THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Cardigan: Development of a Distributer Router

Christopher Lorier

Dissertation

COMP591-12C (HAM)

Abstract

As the Internet continues to grow, many of our traditional approaches to computer networking need to be re-imagined to cope with the ever increasing demands placed on networks. Software Defined Networking (SDN) attempts to facilitate this process by allowing networking devices to be controlled in software. Control in software reduces the effort needed to modify the behaviour of devices, allowing for easier experimentation and customisability.

SDN remains a new concept and network operators are understandably wary of deploying unproven technology in production. Cardigan is a project to ease these concerns by deploying an SDN fabric in a production environment. To emphasise the benefits of customisability, the Cardigan deployment extends the SDN routing platform RouteFlow to allow several software controlled devices to be managed by configuring a single virtual router. This report describes the modifications made to RouteFlow to provide this abstraction and other changes made for the Cardigan deployment.

Acknowledgements

I would like to thank the following people for their support during this project: Richard Nelson, for his patient supervision of this project; Josh Bailey, Joe Stringer and the others involved in the Cardigan project; Martin Knoche and the team at REANNZ, and in particular Jamie Curtis, for providing me the opportunity to work on this project; and everyone at WAND, with special mention to Brad Cowie, Brendon Jones and Shane Alcock, for their technical expertise and for proof reading this report.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

ARP Address Resolution Protocol

BGP Border Gateway Protocol

CPqD Centro de Pesquisa e Desenvolvimento

CSV Comma Separated Value

IP Internet Protocol

IPv4 IP version 4

IPv6 IP version 6

IPC Inter-Process Communication

ISL Inter-Switch Link

LSP Label-Switched Path

MAC Media Access Control

MPLS Multi-Protocol Label Switching

NREN National Research and Education Network

REANNZ Research and Education Advanced Network of New Zealand

RFP RouteFlow Protocol

SDN Software Defined Networking

TCP Transmission Control Protocol

TTL Time To Live

UDP User Datagram Protocol

VLAN Virtual Local Area Network

VM Virtual Machine

WIX Wellington Internet Exchange

Chapter 1

Introduction

Computer networks are becoming ever more important in our lives. More and more devices are utilising network connectivity, from tablets to rice-cookers. This growth is increasing the demands on network hardware, as well as increasing the complexity of the networks that make up the Internet. The protocols and hardware used in networks need to adapt to be able to cope with this growth. Unfortunately, traditional network hardware is implemented as a closed box, with only narrow scope for network operators to modify behaviour to meet the requirements of their network.

Software Defined Networking [1] is an approach to networks that seeks to offer greater flexibility by allowing devices to be controlled remotely by software. Under the SDN model, devices are controlled with an open protocol which can be used to customise the behaviour of networks to a far greater degree than previously possible. Network operators are presented with a blank canvas on which they can design their network behaviour to suit the particular demands of their network.

Unfortunately, the size and importance of the Internet are also barriers to change. Testing in virtual environments cannot be performed at a meaningful scale and the ever increasing importance of connectivity coupled with the risk of interfering with complex interaction between protocols make network operators reluctant to experiment with unproven technologies in production.

Cardigan [18] is a project to assuage some of these doubts by deploying a hybrid SDN–legacy based network in production. The aim is to help dispel misconceptions around SDN and highlight some of the advantages offered by customisable centralised control of a network.

The Cardigan deployment consists of a distributed SDN fabric, based on the open-source routing platform RouteFlow [14]. It connects the Research and Education Advanced Network of New Zealand (REANNZ), a local National Research and Education Network (NREN) to the Wellington Internet Exchange (WIX). It advertises routes within REANNZ onto the WIX and passes production traffic.

RouteFlow is still under development and required extensions in order to meet the requirements of the Cardigan deployment. Support for Internet Protocol (IPv6) was added and the assignment of Media Access Control (MAC) addresses to ports was modified in order to comply with standards.

Fixing incorrect behaviour and including protocols that are standard to any current router does not represent a compelling argument to replace existing hardware. In order to demonstrate the advantages of SDN over legacy hardware, a Platform as a Service [9] model was applied to the management of the network. In this approach, multiple physical devices can be aggregated so that they can be managed as though they were a single device, and externally the network appears as a single logical layer 3 entity. The behaviour of the network is specified by configuring a virtual router and the entire network is automatically configured accordingly.

This abstraction reduces the complexity of the structure of the network and helps ensure consistency of policy. It minimises configuration, both by reducing the number of nodes that need to be provisioned and also by simplifying configuration of protocols within the wider network, for instance interior Border Gateway Protocol (iBGP).

This report describes the development of modifications made to RouteFlow for the Cardigan deployment, as well as the further development of the router aggregation extension following the Cardigan deployment.

1.1 Structure of this Document

- Chapter 2 provides background on SDN, introduces OpenFlow and describes the RouteFlow architecture.
- Chapter 3 describes the work carried out to extend RouteFlow for the Cardigan deployment and the subsequent development of the datapath

aggregation extension.

- Chapter 4 describes further improvements that could be made to the datapath aggregation extension.
- Chapter 5 describes the Cardigan deployment and its impact and demonstrates the behaviour of the datapath aggregation extension.
- Chapter 6 summarises the work undertaken in this project and its impact.

Chapter 2

Background

In this chapter the main concepts and prior work that this project builds upon are explained in detail. Firstly, the principles of SDN and the motivation behind them are explored. Secondly, OpenFlow, a standard for software defined devices is introduced and explained. Finally, the RouteFlow architecture is described, with detailed explanations on the role and design of each component of the architecture and how they relate to each other.

2.1 Software Defined Networking

The architecture of traditional networking hardware is split into two planes, the data plane, which is responsible for forwarding packets and the control plane, which controls the behaviour of the data plane. The data plane is designed to process as many packets at as close to line rate as possible. It performs basic packet processing operations, such as forwarding packets to the correct port or modifying fields such as IP TTL. Packets are sent to the control plane when they require extra processing. These packets are generally network protocol packets that inform the device of the state of the network. The control plane uses the information in these packets to modify the behaviour of the data plane.

SDN is an approach to networking that offers greater configurability and control by allowing devices to be controlled by software. This is achieved by creating an API for communication between the data plane and control plane of networking devices. Separating the control plane from hardware allows it to be controlled in software, allowing the control of devices to be customised to the specific needs of the network.

In the SDN model, the data plane remains on the switch or router, which is referred to as a Datapath. As with legacy hardware, the datapath perfoms simple operations on packets and can forward packets to the control plane if need be. The design of the device is simpler, as it no longer needs to implement the handling of complicated network protocols, only the data plane and the API.

The control plane software is called the Controller and has full control over the behaviour of the datapath. The software can be run remotely on commodity hardware, allowing for centralised control of devices.

There are many benefits to this model, some of which include:

Reduced Cost: Forwarding devices are simpler and the control plane runs on commodity hardware. This can reduce the cost of networks.

Improved Reliability: Jain *et al.* [8] assert that "switch failures typically result from software rather than hardware failures". The use of open-source software as controllers allows operators to diagnose software faults that would otherwise be hidden from them. Separating the controller from the device allows for redundant controllers, that are able to fail over should the control software fail.

Reduced Configuration: Centralised control of network devices reduces the need to repeat configuration on multiple devices. The controller is configured and datapaths are automatically updated accordingly.

Uniform Policy Enforcement: Configuration of multiple devices from a centralised controller ensures all devices are updated correctly when network policy is modified.

Rapid Development: Control in software reduces barriers to experimentation, allowing for rapid development of new protocols or services.

2.2 OpenFlow

OpenFlow [12] is a key technology in SDN, providing a standard specification for datapaths. The OpenFlow Protocol defines messages that are sent between OpenFlow switches and controllers. These messages control the connection, allow for state information to be communicated to the controller, allow the controller to modify the behaviour of the switch and allow for notification of

errors.

