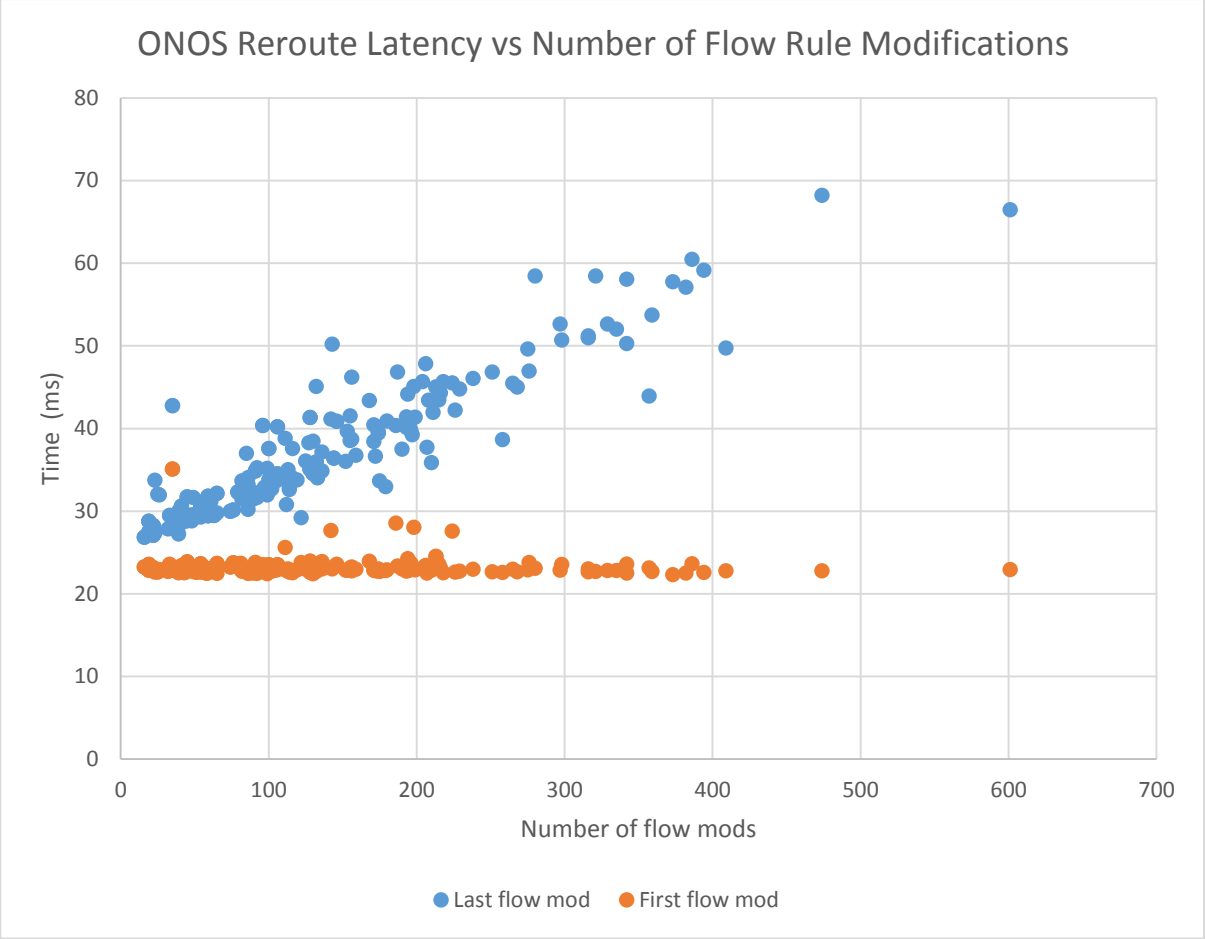

Figure 23- Response time plotted against number of flow rule modifications

5.1.3. Communication Latencies in a Geographically Distributed Testbed

The GENI platform can provide a geographically distributed network, but the virtual machines it provides have a single vCPU and 1 GB of RAM. It has been shown that the specifications of the controller influence the response time, and are required to be much higher for optimum performance. Therefore, the response time of ONOS was not tested in the GENI environment, but the communication latencies were measured. These communication latencies can simply be added to the response times in an ideal testbed, using Equation (1), to obtain the total response time in a distributed network. The round trip times (RTT) for communication between the controller and the three other sites in the distributed network in Figure 21 are shown in Table 7 below.

Table 7 –Compute optimized instance specifications

Site	RTT (ms)			Distance by road (km)
	Average	Max	Min	
Rutgers	53,85	56,39	53,51	3502
Wisconsin	50,29	53,03	49,95	2183
Missouri	24,08	24,43	23,84	1927

For the first two sites, it is observed that meeting a 50 ms restoration time is impossible with a reactive approach, as the switch-controller latency is already above this requirement, even without considering the controller response time. For the third site, the latency is lower, making a reactive approach theoretically possible. However, given that the first flow rule modification is typically only received from ONOS after 24 ms, a reactive approach is still not possible.

It is also clear that the communication latency is proportional to the geographical distance, as expected. The relationship is not linear, as it depends on the infrastructure and equipment latencies along the path, and not only the distance. However, it does show that using a reactive approach would limit the controller placement.