Packet handling is controlled by the use of "Flows". Flows define how a specific set of packets should be handled. The flow specifies the characteristics of the packets to which the flow should be applied and a set of operations to perform on those packets. Flows have a priority, time-out values, counters, a set of matches and a set of actions. These are stored in a table called a flow table. When a datapath receives a packet, it finds the highest priority flow that matches the packet and applies the actions contained in that flow to the packet.

Table 2.1: OpenFlow 1.0 Match Fields

Ingress Port	Ethernet Destination Address	Ethernet Source Address	Ethertype	VLAN ID	VLAN Priority	IPv4 Source Address	IPv4 Destination Address	IPv4 Protocol	IPv4 ToS Bit	TCP/UDP Source Port	TCP/UDP Destination Port

Matches specify masked values for fields used to identify packets. The fields are mostly fields in the packet header but may also include other information such as the ingress port. The match fields used in OpenFlow version 1.0 [2] are listed in Table 2.1. Later versions of the OpenFlow Protocol can match against more fields, including Multiprotocol Label Switching (MPLS) and IPv6 fields.

Actions define how packets that match a flow are handled. There are a large number of possible actions, such as forwarding a packet out a specified port, encapsulating the packet and sending to the controller, modifying a header field, or pushing and popping Virtual Local Area Network (VLAN) or MPLS tags.

2.2.1 OpenFlow Versions

The OpenFlow specification has been updated regularly, with the latest release, version 1.3.2, currently under ratification [16]. The most widely supported version is version 1.0, however, this lacks important features such as MPLS and IPv6 support or the ability to decrement IP TTL fields.

OpenFlow version 1.2 [4] includes support for MPLS and IPv6, as well as other useful features such as groups (see Section 4.1) and multiple flow tables (see Section 4.2).

OpenFlow version 1.0 is the most widely supported by hardware, with support for later versions very rare and generally incomplete. The switches used in the

Cardigan deployment were a Pronto 3290 and a Pronto 3780. These support all of OpenFlow version 1.0, but only have partial support for later versions. Most significantly, they do not support MPLS matching or modification. Because of this, this project has made every effort to use OpenFlow version 1.0 whenever possible. However, when OpenFlow version 1.0 was not sufficient for a required feature, OpenFlow version 1.2 has been used.

2.2.2 Openflow Control Platforms

OpenFlow control platforms are pre-existing implementations of the OpenFlow Protocol. They handle the communication with datapaths and present an Application Programming Interface (API) to developers to create and send OpenFlow Protocol messages. Custom control software can be built on top of these without having to repeat the effort of implementing the sending and receiving of packets or the parsing of OpenFlow Protocol messages.

Three control platforms were used during this project, NOX [6], POX [11] and Ryu [15]. NOX, a C++ based platform and POX, a Python based platform, only implement OpenFlow version 1.0. Ryu is also based in Python, but implements all versions of OpenFlow.

2.3 RouteFlow

RouteFlow [14] is a software-defined layer 3 routing architecture that is currently under development by Centro de Pesquisa e Desenvolvimento (CPqD). It provides a centrally controlled Internet Protocol (IP) routing platform that is easily customisable and works with any Linux based networking software.

RouteFlow consists of a series of virtual routers, each associated with an individual datapath. Network protocol traffic sent to the datapaths is redirected to the associated virtual router. The virtual router learns routes as though it were receiving traffic directly and these routes are replicated in the flow table of the datapath.

The virtual routers are Linux Virtual Machines (VMs) running legacy routing software. Each port on a datapath is associated with a unique interface on the VM. When the datapath receives control plane traffic on a port it forwards the traffic to its controller, which passes the traffic on to the corresponding VM port. Likewise, network protocol packets sent by the VM are forwarded

out the corresponding datapath port.

The forwarding behaviour of the datapath is found by polling the host and route tables on the VM. This information is used to create entries in the datapath flow table.

2.3.1 RouteFlow Design

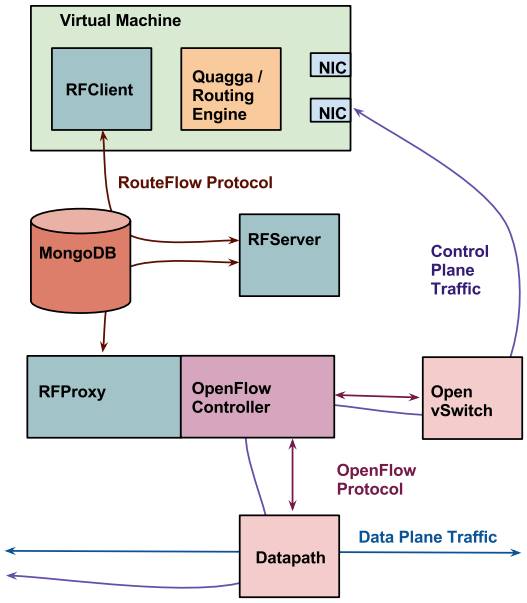

Figure 2.1: RouteFlow Architecture

RouteFlow consists of three main components: RouteFlow Client (RFClient), which controls the VM; RouteFlow Server (RFServer), which controls the association of datapaths and VMs; and RouteFlow Proxy (RFProxy), which controls the datapaths (see Figure 2.1). These communicate between each other using the RouteFlow Protocol (RFP), a protocol defining the interactions between RouteFlow components. Inter-Process Communication (IPC) is performed with MongoDB [13], a NoSQL database. The MongoDB is also used to store the state and configuration of the platform. An instance of Open vSwitch [17] is used to forward control plane traffic between the datapaths and VMs.

RouteFlow Protocol

The RFP provides messages to communicate the state of the network, the mapping between VMs and datapaths, configuration messages and the details of routes learned by VMs to be added to datapaths. The message types relevant

to this project are:

Port Register: These are used to indicate the presence of a port on a VM that is ready for use.

Port Config: These are used to modify the configuration of ports on the VM.

Datapath Port Register: These are used to indicate the presence of a port on a datapath that is ready for use.

Datapath Down: These indicate that a datapath is no longer able to communicate with the controller.

Data Plane Map: These are used to inform RFProxy of the mapping between virtual switch ports and datapath ports.

Route Mod: These contain information about routes learned by the VM so they can be added to datapaths.

The RouteFlow VM and RFClient

RFClient is the part of RouteFlow that interacts with the VM. It is responsible for controlling the state of the VM and communicating configuration information and the state of the routing table to RFServer.

The VM can be any Linux system able to run a routing engine. Typically, LXC [10] or another lightweight virtual container is used. Network protocols are implemented by open-source routing engines, such as Quagga [5]. The routing engine sends routing updates to the Linux kernel, causing the Linux machine to function as a router. Quagga supports many routing protocols including Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP) but any routing software that runs in Linux can be used, allowing for easy implementation of novel routing protocols.

RFClient is written in C++ and runs as a daemon inside the VMs. It uses Netlink sockets to listen to changes in the host or route tables, and sends the routes learned to RFServer using RFP Route Mod messages. If necessary, RFClient performs address resolution of gateways. It does this by attempting to establish a Transmission Control Protocol (TCP) connection to the gateway, forcing the VM to send Address Resolution Protocol (ARP) requests in order to resolve the address.

RFClient also controls the configuration of the VM interfaces by disabling or

enabling interfaces in response to RFP Port Config messages and sends RFP Port Register messages to inform RFServer when interfaces have been enabled.

RouteFlow Proxy

RFProxy manages the datapaths. It interacts with OpenFlow control platforms to exchange OpenFlow protocol messages with the datapaths, installing and removing flows and receiving status information.

RFProxy informs RFServer of the state of datapaths, sending RFP Datapath Port Register messages when a port becomes active and RFP Datapath Down messages when communication to a datapath has been lost. RouteFlow does not detect when individual ports go down, even though this is possible in all versions of OpenFlow.

RFProxy manages the installation and deletion of flows on datapaths. It listens for RouteMod messages from RFserver, converts them into OpenFlow Protocol Modify State messages and sends them to the datapaths.

RFProxy also is responsible for redirecting control plane traffic between the VMs and datapaths. An OpenFlow controlled virtual switch, such as Open vSwitch, connects to each port on the VM. RFProxy runs as the controller for this switch as well as the RouteFlow datapaths. The virtual switch has one flow, matching all traffic and redirecting it to the controller. Likewise the datapaths have flows redirecting all routing protocol traffic to the controller. Whenever a packet is received from the virtual switch, RFProxy forwards the packet via the corresponding datapath port. Similarly, traffic received from the datapaths are forwarded via the virtual switch. The associations between ports are received from RFServer as RFP Data Plane Map messages.

There are multiple versions of RFProxy, each used with a different OpenFlow control platforms. Because of the need for differing versions, RFProxy is designed to be as lightweight as possible, allowing it to be easily ported to new platforms.

RouteFlow Server

RFServer is a standalone application written in Python and is responsible for the core system logic. RFServer is aware of the full state of the network and is responsible for associating datapaths with VMs and distributing RFP messages between RFProxy and RFClient instances.

RFServer is configured by providing a CSV file specifying the mapping between ports on the VM to the ports on the datapaths. These are read into a table in MongoDB.

The current state of the datapaths and VMs in the network is stored in another table in MongoDB called "RFTable". When ports are enabled, either on the datapath or on the VM, a registration message is sent to RFserver. When both registration messages are received, RFServer sends a RFP Datapath Port Map message to RFProxy, informing it of the mapping between virtual switch ports and datapath ports. Once it has confirmed that forwarding between the VM and datapath is working, RFServer sets the port state to be active. When RFP Datapath Down messages are received, RFServer instructs RFClient to disable all interfaces on the corresponding VM with RFP Datapath Config messages.

RFServer controls the state of datapaths by sending Route Mod messages to RFProxy. RFServer generates Route Mod messages when a datapath first comes up, adding the flows redirecting control protocol traffic to the controller. RFServer is also responsible for transferring Route Mod messages received from RFClient to the correct RFProxy instance, as well as ensuring they contain the correct datapath information.

Chapter 3

Implementation

This chapter describes the work carried during this project. The work focuses on modifying RouteFlow in order to achieve two goals. The first goal was to make RouteFlow fit for purpose for the Cardigan deployment. This required correcting the behaviour when assigning MAC addresses to ports, the addition of IPv6 support and the ability to aggregate multiple datapaths to function as a single logical layer 3 entity. The implementation of aggregation used in the Cardigan deployment relied on every datapath being directly connected to every other. The second goal of this project was the further development of the aggregation extension to allow any number of arbitrarily interconnected datapaths to be aggregated into a single logical entity.

3.1 Ethernet Addresses

In order to reduce the number of flows installed on each switch, RouteFlow used a single MAC address for all device ports. However, the EUI-48 standards state "EUI-48 identifiers are intended to identify items of real physical equipment or parts of such equipment such as separable subsystems or individually addressable ports" [7]. Flow capacity was not a limiting factor for Cardigan, so correct implementation of standards was seen as preferential.

To have a separate MAC address for each port means that for every route added to the datapath, a separate flow for each port other than the egress must be added. Each flow matches based on the ingress port and Ethernet destination address in addition to the IP destination address range specified by the route.

Implementation of this feature was fairly simple. Ingress port matching was already included in the RFP, though it was yet to be implemented in RFProxy.

Multiple approaches were considered to generate the extra Route Mod messages required to install the new flows on datapaths. Either RFClient or RFServer could have performed the task but ultimately RFServer was chosen. The most obvious approach was for RFClient to generate extra Route Mod messages each time it learned a route. The current Ethernet Destination Address matches were already being created by RFClient as part of the generation of Route Mod messages and RFClient was the only part of RouteFlow to have access to the address information. However, for aggregation, it is necessary for RFServer to generate Route Mod messages to distribute routes to the internal ports. If the additional Route Mod messages were generated by RFClient, RFServer would receive multiple Route Mod messages for each route learned. This would require RFServer to have to determine whether or not the route had already been distributed among internal ports each time it received a Route Mod message. Implementing the MAC address matching in RFServer also allows for the reuse of the code to distribute routes to internal ports.

RFClient was modified to no longer send Ethernet Destination Address matches and the RFP Port Register message was updated to allow RFClient to communicate the MAC address of each port to RFServer. RFServer stores that information in RFTable and when it receives a Route Mod message RFServer creates a new Route Mod message for each RFTable entry belonging to the correct VM.

3.2 IPv6

One of the requirements for the Cardigan deployment was the ability to route IPv6 traffic. This proved a very challenging feature to implement, as it impacted on many different aspects of the RouteFlow architecture. It required modification to all three main components and exposed some of the most immature code in the RouteFlow code base. Covering a broad range of code also meant it required a broad knowledge base, including Netlink and Ryu as well as the details of IPv6.

RFServer required the least modification of the three main components. The creation of Route Mod messages to configure datapaths as they are brought

up was extended to include IPv6 control protocols, such as Internet Control Message Protocol version 6 (ICMPv6).

RFClient was modified to be protocol agnostic. This required modifying the code for retrieving address information from the route and host tables, as well as modifying the code for resolving MAC addresses for hosts. This was made more difficult by some remnants of previous approaches to Address Resolution Protocol (ARP) requests that had not been properly updated with other changes to the code, including unused variables and inappropriate casting of values.

IPv6 is not supported in OpenFlow version 1.0, so RFProxy required updating to use OpenFlow version 1.2. To implement this change, a new OpenFlow control platform needed to be used. Before this project there were RFProxy implementations for use with two control platforms, NOX and POX. Unfortunately, neither of these support versions of OpenFlow after 1.0. Ryu was chosen as the control platform to use, for the following reasons:

- Ryu includes support for later versions of OpenFlow.
- Ryu is Python based, allowing reuse of code from the POX implementation.
- Some work had already started on a Ryu RFProxy extension.

However, the work undertaken to create a Ryu RFProxy implementation was at too early a stage to be used for Cardigan, so a new implementation was created.

At the time, Ryu was very poorly documented. For instance, to set a flow to match on a specific IPv6 destination address Ryu uses the function `set_ipv6_dst` which takes the variables `dst` and `mask`. As Python is a dynamically typed language, nowhere in the code does it explicitly specify what types these variables should be. Only through careful examination of the Ryu code, coupled with trial and error was it able to be determined that these variables were Python tuples containing eight 16bit integers, an unusual method of storing an IPv6 address.

3.3 Aggregation

To create a Platform as a Service model for network management [9], multiple OpenFlow switches are aggregated to created a single logical virtual router. Operators are presented with the abstraction of a single router, a platform they are familiar with that gives them full control of the external behaviour of the fabric. Externally the fabric interacts with the network as though it was a single device.

As an important goal of this abstraction is reducing configuration, the configuration of the aggregation should not be onerous. With this in mind, the router aggregation was implemented so that the only configuration required is the usual RouteFlow configuration file and a single additional file, detailing Inter-Switch Links (ISLs).

The underlying structure of the network is configured automatically. There were two distinct approaches used at different stages of this progress. For the Cardigan deployment, a full mesh topology is required. This means that every datapath has an ISL connecting it directly to every other datapath. This was then further extended to allow for any arbitrary topology of interconnected datapaths. The only requirement being that each datapath has a path to every other datapath, either directly or via other datapaths in the fabric.