5.2. Proposed FRR Approach

For the proposed FRR application, it was first verified that the expected paths were installed, and the correct path costs reported, for the three different types of failures. The primary metric, connectivity, was then tested to verify the coverage against failures. The back-up path length and number of label stacks applied were then measured, to determine the efficiency of the paths in terms of length and traffic overhead. Finally, functionality in the GENI platform is also verified. This shows that the application can configure a geographically distributed network that uses physical interfaces and links. Traffic loss was not measured, as it depends heavily on implementation specific details, such as failure detection time. The important metric is the time taken to switch over to the back-up rules once the failure has been detected. This time has already been shown to be well below 50 ms, using the same fast failover groups [17]. Therefore, if the FRR application provides full protection against failures, it is guaranteed to meet the 50 ms restoration requirement for any single failure.

5.2.1. Path Verification

To verify that the back-up paths operated as expected, a simple network was used to verify the actual paths taken in the event of three different failures. This network, shown in Figure 24, consists of 7 switches, two traffic-generating hosts, 9 single links, and a link aggregate. Each single link, as well as the link aggregate, has a path cost of 1. The path cost is measured from the point of local repair (PLR) to the destination switch (and not the destination host).

Three failures were tested, namely:

- Failure of the active link within the s1-s3 link aggregate
- Failure of the entire s1-s3 aggregate
- Failure of node s3

In Figure 24, the primary path from host 10.0.0.1 to host 10.0.0.2 is marked in green. Upon failure of the active link (a) within the aggregate, the back-up action performed should reroute the traffic through the remaining link b. This detour segment is marked on the figure in lighter green.

If the entire aggregate fails, node s1 should then apply the link-protecting back-up action, routing the traffic through node s2 along the yellow path.

If node s3 fails, node s1 will not be able to determine that this is the case, and should apply the link protecting back-up action, since a link failure is assumed. When the traffic reaches s2, it should apply the node-protecting action, since the traffic is tagged as already encountering a failure. The primary path from s1 to s7 is through the failed node, s3, which means that a routing label must be attached to the traffic to route it through the node protecting path s2-s1-s4-s5-s6-s7. Two equal cost paths exist between s4 and s7, so the primary path is either through s4-s1-s3-s7, or through s4-s5-s6-s7. If the path installed does go through s3, a routing label for s5 must be attached. Otherwise, a label for s4 is sufficient to route the traffic along the intended path. The selection of which label must be applied is performed when calculating the label stack.

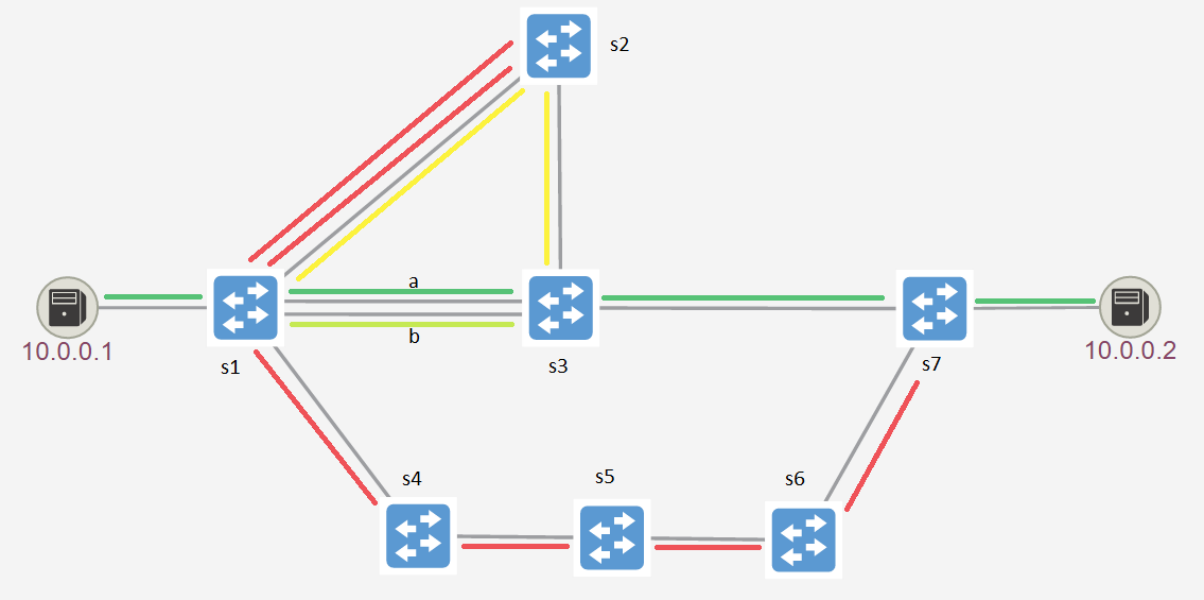

Figure 24- Path verification network

To verify that the proposed FRR application functioned as just described, the network in Figure 24 was generated in the Mininet platform, and the traffic paths monitored for each failure. Additionally, the path costs calculated by the application were recorded, to verify that they were correct. These results are presented in Table 8.