Forwarding between datapaths follows the shortest path by hop-count. In the full-mesh design this only requires forwarding directly to the egress node. For arbitrary topologies, Label-Switched Paths (LSPs) are created between each pair of nodes.

The behaviour of the VM should be the same when controlling one datapath or several, such that no additional knowledge is required to control the network other than to create the initial ISL configuration file. Because the routing platform, and consequently the routing protocols, are unmodified, correct interoperation with traditional networks is assured.

The logic to achieve this is contained within RFServer, which controls the configuration of ISLs and is aware of the underlying topology. As RFServer receives Route Mod messages from RFClient, these are distributed among the datapaths to ensure correct forwarding behaviour is maintained.

3.3.1 Configuration and Port Handling

To configure the ISLs a second CSV file is used, mapping the endpoints of ISLs to their corresponding ports on other datapaths. The MAC addresses for these ports are also specified, as unlike external ports the MAC information cannot be received from RFClient. A new table was created in the MongoDB to store this information.

Handling RFP Datapath Port Registration messages for ISLs follows a similar procedure to external Datapath Port Registration messages. When a Datapath Port Registration message is received, both configuration tables are checked. If there is an ISL entry, a second port state table in the MongoDB is used instead of RFTable.

RFProxy informs RFServer of datapaths becoming unresponsive with a RFP Datapath Down message. RFServer then sends messages to RFClient resetting each VM port individually. Because of this, the behaviour to inform RFClient when a Datapath Down message has been received does not need to be modified (with one exception, see Section 3.3.2). RFClient disables each VM port associated with the datapath, while paths between the remaining datapaths are recalculated to allow the rest of the fabric to remain functional.

3.3.2 Splitting

Should one part of the network ever be completely separated from the rest of the network, correct operation of the full network becomes impossible. Packets will be unable to be forwarded out of ports on the opposite side of the split. The VM runs unmodified and is unaware of the structure of the underlying network, so it will continue to advertise routes to ports on one side of the divide to hosts connected to the other side. Even if the VM was aware of the separation, responding to this scenario would require a significant modification of the routing engine. This also cannot be treated as a rare failure condition that requires external response as it is likely to occur whenever a datapath is enabled. Datapaths and ISLs may come up in any order and there is no reason to expect they will do so in a manner that ensures they are connected to each other.

To handle this scenario, the first datapath to be enabled is designated by RFServer as the “root datapath”. As ISLs become active, they are only enabled

when there is a path between one of the datapaths and the root. When a datapath is disabled, any connected datapaths are tested for a path to the root. If there is no such path, they are disabled. If the root datapath itself ever goes down, another root datapath is chosen arbitrarily from the currently active datapaths.

A more sophisticated approach could find the largest section of the network to remain active or allow configuration of priorities to datapaths. The naïve approach was adequate for this project as this is generally only an issue very briefly as the fabric is first started.

This issue was irrelevant to the Cardigan deployment, as it consists of only two datapaths and each only has one external port. If any part of the network fails, correct operation is impossible regardless.

3.3.3 Inter-Switch Paths

To ensure correct handling of packets, when RFServer receives a Route Mod message it must update the forwarding behaviour of each datapath in the fabric.

With a full mesh topology this simply requires forwarding directly to the egress datapath, which then forwards out the correct port. When RFServer receives a Route Mod message, it creates additional Route Mod messages for each external port on the fabric as well as each ISL port on the egress node. The flows added to the egress node forward directly out the egress port and others forward to the egress node via the appropriate ISL port.

For more complicated topologies, LSPs are needed between datapaths in order to prevent loops. A LSP is a virtual circuit created in a packet-switching network where packets are forwarded based on labels added to packets. When the packet enters the fabric, a label specifying the path to take through the fabric is added to the packet header. The label is removed before the packet exits the fabric, returning it to its original state.

Networks with centralised control must take care when modifying the forwarding behaviour of the network as it is impossible to guarantee that updates to the flow tables on separate datapaths will occur simultaneously. While updates are being sent to datapaths, some datapaths will have the new flows loaded while others are still using the old flows. The forwarding behaviour of

the network will be unpredictable and may include loops. By using LSPs, the new flows will use new labels that will not be matched by the previous flows and the packets will be dropped.

LSPs provide a further benefit in the case that the previous path is still able to function, I.E. when the topology change has been caused by an ISL becoming active. The old LSP can remain in use while the new flows are being installed, meaning no traffic is lost during the change over. To achieve this, flows are installed using the following procedure: firstly, flows for the new LSPs are loaded onto the datapaths; secondly, all routes currently loaded onto the datapaths are modified to use the new LSP; and, finally, the flows for the old LSP are deleted.

The LSP flows are loaded onto the datapaths as soon as they are calculated. This means that when a new route is learned the new flows only need to be installed at the source of each LSP. The routes are added simply by loading flows for each external port, pushing the label associated with the path and forwarding out the appropriate port.

To install LSPs on a datapath, a flow is added for each ISL port. The flow matches the path label as well as the correct Ethernet destination address and ingress port. The flow updates the Ethernet source and destination addresses, swaps the label to the correct value to be matched at the next hop and forwards to the correct egress port. The exception to this is the case of the penultimate hop, which performs the same actions but removes the label instead. This is a technique known as penultimate hop popping and is used at present for simplicity, but this should not be used in a multiple table set-up (see Section 4.2).

To reduce the number of flows on each datapath, the LSPs share labels whenever possible. All LSPs to the same destination datapath that pass through a single datapath use the same labels from that datapath onwards.

Ordinarily label-switched paths use MPLS tags as labels. However, because matching or modifying MPLS tags is not supported in OpenFlow version 1.0, VLAN tags have been used in this implementation instead.

3.3.4 Path Calculation

LSPs are calculated whenever there is a change in topology, either caused by an ISL becoming active or a Datapath Down message being received.

LSPs are calculated using Dijksta's algorithm [3] with equal weights. When an ISL becomes active, the ISL between the two datapaths is necessarily the shortest path between them and the LSPs are updated. The two datapaths are checked for any other existing LSPs that can be improved by using the new link. Any replacement LSP that is created gets pushed onto a queue. Then, the head of that queue is popped and checked to see if it can be extended. Any datapath with an ISL connecting it to the last datapath of the new LSP compares the new LSP to its existing LSP with the same destination. If the existing LSP can be improved, a replacement LSP is created and appended to the queue. In this way every LSP is visited from smallest to largest, so any LSP found that can be modified will necessarily become the new shortest path.

When a Datapath Down message is received, first all LSPs that pass through that datapath have their lengths recorded and are removed. Then for each deleted LSP, all LSPs to the same destination with the same length are found and tested for possible extensions. This follows a similar procedure as when an ISL becomes active. Any LSP shorter than the deleted path does not need to be tested, as any extension to it would necessarily be shorter than the previously existing LSP.

3.3.5 Resilience

To ensure the fabric continues to operate correctly after datapaths become unreachable it is enough to recalculate and reload LSPs, unless the network becomes split. However, reintegrating datapaths when they become reachable again requires reloading all routes currently known by the VM onto the datapath.

When a datapath becomes inactive the corresponding interfaces on the VM are set down by RFClient. The VM then removes routes through these interfaces from its routing table and RFClient sends Route Mod messages reflecting these changes. LSPs are recalculated automatically and the rest of the fabric will continue to function correctly.

When the datapath becomes active again, it must be updated with all the

routes currently loaded onto the fabric. This must be performed by RFServer, as RFClient is unaware of the underlying topology. To do this, RFServer needs to store all routes as it adds them to datapaths and remove them as they are deleted. When a datapath becomes active, it resends all currently active routes without needing modifications to RFClient.

3.3.6 Synchronicity

RFServer runs in two separate threads, one polling for messages from RFProxy and one polling for messages from RFClient. Care must be taken to ensure the two threads are operating synchronously with one another.

When RFServer resends routes to a datapath due to a topology change, whether updating routes to use new paths or reloading routes onto a datapath that has been down and has returned, these will be sent by the thread polling RFProxy. These actions need to occur atomically, as new routes may be received from RFClient before all the routes are reloaded. This could cause incorrect behaviour, if, for instance, a message to delete a route arrived at a datapath earlier than the message to add it. An approach using queues to minimise the impact of topology changes was investigated, but was rejected in favour of a simple lock on sending Route Mod messages. This prevents the thread polling RFClient from sending any Route Mod messages until the routes are reloaded. This is not the most efficient approach possible, but it is mitigated by the fact that the IPC system already functions as a queue and the assumption that topology changes should be a relatively rare event.

Chapter 4

Work Ahead

This chapter describes some of the further improvements that could be made to the datapath aggregation extensions: how load balancing could be approached, how flows can be reduced with the use of flow tables in later versions of OpenFlow and how resilience could be improved with the use of OpenFlow barriers and by automatically detecting port activity.

4.1 Load Balancing

One key advantage aggregation can offer is the ability to balance traffic load between links. This can reduce cost when provisioning networks and ease congestion in the network. The best approach to load-balancing is dependent on the structure of the network. For example, the Valiant [20] approach to load balancing has capacity requirements for ISLs that decrease inversely proportional to the number of nodes. However, this approach has the requirement of a full-mesh topology.

Load balancing is also limited by current implementations of OpenFlow. Open vSwitch has support for most OpenFlow version 1.1 and 1.2 features, but not all. Most crucially, support for group tables is not implemented. Group tables allow flows to have multiple sets of actions, called "buckets". These can be applied by various selection techniques including applying all buckets to copies of the packet or picking a bucket at random. Random bucket selection allows for a much simpler approach to load-balancing than is possible with only pre-written rules.

Because of these challenges, implementation of load balancing was not included

in this project. It was hoped that a simple interface could be created by which load balancing modules customised to suit the topology of the given network could be loaded. However, in implementing shortest path forwarding, this interface was violated several times. How packet forwarding is approached and, consequently, how load-balancing is approached affect all of the functions of RFServer. Determining when ports should be made active or brought down, how routes are loaded onto datapaths and the initial configuration of datapaths are all dependant on load-balancing. Furthermore, careful synchronisation of events is necessary, which complicates the interaction between the module and RFServer. In the end the interface was discarded and an alternative that resolves these issues has not been created.

4.2 Flow Tables

From version 1.1 onwards, OpenFlow supports multiple flow tables. Instead of forwarding packets to a port on the switch, flows can send packets to a second table, where they can be matched again and have new actions applied to the packets. This can be used to greatly reduce the number of flows required for matching packets. For this project, however, this was not implemented as it was not necessary for the Cardigan deployment and it is not possible in OpenFlow version 1.0.

Since multiple tables are supported in the same versions of OpenFlow as MPLS, table based label handling can utilise MPLS labels. However, as there could be conflict between externally added MPLS tags and tags added for passing through the aggregated datapaths, penultimate hop popping should not be used. When tables are supported, they could be used as follows:

Table 0: The first table matches on ingress port and Ethernet destination address. If the packet arrived at an ISL port and has the correct Ethernet destination address, it is forwarded to table 1. If the packet arrived at an external port and has the correct Ethernet destination address, it is forwarded to table 3. Otherwise the packet is dropped.

Table 1: Table 1 matches on MPLS label. It then determines whether this is the final node in the path. If not, it swaps the label to the next value and forwards the packet out the appropriate egress port. If this is the final node, it pops the first label and sends the packet to table 2.

Table 2: Unlike VLAN tags, MPLS tags do not contain Ethertype information for the payload. So table 2 contains flows matching on labels associated with the Ethertype field to be added to the packet. After the Ethertype information is re-added, the packet is forwarded to table 3.

Table 3: Table 3 matches on layer 3 destination address. If the egress port is on the local datapath, the packet is fowarded without modification. If the egress port is on another datapath, a label representing the Ethertype and another representing the path are pushed onto the packet and the packet is forwarded via the appropriate ISL.

4.3 Barriers

In this project, it has been assumed that routes will be added to switches in the order that they are sent. This cannot be guaranteed to be the case. It was initially thought that the worst possible outcome is that traffic will be lost briefly as it tries to use a path that is not fully created but in some cases loops could be formed. For instance, consider the case in which the egress port for traffic to a specific address range changes to a port on a different datapath. If the original egress datapath changes its forwarding behaviour before the new egress datapath, the two datapaths would forward traffic sent to that address range back and forth between themselves.

This has been mitigated by always sending routes to the egress node last but to truly guarantee the correct order of flow installation, the OpenFlow Protocol provides Barrier Request and Barrier Reply messages. A controller can send a Barrier Request message to ensure a datapath is up to date. When the datapath receives a Barrier Request message the datapath applies all outstanding OpenFlow Protocol messages and then the datapath sends a Barrier Reply messages back to the controller. Route changes could be sent to the new egress datapath first, followed by a Barrier Request message. Flows can be installed on the remaining datapaths after the Barrier Reply is received.

Using barriers would require addition of a Barrier Request message to RFP as well as significant modification to RFProxy. Unfortunately there was not sufficient time to do this after this issue was discovered in this project.

4.4 Port Activity Detection

RouteFlow contains no ability to detect when an individual port on a datapath has become inactive. This can cause RouteFlow to advertise routes that it cannot perform the necessary forwarding behaviour to provide. This problem is exacerbated with the use of ISLs. With a single datapath some routes may be erased eventually, for instance as BGP sessions time out, though in many circumstances this will not occur and the routes will remain forever. With datapath aggregation the inactive links between datapaths will never be discovered. For this implementation the inability to detect link status was seen as a flaw in RouteFlow that needs to be addressed in future but is out of scope for this project.

Chapter 5

Outcomes

This chapter describes the impact of the work undertaken in this project. Firstly, the Cardigan deployment and the results of that deployment are described and, secondly, the further development of the aggregation code is evaluated.

5.1 Cardigan

Cardigan was deployed on the 23rd of January 2013 using code developed in this project, passing production traffic between REANNZ and the WIX. The initial deployment included the modified Ethernet address matching and the earlier full-mesh implementation of the aggregation code. The IPv6 code was added soon afterward.

The deployment consists of two datapaths, one at REANNZ and one at WIX (see Figure 5.1). Each datapath has one port connecting to the respective networks and the datapaths are connected to each other with an ISL. Routes are being learnt from the WIX and REANNZ and distributed to the datapaths resulting in approximately 1,000 flows on each datapath at a time. This has run for over 5 months without major incident.

A paper on the Cardigan deployment, co-authored by J.P. Stringer, Q. Fu, C. Lorier, R. Nelson and C.E. Rothenberg, was submitted to ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN) 2013. It was accepted to appear at the conference as a short 2-page paper as part of the poster session.