Table 8 – Path verification results

Failure	Path followed	Reported Path Cost (from PLR)	Expected path followed
None	s1-*a*-s3-s7	2	Yes
Link *a* of s1-s3 aggregate	s1-*b*-s3-s7	2	Yes
s1-s3 aggregate	s1-s2-s3-s7	3	Yes
s3	s1-s2-s1-s4-s5-s6-s7	6	Yes

These results show that the proposed FRR application correctly calculated and installed the expected primary and back-up paths, protecting the traffic against all three types of failures without requiring controller involvement after the failure.

5.2.2. Connectivity

The connectivity tests were performed for several networks, using the full protection scheme provided by the FRR application. These tests were performed in the Mininet environment, using the failure generating application. For each single failure (link, link aggregate, or node), a full connectivity test was performed to ensure that all paths were functional. The only allowed loss of connectivity was for the traffic generating hosts connected directly to the failed node.

The results of these tests are presented in Table 9 below, showing that for both networks tested the protection scheme was successful in protecting against all single failures.

Table 9 – Connectivity verification results

Network	Single link failure	Aggregate failure	Node failure
Example carrier core	Pass	Pass	Pass
South Africa core	Pass	Pass	Pass

5.2.3. Path Lengths and Label Stacks

The length of the back-up paths relative to the primary path is another important performance metric of the proposed FRR solution. As mentioned previously, the FRR application recorded the primary and back-up forwarding paths, path costs, and label stacks. A separate application was then used to process this data to calculate the actual path costs and total label stacks applied for node failures, as these paths are a combination of the link protecting and node protecting paths.

The ideal link protecting path costs are calculated using the paths that would be followed if routing protocol behaviour did not influence the path selection. However, the routing protocol did influence the actual paths installed, as the node-protecting rules had to be used if the link protecting label stack contained a routing label pointing directly to a potentially failed node. This slightly increases the actual link-protecting path costs. The third link-protecting path cost is calculated for a scenario where a node failure is initially assumed. In this scenario, the node-protecting paths are always used, unless the final destination is the next hop. If the final destination is the next hop, no node protecting rules exist, so the link protecting rules are used.

The ideal node protecting path cost was also calculated for this scenario where node failure is assumed. It should be noted that a node protecting path is always longer than or the same

length as the link protecting path for the same failure. However, the measured average node-protecting path ratios are lower than the average link-protecting path ratios. This is because a node-protecting path does not exist if the destination is the next hop, and the link-protecting path for this case is typically longer than the average path.

Since the behaviour of the installed rules was to initially assume link failure, and then assume node failure once a second failure was detected, the actual path followed in the case of a node failure is a combination of the link protecting and node protecting paths. This path is calculated by following the link protecting path to the second failure detecting node, and then following the node protecting path to the final destination. The average node-protecting path cost ratio, if link failure is always assumed first, is recorded in Table 10 as the 'link-then-node' ratio.

The actual installed paths were optimized, by using the node protecting paths at the first failure detecting node (instead of the link protecting paths) when it does not result in a longer link protecting path. This average path ratio is recorded in Table 10 as the 'conditional link-then-node' ratio. The results show that for the US carrier core network, this optimization substantially lowers the path cost compared to the simple link-then-node path costs. For the SA carrier core network, no improvement is seen.

For comparison between networks, the path costs are recorded as a ratio of back-up path to the primary path. The primary and back-up forwarding matrices were calculated 10 times, and the statistical mean of the path cost ratios over each iteration and back-up path in the network is recorded in Table 10.

Table 10 –Path cost measurements

Network	Link protecting path cost			Node protecting path cost		
	Ideal	Actual	Node failure assumed	Ideal	Link-then-node	Conditional link-then-node
US carrier core	1,4063	1,4064	1,4524	1,2741	1,3145	1,2863
SA carrier core	1,9352	1,9352	1,9519	1,3817	1,3978	1,3978

The label stacks applied to reroute the traffic along the explicit post-convergence path in each case are also important, as each label increases the traffic overhead slightly. Additionally, some MPLS switches can only handle a limited number of MPLS labels attached to a particular packet.

The link-protecting and node-protecting label stack lengths were measured using the installed paths, taking into account the various optimizations just described.

The results for 10 iterations using the US carrier core network with link aggregates (Figure 19) are presented in Table 11 below.

Table 11 – Label stack lengths for US carrier core

	Percentage of paths with a label stack length					Average stack length		
	0	1	2	3	4	Avg.	Max	Min
Ideal link	86,21	13,79	0	0	0	0,1379		
Conditional link	80,53	19,47	0	0	0	0,1947	0,2426	0,1721
Node	72,45	27,07	0,13	0,31	0,05	0,2846	0,3315	0,2627
Link-then-node	77,52	22,37	0,09	0	0,02	0,2263	0,2917	0,1993

These results show that for a vast majority of back-up paths, no labels are required to reroute the traffic. Very few node-protecting paths require more than one label to be applied. Since the node-protecting path is not used if the link-protecting path does not go through the failed node, the actual installed rules have an average label stack depth as recorded under 'link-then-node' in Table 11. A small percentage of paths, during some of the iterations, require 4 labels to be installed, which could be an issue in implementations with a limit on the number of MPLS labels allowed on a single packet.