The code developed for the Cardigan deployment has been included in the

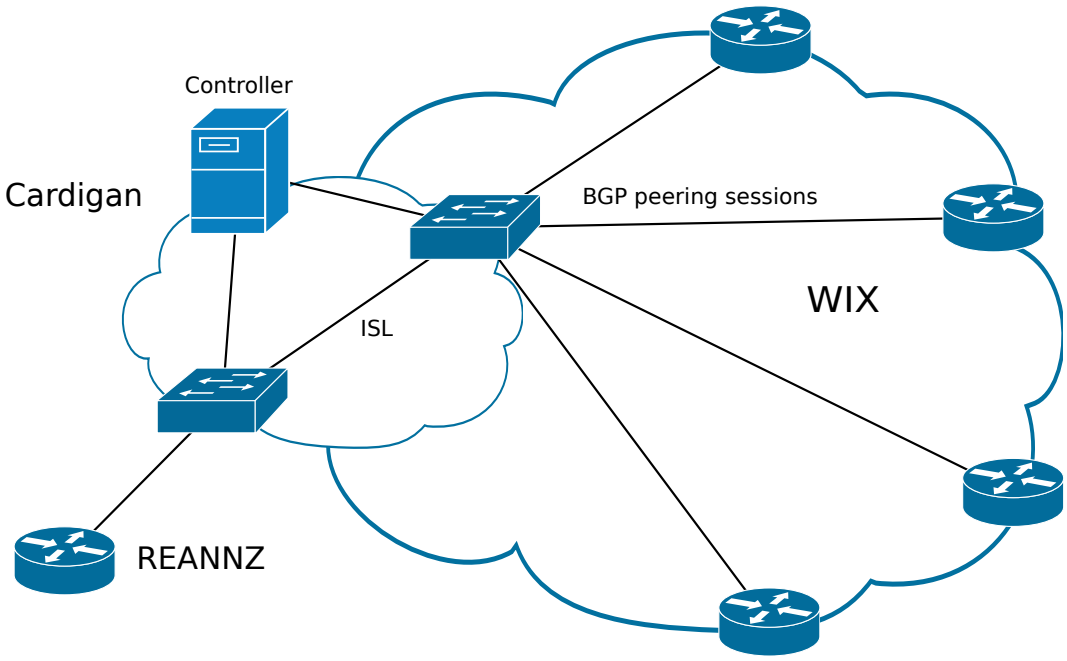

Figure 5.1: Cardigan pilot deployment

main RouteFlow code base. The process for having code accepted is informal but the code is vetted by the lead RouteFlow developers before being included.

5.2 Aggregation

The aggregation code has been tested in a virtual network with various configurations and topologies. The virtual router has routes added and removed as datapaths are enabled and disabled to verify that routes are being added correctly in response to changes in the network.

To demonstrate the main features of this code a network was set up using four Open vSwitch switches controlled by a single RouteFlow VM (see Figure 5.2). These are connected in a loop, so that each switch is connected by an ISL to two other switches. There are three external VMs connected to switches 1, 2 and 4. Appendix A contains tables showing the output from using the Open vSwitch `ovs-ofctl dump-flows` command on each switch after various changes are made to the fabric. The output has been edited to remove the default flows handling network control traffic and for readability.

Table A.1 shows the network in its initial state after the paths between switches have been created. Each datapath has two ISLs and therefore learns three

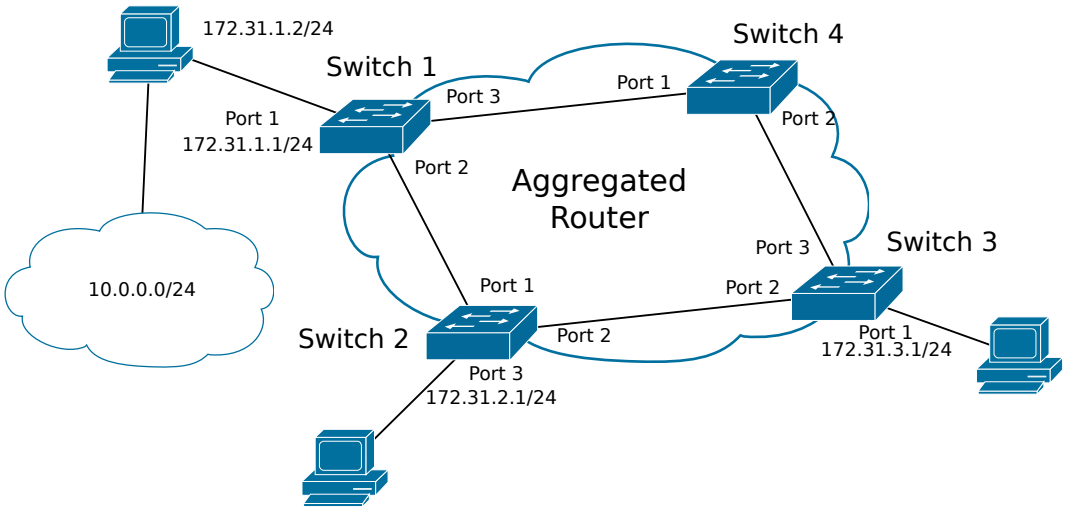

Figure 5.2: Virtual network used for demonstration

flows, one for each LSP. The flows output via the appropriate ISL and update Ethernet addresses and labels. The flows match packets arriving on the other ISL based on Ethernet destination address and label. Because paths use penultimate hop popping, the penultimate switch on each path removes the tags and the final switch will forward based on IP destination address.

Table A.2 shows the flows after the route to 10.0.0.0/24 via 172.31.1.2 has been added to the VM routing table. RFClient has resolved the gateway address and the routes to 10.0.0.0/24 and the gateway have been loaded onto the datapaths. To add the routes to the egress datapath, flows have been installed for each ISL port. To add the routes to other datapaths, flows are installed for each external port. Switch 3 has no external ports and therefore learns no new flows when a route is added.

Table A.3 shows flows adjusting to the removal of Switch 2 from the network. The datapaths are automatically updated to use the alternative path through Switch 4. For example, the first route on Switch 3 has been modified to output to port 3 (via Switch 4) instead of port 2 (via Switch 3).

Table A.4 shows the state of the fabric after Switch 3 has been re-added to the network. The routes currently active in the fabric have been reloaded automatically onto Switch 3.

Chapter 6

Conclusion

This report has described the advantages of applying the abstraction of a single router to the management of a network, based on the work of Keller and Rexford [9]. Aggregating multiple datapaths to function as a single logical layer 3 entity reduces configuration, improves consistency of policy and simplifies the structure of the networking. Using the abstraction of a single router allows the network to remain fully configurable while presenting an interface network operators are familiar with.

An implementation of this model has been created by extending the SDN routing architecture RouteFlow. Multiple OpenFlow switches are controlled by a centralised server that receives routing updates from a VM running Linux routing software. The datapaths are updated automatically in response to changes in the topology of the fabric to ensure correct forwarding behaviour.

The datapath aggregation abstraction was used in the Cardigan deployment to illustrate the benefits afforded by SDN. The Cardigan deployment is one of the first deployments of an SDN fabric in a production environment in the world and it is the SDN fabric to be deployed as a full member of a public Internet exchange point in New Zealand.

This report has also detailed the improvements made to RouteFlow as part of the Cardigan project. These include supporting new protocols, the addition of limited OpenFlow 1.2 support, porting to a new OpenFlow control platform and correcting protocol compliance. These changes have been included in the main RouteFlow development branch and the source code is freely available from the RouteFlow public repository [19].

Glossary

Address Resolution Protocol (ARP) the protocol used to find the MAC address for a host given its IPv4 address.

Border Gateway Protocol (BGP) the protocol used to share information about routes to networks throughout the Internet.

C++ a compiled, statically typed programming language. It is an extension of the C programming language, adding object oriented features and other enhancements such as exception handling.

Controller the software used to control datapaths in the SDN model.

Datapath a forwarding element in the SDN model.

Ethernet a set of standards and technologies used for communication between devices within a network. Ethernet splits packets into frames, which specify the MAC address of the intended recipient and provide error checking.

Flow an entry in the flow table of an OpenFlow switch. It specifies how certain traffic should be handled.

Interior Border Gateway Protocol (iBGP) a protocol used to share information about routes to networks within an organisation. It requires that all routers within that organisation be connected in a full-mesh topology.

Internet Control Message Protocol version 6 (ICMPv6) the version of the Internet Control Message Protocol (ICMP) for IPv6. ICMP is used to send control messages to report errors and provide diagnostic functions.