The variance in the average stack length shows that some iterations of the forwarding algorithm yield more optimum results than others, even though the path costs are very similar for all runs. This is because the primary forwarding matrix arbitrarily selects any of the ECMP paths between two destinations. The back-up path algorithm also independently selects one of the ECMP paths. If the selected paths are very different, then the back-up algorithm must attach many labels to explicitly route the traffic through the desired post convergence path.

Therefore, future work could include calculating the forwarding matrix multiple times, and selecting the best set of rules, based on maximum number of labels required and the average label stack length.

The results for 10 iterations of the same test in the SA core carrier network are presented in Table 12. This network contains fewer ECMP paths than the US core network, so the two issues just discussed are not present. There are no paths requiring more than 1 label, and there is no variance in the results, so any particular iteration can be used as the most optimal.

Table 12 – Label stack lengths for SA carrier core

	Percentage of paths with a label stack length of:		Average stack length
	0	1	
Ideal link	72,22	27,78	0,2778
Conditional link	72,22	27,78	0,2778
Node	85,48	14,52	0,1452
Link-then-node	87,10	12,90	0,1290

5.2.4. Functionality Verification in a Distributed Testbed

To verify that the proposed FRR application can correctly configure a geographically distributed network that uses physical links and interfaces, the application was tested in the distributed network shown in Figure 21. The communication latencies between sites are shown in Table 7, and clearly show that a fast response to a failure cannot rely on the controller, but must be a local action.

The first test was to determine if the NOS can correctly discover the network. The link discovery module had to be modified slightly to correctly discover the topology, since the links were not direct links, but stitched inter-site links. The link discovery packets were dropped by the backbone switches, so a different broadcast address was used to prevent this.

The second step was to determine if the application can correctly configure the switches with the forwarding rules. Since the switches are the same software switches used in the Mininet platform, this test succeeded without any modification to the controller. Any OpenFlow 1.3 switch could be used, however, as long as it supported all the same functionality, as some of the functions used are optional in the OpenFlow standard.

Third, the functionality of the failover action was verified. The interfaces at each end of a particular link were administratively disabled, and the traffic flow observed. The traffic was correctly rerouted along the link-protecting back-up path.

5.3. Chapter Discussions

This chapter presented the results for both the available reactive approach as well as the proposed proactive approach.

The response time of the reactive approach was measured using different computational specifications, as well as differing amounts of installed intents. As expected, it was observed that increasing the computational resources of the controller did have an effect on the performance of the controller. This shows that the controller is well designed to efficiently use the available resources, but also shows that more expensive computational resources are required to obtain the best performance. It was also observed that the response time of the controller rapidly approaches, and exceeds, the 50 ms requirement as the number of installed intents increases. Finally, the communication latencies in the distributed network were measured, showing that a reactive approach is unlikely or impossible even if the response time of the controller was lower.

The proposed FRR application was tested in a number of different ways. First, correct operation was verified using a simple network, and each of the three kinds of single failure: single link, entire aggregate, and node. The reported path costs were also found to be correct, by comparing them to the expected back-up path through the network.

Protection against all failures was then verified in both networks modelling different carrier core networks. By introducing every possible single failure, one at a time, and testing the connectivity between all traffic generating hosts, it was shown that all traffic paths are rerouted around any single failure.

Next, the path costs and label stack lengths were measured to determine the efficiency of the solution. The costs of the back-up paths were higher than the primary paths, as expected. However, by applying link and node rules in an efficient manner, the FRR application was able to protect against all failures with shorter back-up paths than a simple approach of always following the node-protecting path from the first node that detects a failure. It was also found that the average label stack is low, with most paths not requiring any label to be applied. This is efficient from a traffic overhead point of view. However, with the large number of ECMP paths in the larger carrier network, some iterations of the forwarding algorithms yielded sub-optimal label-stack sets.

Finally, the functionality of the proposed application was verified in a geographically

distributed network, using physical links and interfaces. The only modification required was a trivial change in the controller's link discovery program, showing the proposed application can work in a real network, and not only the emulated Mininet platform.

The next chapter will present conclusions, as well as recommendations for future work.

Chapter 6

Conclusions and Recommendations

The previous chapters have presented the background, implementation, and results of the proposed solution. This chapter presents a summary of the work, conclusions that can be drawn, and recommendations for future work.

6.1. Summary

Given that resilience against failures is a critical feature of current networks, it is important that SDN is able to maintain the 50 ms fast restoration requirement present in current core networks. A thorough literature review was presented, and it was found that there was a need for a fast restoration solution that will meet all the requirements of a carrier core network. Reactive solutions guarantee that the optimal path is used, but suffer from processing and communication delays. Proactive approaches remove these delays, and will be fast enough. However, there are additional complexities due to the need to efficiently install back-up paths, which often require additional flow rules and pre-provisioned tunnels.

Chapter 3 presented the requirements of carrier core networks, and it was found that single failures include failures of single links, entire link aggregates, or nodes. The design of the fast reroute algorithm was also presented, along with an overview of the functional architecture of the proposed SDN application to implement the algorithm.

Chapter 4 presented the implementation of the entire SDN platform that was used to conduct the tests, detailing both the Mininet testbed as well as the geographically distributed GENI testbed. The implementation of the available reactive restoration application in ONOS was briefly presented, before thoroughly detailing the implementation of the proposed proactive FRR application. The performance metrics for each application were introduced, along with testing tools and network topologies used to measure the results.

Finally, Chapter 5 presented the measured results, along with discussions of each result. The response time of the reactive application was thoroughly tested, and the proactive approach was also tested using several topologies. The functionality of the proactive FRR application was then verified in a distributed testbed that made use of physical links and interfaces.

6.2. Conclusions

In this dissertation, the response time of a controller, ONOS, was investigated with regards to failure response. Using the reactive restoration application, it was found that the time it takes ONOS to fully reconfigure the network is dependent on the number of flow rule modifications required, and exceeds the 50 ms requirement when a relatively low number of modifications are required. Combined with the communication latencies observed in a geographically distributed network testbed, it is clear that a reactive approach is not practical for achieving carrier-grade resilience.

The proposed FRR application was tested with a number of different topologies, and its functionality was verified. Connectivity tests showed that for each topology, traffic was protected against any single link, link aggregate, or node failure. The back-up path lengths were measured, showing that the optimizations used in the application's path selection algorithm were efficient. The required label stacks were also measured, showing that most back-up paths do not require additional segment routing labels, and very few require more than 1 label. This shows that it is possible to efficiently protect against all single failures without controller involvement. As far as the author is aware, this is the first implementation of an SDN based FRR application that uses segment routing.

Since it has been shown that OpenFlow fast-failover groups are capable of switching to the back-up rules in well under 50 ms [17], the proposed proactive application is also definitely capable of meeting the 50 ms requirement.

Therefore, this dissertation has shown that a reactive approach will not be suitable for a carrier network, and has shown through the developed application that a proactive approach is possible, and efficient.

6.3.Recommendations

This dissertation presented an SDN application that was capable of protecting traffic against single failures within a network under a single administrative domain. This application could be extended in several ways to improve its suitability for real world deployment. Additionally, there are other issues that are worth exploring.

The first recommendation is that future work considers the impact of multiple failures within the network. The application could easily be extended to reconfigure the forwarding rules and install new back-up rules after a failure, based on the new topology. This process was not

implemented by the application, as the behaviour of the pre-installed back-up rules needed to be evaluated in isolation. Although the controller should be able to reconfigure the network in a few seconds, additional failures may occur simultaneously or before the network is reconfigured. The single-failure rules installed by the proposed application would likely handle some of these instances. However, the failure combinations that will cause traffic loss should be identified, and additional back-up rules calculated to protect the traffic. The statistical possibility of each combination of failures, the number of additional forwarding rules required, and the failures' effect on traffic are all factors that should be considered in deciding whether the additional rules should be installed.

Since only a single administrative domain was considered, edge switches represent single points of failure for traffic that exits the network through that particular edge switch. However, practical networks, such as the Internet, consist of multiple administrative domains. Future work should look at the possibility of re-routing traffic through a different edge switch, by communicating between the administrative domains. This would require an open, standardized, widely adopted interface between different SDN controllers, which does not currently exist.

Traffic engineering was also left as future work. Optimization of the primary forwarding path, as well as the back-up paths, should be considered. The shortest-path algorithm used by the proposed application for both primary and back-up paths does support a topology with weighted links. This opens the possibility for traffic engineering, by setting the link weights based on the link properties, current network congestion, and operator goals. This would ensure that the shortest-path route is optimal based on these criteria. However, some traffic flows would need to follow different paths, based on their QoS requirement, traffic type, or SLA. Segment routing enables this by adding extra routing labels at the ingress node, to force the traffic through intermediate nodes. However, if this intermediate node fails, the traffic may be dropped by the current solution, even though an alternative, sub-optimal route exists. Future work should consider a solution to this, possibly by removing the top label and forwarding based on the next label, if it is determined that the top-of-stack node has failed.

A further recommendation is that the application be implemented in a distributed controller. Ryu is a centralized controller, and only uses one controller instance. This presents a single point of failure in the control plane, and possibly a performance bottleneck. A distributed controller should be used to provide control plane resilience. The performance is less of an issue, since all forwarding rules are calculated ahead of time.

Finally, it was found that ONOS does not currently support OpenFlow fast-failover groups at the driver level. Since ONOS is aimed at wide area networks, this feature should be added to support a carrier grade fast restoration solution, such as the one proposed in this dissertation.

References

[1] Internet World Stats, “Internet Users in the World by Regions June 2016,” 30 June 2016. [Online]. Available: http://www.internetworldstats.com/stats.htm. [Accessed August 02 2016].

[2] D. Kreutz, F. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, “Software-Defined Networking: A Comprehensive Survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14-76, January 2015.

[3] Google, “OpenFlow @ Google,” April 2012. [Online]. Available: http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf. [Accessed 16 September 2016].

[4] 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,” in *9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, 2010.

[5] A. Greenberg, G. Hjalmtysson, D. A. 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. 5, pp. 41-54, 2005.

[6] D. Marschke, J. Doyle and P. Moyer, Software Defined Networking (SDN): Anatomy of OpenFlow, vol. 1, Lulu Publishing Services, 2015.