Internet Protocol (IP) the protocol used to locate hosts on the Internet. There are two versions of IP in use today, IP version 4 (IPv4) and IP version 6 (IPv6).

Internet Protocol version 6 (IPv6) the latest version of IP. IPv6 features a much larger address space than IPv4 to accommodate the growth of the Internet.

Layer 2 the second layer of the OSI model of computer networking, also known as the data link layer. Layer 2 is responsible for passing traffic between hosts connected to the same network. The most widespread layer 2 protocol is Ethernet.

Layer 3 the third layer of the OSI model of computer networking, also known as the network layer. Layer 3 passes traffic between networks. The chief protocol of layer 3 is IP.

Media Access Control (MAC) address a 48 bit address used to identify individual pieces of hardware connected to a network.

Multiprotocol Label Switching (MPLS) a protocol used to create virtual circuits in packet switched networks by assigning labels to packets.

Netlink a socket family used to send networking information between the Linux kernel and processes running in user-space.

Open Shortest Path First (OSPF) a protocol used to route IP packets within an organisation.

Python a dynamically typed, interpreted, high-level programming language. Python offers a large standard library and is designed to result in code that is intuitive and easy to read.

Virtual Local Area Network (VLAN) a virtual layer 2 network created by applying tags to packets representing separate broadcast domains.

References

[1] ONF Market Education Committee et al. Software-defined networking: The new norm for networks. *ONF White Paper. Palo Alto, US: Open Networking Foundation*, 2012.

[2] OpenFlow Switch Consortium et al. Openflow switch specification version 1.0., 2009.

[3] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[4] The Open Networking Foundation. Openflow switch specification version 1.2., 2011.

[5] Free Software Foundation. Quagga Software Routing Suite. http://www.nongnu.org/quagga/.

[6] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[7] IEEE Standards Association. Guidelines for use of the 24-bit Organisationally Unique Identifiers (OUI). http://standards.ieee.org/develop/regauth/tut/eui.pdf.

[8] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, August 2013.

[9] Eric Keller and Jennifer Rexford. The "platform as a service" model

for networking. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, volume 4. USENIX Association, 2010.

[10] lxc Linux Containers. Linux containers. http://lxc.sourceforge.net/.

[11] J Mccauley. Pox: A python-based openflow controller. http://www.noxrepo.org/pox/about-pox/.

[12] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[13] Peter Membrey, Eelco Plugge, and Tim Hawkins. *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress, 2010.

[14] Marcelo R Nascimento, Christian E Rothenberg, Marcos R Salvador, Carlos NA Corrêa, Sidney C de Lucena, and Maurício F Magalhães. Virtual routers as a service: The routeflow approach leveraging software-defined networks. 2011.

[15] Nippon Telegraph and Telephone Corporation. Ryu. http://http://osrg.github.io/ryu/.

[16] Open Networking Foundation. ONF Specifications: OpenFlow. urlhttps://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[17] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[18] Jonathan Philip Stringer, Qiang Fu, Christopher Lorier, Richard Nelson, and Christian Esteve Rothenberg. Cardigan: Deploying a distributed routing fabric. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Hong Kong, China, August 2013.

[19] The RouteFlow Community. RouteFlow source code. http://github.com/CPqD/RouteFlow.

[20] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 263–277, New York, NY, USA, 1981. ACM.

Appendix A

Demonstration Datapath Flow Tables

Table A.1: Initial Datapath Flow Tables

Datapath	Priority	Matches	Actions
Switch 1:	49200	in_port=2 dl_vlan=3 dl_dst=66:aa:aa:aa:aa:aa:12 dl_type=0x8100	output:3 set_field:66:aa:aa:aa:aa:aa:14→eth_src set_field:66:dd:dd:dd:dd:dd:41→eth_dst strip_vlan
	49200	in_port=3 dl_vlan=2 dl_dst=66:aa:aa:aa:aa:aa:14 dl_type=0x8100	output:2 set_field:66:aa:aa:aa:aa:aa:12→eth_src set_field:66:bb:bb:bb:bb:bb:21→eth_dst set_field:2→vlan_vid
	49200	in_port=3 dl_vlan=1 dl_dst=66:aa:aa:aa:aa:aa:14 dl_type=0x8100	output:2 set_field:66:aa:aa:aa:aa:aa:12→eth_src set_field:66:bb:bb:bb:bb:bb:21→eth_dst strip_vlan
Switch 2:	49200	in_port=2 dl_vlan=3 dl_dst=66:bb:bb:bb:bb:bb:23 dl_type=0x8100	output:1 set_field:66:bb:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:aa:12→eth_dst set_field:3→vlan_vid
	49200	in_port=1 dl_vlan=2 dl_dst=66:bb:bb:bb:bb:bb:21 dl_type=0x8100	output:2 set_field:66:bb:bb:bb:bb:bb:23→eth_src set_field:66:cc:cc:cc:cc:cc:32→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=1 dl_dst=66:bb:bb:bb:bb:bb:23 dl_type=0x8100	output:1 set_field:66:bb:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:aa:12→eth_dst strip_vlan
Switch 3:	49200	in_port=2 dl_vlan=4 dl_dst=66:cc:cc:cc:cc:cc:32 dl_type=0x8100	output:3 set_field:66:cc:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:dd:43→eth_dst strip_vlan
	49200	in_port=3 dl_vlan=1 dl_dst=66:cc:cc:cc:cc:cc:34 dl_type=0x8100	output:2 set_field:66:cc:cc:cc:cc:cc:32→eth_src set_field:66:bb:bb:bb:bb:bb:23→eth_dst strip_vlan
	49200	in_port=3 dl_vlan=2 dl_dst=66:cc:cc:cc:cc:cc:34 dl_type=0x8100	output:2 set_field:66:cc:cc:cc:cc:cc:32→eth_src set_field:66:bb:bb:bb:bb:bb:23→eth_dst set_field:1→vlan_vid
Switch 4:	49200	in_port=1 dl_vlan=4 dl_dst=66:dd:dd:dd:dd:dd:41 dl_type=0x8100	output:2 set_field:66:dd:dd:dd:dd:dd:43→eth_src set_field:66:cc:cc:cc:cc:cc:34→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=3 dl_dst=66:dd:dd:dd:dd:dd:43 dl_type=0x8100	output:1 set_field:66:dd:dd:dd:dd:dd:41→eth_src set_field:66:aa:aa:aa:aa:aa:14→eth_dst set_field:1→vlan_vid
	49200	in_port=2 dl_vlan=1 dl_dst=66:dd:dd:dd:dd:dd:43 dl_type=0x8100	output:1 set_field:66:dd:dd:dd:dd:dd:41→eth_src set_field:66:aa:aa:aa:aa:aa:14→eth_dst strip_vlan