[7] ITU-T, “Y.3300 : Framework of software-defined networking,” June 2014. [Online]. Available: https://www.itu.int/rec/T-REC-Y.3300-201406-I. [Accessed 3 June 2015].

[8] N. Beheshti and Y. Zhang, “Fast Failover for Control Traffic in Software-defined Networks,” in *Global Communications Conference (GLOBECOM)*, Anaheim, California, 2012.

[9] V. Pashkov, A. Shalimov and R. Smeliansky, “Controller Failover for SDN Enterprise Networks,” in *Science and Technology Conference (Modern Networking Technologies)*, 2014.

[10] A. Raj and O. C. Ibe, “A survey of IP and multiprotocol label switching fast reroute schemes,” *Computer Networks*, vol. 51, no. 8, p. 1882–1907, 2007.

[11] S. Sharma, D. Staessens, D. Colle, M. Pickavet and P. Demeester, “OpenFlow: Meeting carrier-grade recovery requirements,” *Computer Communications*, vol. 36, no. 6, pp. 656-665, 2013.

[12] A. Atlas and A. Zinin, “Basic Specification for IP Fast Reroute: Loop-Free Alternates,” September 2008. [Online]. Available: https://tools.ietf.org/html/rfc5286. [Accessed 13 September 2016].

[13] S. Bryant, C. Filsfils, S. Previdi, M. Shand and N. So, “Remote Loop-Free Alternate (LFA) Fast Reroute (FRR),” April 2015. [Online]. Available: https://tools.ietf.org/html/rfc7490.

[Accessed 13 September 2016].

[14] Alcatel-Lucent, “Fast Reroute with Segment Routing,” Alcatel-Lucent, 2015.

[15] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona and P. Francois, “The Segment Routing Architecture,” in *IEEE Global Communications Conference*, 2015.

[16] S. H. Yeganeh, A. Tootoonchian and Y. Ganjali, “On Scalability of Software-Defined Networking,” *IEEE Communications Magazine,* pp. 136-141, February 2013.

[17] N. L. M. v. Adrichem, B. J. v. Asten and F. A. Kuipers, “Fast Recovery in Software-Defined Networks,” Delft University of Technology, Delft, The Netherlands, 2014.

[18] M. Borokhovich, L. Schiff and S. Schmid, “Provable Data Plane Connectivity with Local Fast Failover,” in *Proceedings of the third workshop on Hot topics in software defined networking*, Chicago, 2014.

[19] R. Izard, “How to Work with Fast-Failover OpenFlow Groups,” Atlassian , 26 April 2016. [Online]. Available: https://floodlight.atlassian.net/wiki/display/floodlightcontroller/How+to+Work+with+Fast-Failover+OpenFlow+Groups. [Accessed 17 September 2016].

[20] O. Salman, I. H. Elhajj, A. Kayssi and A. Chehab, “SDN Controllers: A Comparative Study,” in *Proceedings of the 18th Mediterranean Electrotechnical Conference*, Limasson, Cyprus, 2016.

[21] Open Networking Foundation, “OpenFlow Switch Specification Version 1.3.0,” 25 June 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf. [Accessed 17 September 2016].

[22] 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*, pp. 105-110, July 2008.

[23] Open Networking Foundation, “Services Area,” 2016. [Online]. Available: https://www.opennetworking.org/technical-communities/areas/services. [Accessed 23 September 2016].

[24] F. L. Faucheur, “IETF Multiprotocol Label Switching (MPLS) Architecture,” in *1998 1st IEEE International Conference on ATM*, 1998.

[25] P. Moyer, “Evolving SP WAN Architectures; how do routers keep up?,” Brocade, 24 August 2012. [Online]. Available: http://community.brocade.com/t5/Service-Providers/Evolving-SP-WAN-Architectures-how-do-routers-keep-up/ba-p/35767. [Accessed 20 September 2016].

[26] F. Lazzeri, G. Bruno, J. Nijhof, A. Giorgetti and P. Castoldi, “Efficient Label Encoding in Segment-Routing Enabled Optical Networks,” in *International Conference on Optical Network Design and Modeling*, Pisa, Italy, 2015.

[27] A. Sgambelluri, A. Giorgetti, F. Cugini, G. Bruno, F. Lazzeri and P. Castoldi, “First

Demonstration of SDN-based Segment Routing in Multi-layer Networks," in *Optical Fiber Communication Conference* , Los Angeles, California, 2015.

[28] L. Davoli, L. Veltri, P. L. Ventre, G. Siracusano and S. Salsano, "Traffic Engineering with Segment Routing: SDN-Based Architectural Design and Open Source Implementation," in *Fourth European Workshop on Software Defined Networks (EWSDN)*, 2015.

[29] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher and S. Ueno, "Requirements of an MPLS Transport Profile," September 2009. [Online]. Available: https://tools.ietf.org/html/rfc5654. [Accessed 22 March 2016].

[30] K. Miller, "Calculating Optical Fiber Latency," 9 January 2012. [Online]. Available: http://www.m2optics.com/blog/bid/70587/Calculating-Optical-Fiber-Latency. [Accessed 5 December 2016].

[31] ONOS, "Master: Experiment C - Intent Install/Remove/Re-route Latency," 17 September 2016. [Online]. Available: https://wiki.onosproject.org/pages/viewpage.action?pageId=11178307. [Accessed 28 November 2016].

[32] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood and A. W. Moore, "OFLOPS: An Open Framework for OpenFlow Switch Evaluation," in *Passive and Active Measurement: 13th International Conference*, Vienna, Austria, 2012.

[33] A. Sgambelluri, A. Giorgetti, F. Cugini, A. Giorgetti, F. Cugini and P. Castoldi, "OpenFlow-Based Segment Protection in Ethernet Networks," *Journal of Optical Communications and Networking*, vol. 5, no. 9, pp. 1066-1075, 2013.

[34] A. Capone, C. Cascone, A. Q. T. Nguyen and B. Sansò, "Detour planning for fast and reliable failure recovery in SDN with OpenState," in *11th International Conference on the Design of Reliable Communication Networks*, Kansas City, Missouri, 2015.

[35] N. L. M. v. Adrichem, F. Iqbal and F. A. Kuipers, "Computing backup forwarding rules in Software-Defined Networks," 31 May 2016. [Online]. Available: http://arxiv.org/abs/1605.09350. [Accessed 31 May 2016].

[36] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs and P. Sköldström, "Scalable fault management for OpenFlow," in *IEEE International Conference on Communicatiosn*, Ottawa, Canada, 2012.

[37] Cisco Networking Academy, Connecting Networks Companion Guide, Cisco Press, 2014.

[38] R. D. Doverspike, K. Ramakrishnan and C. Chase, "Structural Overview of ISP Networks," in *Guide to Reliable Internet Services and Applications*, London, Springer Science & Business Media, 2010, pp. 19-93.

[39] C. Filsfils, P. Francois, M. Shand, B. Decraene, J. Uttaro, N. Leymann and M. Horneffer, "Loop-Free Alternate (LFA) Applicability in Service Provider (SP) Networks," June 2012. [Online]. Available: https://tools.ietf.org/html/rfc6571. [Accessed 10 October 2016].

[40] P. Marais, "Next Generation Networks," 8 March 2007. [Online]. Available: https://www.sec.gov/Archives/edgar/data/1214299/000111667907001135/ex99-3.htm.

[Accessed 10 October 2016].

[41] P. Francois, C. Filsfils, A. Bashandy, B. Decraene and S. Litkowski, “Topology Independent Fast Reroute using Segment Routing,” 17 May 2016. [Online]. Available: https://datatracker.ietf.org/doc/draft-francois-rtgwg-segment-routing-ti-lfa/?include_text=1. [Accessed 10 October 2016].

[42] Linux Foundation, “Production Quality, Multilayer Open Virtual Switch,” 2016. [Online]. Available: http://openvswitch.org/. [Accessed 6 November 2016].

[43] S. Salsano, “[ovs-discuss] match/push/pop more than 3 MPLS labels,” 11 March 2015. [Online]. Available: https://mail.openvswitch.org/pipermail/ovs-discuss/2015-March/016903.html. [Accessed 6 November 2016].

[44] GENI, “About GENI,” 2016. [Online]. Available: http://www.geni.net/. [Accessed 4 December 2016].

[45] B. Lantz, B. Heller and N. McKeown, “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks,” in *Hotnets ’10*, Monterey, CA, USA., 2010.

[46] Open Networking Lab, “Intent Framework,” 24 May 2016. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Intent+Framework. [Accessed 11 November 2016].

[47] ONOS, “Experiment C Plan - Intent Install/Remove/Re-route Latency,” 30 August 2016. [Online]. Available: https://wiki.onosproject.org/pages/viewpage.action?pageId=3441828. [Accessed 28 November 2016].

[48] NetworkX, “Overview — NetworkX,” 2016. [Online]. Available: http://networkx.github.io/. [Accessed 5 December 2016].

[49] Broadband Infraco, “National Connectivity,” 2010. [Online]. Available: www.infraco.co.za/TechCapabilities/SitePages/National%20Architecture.aspx. [Accessed 11 November 2016].

[50] ONOS, “Installing and Running ONOS,” 2016. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Installing+and+Running+ONOS. [Accessed 28 November 2016].

[51] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. v. Reijendam, P. Weissmann and N. McKeown, “Maturing of OpenFlow and Software-defined Networking through deployments,” *Computer Networks*, vol. 61, no. 1, pp. 151-175, March 2014.

[52] R. Kloti, V. Kotronis and P. Smith, “OpenFlow: A Security Analysis,” in *Proceedings of the 21st IEEE International Conference on Network Protocols*, Göttingen, Germany, 2013.

[53] ONOS, “Members | ONOS,” ONOS, 2014. [Online]. Available: http://onosproject.org/members. [Accessed 4 June 2015].

[54] Brocade, “Components of OpenFlow on the switch,” Brocade, 2016. [Online]. Available: https://www.brocade.com/content/html/en/deployment-guide/brcd-fastiron-openflow-dp/GUID-71E1D938-5EE3-473E-B916-F167ED277E9B.html. [Accessed 17 September

2016].

[55] G. S. A. A. P. Pan, “ Fast Reroute Extensions to RSVP-TE for LSP Tunnels,” May 2005. [Online]. Available: https://tools.ietf.org/html/rfc4090. [Accessed 2016 May 20].

Appendix A

Modification to Open vSwitch to Support Additional MPLS Labels

By default, Open vSwitch supports a maximum of 3 MPLS labels attached to any particular packet [43]. If a flow rule or group action attempts to add a fourth label, the packet is dropped and the cause of the error is recorded in the Open vSwitch log.

This limit is fully defined in a single file (‘include/openvswitch/flow.h’) in the Open vSwitch source code. At the beginning of this file, the maximum number of MPLS flow labels is defined as 3:

```
#define FLOW_MAX_MPLS_LABELS 3
```

The rest of the codebase uses this definition whenever it requires the number of supported labels, so it is only necessary to change the value in one location.

A second modification is required, further down in the same file, as the number of labels is used implicitly without referring to the defined value. The original code, which is used to verify that the size of a flow is as expected before allowing the code to compile, is:

```
BUILD_ASSERT_DECL(offsetof(struct flow, igmp_group_ip4) + sizeof(uint32_t)
            == sizeof(struct flow_tnl) + 248 && FLOW_WC_SEQ == 36);
```

The second last term (‘248’) of this assert is the length of the flow, and needs to be modified to reflect the change in size of the MPLS section of the flow. An MPLS header is 4 bytes long, and padding is used by rounding the number of supported labels up to the nearest multiple of two. Therefore, with the default number of supported labels, the length of the MPLS header section in the flow is 16 bytes. To support an arbitrary number of labels, this is subtracted from the total of 248 bytes, and the length of the MPLS section added by calculating it from the defined value. The patch in [43] uses a different flow length, and requires modification as the flow length has changed in the latest version of OVS. The code is therefore changed to:

```
BUILD_ASSERT_DECL(offsetof(struct flow, igmp_group_ip4) + sizeof(uint32_t)
            == sizeof(struct flow_tnl) + 232 + (4 * ROUND_UP(FLOW_MAX_MPLS_LABELS, 2))
            && FLOW_WC_SEQ == 36);
```

With this modification, the codebase can be successfully recompiled and installed, and will now support the defined maximum number of MPLS labels.

Appendix B

Host Specific Labels

In a segment routing (SR) forwarding protocol, there are a number of complexities introduced by using a stack of multiple MPLS labels. Correct handling of the final label in the stack is one of these complexities, and a number of different methods are possible.

The top-of-stack label specifies a destination node, and this label must be removed before the destination node can make the next routing decision. This label can either be removed at the destination node, or at the node before the destination node. This second technique is called penultimate hop popping (PHP).

It is necessary, however, to know when the last MPLS label is being removed, as the Ethernet type must be changed from MPLS back to the protocol of the encapsulated packet (e.g. IPv4 or IPv6). If only one traffic protocol is received at the switches, it is only necessary to know that the label is the last label. This can be done by adding additional flow rules to match based on the bottom-of-stack (BOS) flag that is set on the last label.

With non-PHP, a switch must remove the label corresponding to itself before it can make the next routing decision, based on the label beneath. This requires the use of two flow tables and two lookups, but BOS-matching flow rules are only required on one switch, and no extra groups are required.

PHP, on the other hand, avoids this extra table and lookup by removing the destination node's label prior to forwarding. This is done by simply popping the top label when it corresponds to a neighbouring node. However, this requires BOS-matching flow rules for a node to be installed on all of the node's neighbouring switches. Additionally, since the label is only popped in one of the group's action buckets, and not in back-up action (as the back-up action will forward the packet to a different node), additional groups are also required.

However, neither of these solutions will work if multiple protocols are received at the edge switches. For instance, IPv4 and IPv6 have different Ethernet types, and therefore the action which pops the last label must know which protocol is underneath the label stack. Not only would this increase the number of rules and groups required, it also requires another criterion to match on to select the correct protocol. This information cannot be added to the MPLS header, as it only contains three fields besides the label: traffic class, bottom-of-stack, and time-

to-live. The traffic class field is used by the FRR application.

Therefore, to support multiple protocols, and also remove the need for flow rules and groups that match on the BOS flag, a host-specific label is added to incoming traffic before adding the label for the destination. PHP can then safely be used without BOS flow rules, as any switch label will not be the last label. Whenever the egress node receives a packet labelled with a host-specific label, it knows that the label is the last label and the Ethernet type must be changed when popping the label. If the incoming traffic is already MPLS traffic, it is not necessary to attach the host-specific label.

The phrase ‘host-specific labels’ is used because the focus for this dissertation was on the core network, and only IP traffic-generating hosts are connected to the edge switches. In a practical network, an edge switch would be connected to other networks, possibly of differing protocols. In this case, the label would specify the protocol and destination network, rather than the destination host.

This approach does require an additional MPLS label (unless the incoming traffic is already MPLS traffic), which increases the traffic overhead by adding 4 bytes to each packet. However, it also allows for a simpler forwarding protocol, as no BOS matching flow rules and groups are needed. A further advantage is that it ensures all traffic within the network is MPLS traffic, which may allow for cheaper switches to be used, as IP routers can be more expensive.