Table A.2: Datapath Flow Tables After Addition of Route

Datapath	Priority	Matches	Actions
Switch 1:	16640	ip in_port=3 dl_dst=66:aa:aa:aa:aa:aa:14 nw_dst=10.0.0.0/24	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	16640	ip in_port=2 dl_dst=66:aa:aa:aa:aa:aa:12 nw_dst=10.0.0.0/24	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	49200	in_port=2 dl_vlan=3 dl_dst=66:aa:aa:aa:aa:aa:12 dl_type=0x8100	output:3 set_field:66:aa:aa:aa:aa:aa:14→eth_src set_field:66:dd:dd:dd:dd:41→eth_dst strip_vlan
	49200	in_port=3 dl_vlan=2 dl_dst=66:aa:aa:aa:aa:aa:14 dl_type=0x8100	output:2 set_field:66:aa:aa:aa:aa:aa:12→eth_src set_field:66:bb:bb:bb:bb:21→eth_dst set_field:2→vlan_vid
	49200	in_port=3 dl_vlan=1 dl_dst=66:aa:aa:aa:aa:aa:14 dl_type=0x8100	output:2 set_field:66:aa:aa:aa:aa:aa:12→eth_src set_field:66:bb:bb:bb:bb:bb:21→eth_dst strip_vlan
	16720	ip in_port=2 dl_dst=66:aa:aa:aa:aa:aa:12 nw_dst=172.31.1.2	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	16720	ip in_port=3 dl_dst=66:aa:aa:aa:aa:aa:14 nw_dst=172.31.1.2	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
Switch 2:	16640	ip in_port=3 dl_dst=12:a1:a1:a1:a1:03 nw_dst=10.0.0.0/24	output:1 set_field:66:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:12→eth_dst
	49200	in_port=2 dl_vlan=3 dl_dst=66:bb:bb:bb:bb:bb:23 dl_type=0x8100	output:1 set_field:66:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:12→eth_dst set_field:3→vlan_vid
	49200	in_port=1 dl_vlan=2 dl_dst=66:bb:bb:bb:bb:bb:21 dl_type=0x8100	output:2 set_field:66:bb:bb:bb:bb:bb:23→eth_src set_field:66:cc:cc:cc:cc:32→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=1 dl_dst=66:bb:bb:bb:bb:bb:23 dl_type=0x8100	output:1 set_field:66:bb:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:12→eth_dst strip_vlan
	16720	ip in_port=3 dl_dst=12:a1:a1:a1:a1:03 nw_dst=172.31.1.2	output:1 set_field:66:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:12→eth_dst
Switch 3:	16640	ip in_port=1 dl_dst=12:a1:a1:a1:a1:02 nw_dst=10.0.0.0/24	output:2 set_field:66:cc:cc:cc:cc:32→eth_src set_field:66:bb:bb:bb:bb:23→eth_dst push_vlan:0x8100 set_field:1→vlan_vid
	49200	in_port=2 dl_vlan=4 dl_dst=66:cc:cc:cc:cc:32 dl_type=0x8100	output:3 set_field:66:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:43→eth_dst strip_vlan
	49200	in_port=3 dl_vlan=1 dl_dst=66:cc:cc:cc:cc:34 dl_type=0x8100	output:2 set_field:66:cc:cc:cc:cc:32→eth_src set_field:66:bb:bb:bb:bb:23→eth_dst strip_vlan
	49200	in_port=3 dl_vlan=2 dl_dst=66:cc:cc:cc:cc:34 dl_type=0x8100	output:2 set_field:66:cc:cc:cc:cc:32→eth_src set_field:66:bb:bb:bb:bb:23→eth_dst set_field:1→vlan_vid
	16720	ip in_port=1 dl_dst=12:a1:a1:a1:a1:02 nw_dst=172.31.1.2	output:2 set_field:66:cc:cc:cc:cc:32→eth_src set_field:66:bb:bb:bb:bb:23→eth_dst push_vlan:0x8100 set_field:1→vlan_vid
Switch 4:	49200	in_port=1 dl_vlan=4 dl_dst=66:dd:dd:dd:dd:41 dl_type=0x8100	output:2 set_field:66:dd:dd:dd:dd:43→eth_src set_field:66:cc:cc:cc:cc:34→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=3 dl_dst=66:dd:dd:dd:dd:43 dl_type=0x8100	output:1 set_field:66:dd:dd:dd:dd:41→eth_src set_field:66:aa:aa:aa:aa:14→eth_dst set_field:1→vlan_vid
	49200	in_port=2 dl_vlan=1 dl_dst=66:dd:dd:dd:dd:43 dl_type=0x8100	output:1 set_field:66:dd:dd:dd:dd:41→eth_src set_field:66:aa:aa:aa:aa:14→eth_dst strip_vlan

Table A.3: Datapath Flow Tables After Datapath Down

Datapath	Priority	Matches	Actions
Switch 1:	16640	ip in_port=3 dl_dst=66:aa:aa:aa:aa:aa:14 nw_dst=10.0.0.0/24	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	16640	ip in_port=2 dl_dst=66:aa:aa:aa:aa:aa:12 nw_dst=10.0.0.0/24	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	49200	in_port=2 dl_vlan=3 dl_dst=66:aa:aa:aa:aa:aa:12 dl_type=0x8100	output:3 set_field:66:aa:aa:aa:aa:aa:14→eth_src set_field:66:dd:dd:dd:dd:dd:41→eth_dst strip_vlan
	16720	ip in_port=2 dl_dst=66:aa:aa:aa:aa:aa:12 nw_dst=172.31.1.2	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	16720	ip in_port=3 dl_dst=66:aa:aa:aa:aa:aa:14 nw_dst=172.31.1.2	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
Switch 3:	16640	ip in_port=1 dl_dst=12:a1:a1:a1:a1:02 nw_dst=10.0.0.0/24	output:3 set_field:66:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:43→eth_dst push_vlan:0x8100 set_field:1→vlan_vid
	49200	in_port=2 dl_vlan=4 dl_dst=66:cc:cc:cc:cc:32 dl_type=0x8100	output:3 set_field:66:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:43→eth_dst strip_vlan
	16720	ip in_port=1 dl_dst=12:a1:a1:a1:a1:02 nw_dst=172.31.1.2	output:3 set_field:66:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:43→eth_dst push_vlan:0x8100 set_field:1→vlan_vid
Switch 4:	49200	in_port=1 dl_vlan=4 dl_dst=66:dd:dd:dd:dd:41 dl_type=0x8100	output:2 set_field:66:dd:dd:dd:dd:43→eth_src set_field:66:cc:cc:cc:cc:34→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=1 dl_dst=66:dd:dd:dd:dd:43 dl_type=0x8100	output:1 set_field:66:dd:dd:dd:dd:41→eth_src set_field:66:aa:aa:aa:aa:14→eth_dst strip_vlan

Table A.4: Datapath Flow Tables After Datapath Restored

Datapath	Priority	Matches	Actions
Switch 1:	16640	ip in_port=3 dl_dst=66:aa:aa:aa:aa:aa:14 nw_dst=10.0.0.0/24	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	16640	ip in_port=2 dl_dst=66:aa:aa:aa:aa:aa:12 nw_dst=10.0.0.0/24	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	49200	in_port=2 dl_vlan=3 dl_dst=66:aa:aa:aa:aa:aa:12 dl_type=0x8100	output:3 set_field:66:aa:aa:aa:aa:aa:14→eth_src set_field:66:dd:dd:dd:dd:dd:41→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=4 dl_dst=66:aa:aa:aa:aa:aa:12 dl_type=0x8100	output:3 set_field:66:aa:aa:aa:aa:aa:14→eth_src set_field:66:dd:dd:dd:dd:dd:41→eth_dst set_field:4→vlan_vid
	49200	in_port=3 dl_vlan=5 dl_dst=66:aa:aa:aa:aa:aa:14 dl_type=0x8100	output:2 set_field:66:aa:aa:aa:aa:aa:12→eth_src set_field:66:bb:bb:bb:bb:bb:21→eth_dst strip_vlan
	16720	ip in_port=2 dl_dst=66:aa:aa:aa:aa:aa:12 nw_dst=172.31.1.2	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
	16720	ip in_port=3 dl_dst=66:aa:aa:aa:aa:aa:14 nw_dst=172.31.1.2	set_field:12:a1:a1:a1:a1:01→eth_src set_field:02:b1:b1:b1:b1:b1→eth_dst output:1
Switch 2:	16640	ip in_port=3 dl_dst=12:a1:a1:a1:a1:03 nw_dst=10.0.0.0/24	output:1 set_field:66:bb:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:12→eth_dst
	49200	in_port=2 dl_vlan=4 dl_dst=66:bb:bb:bb:bb:bb:23 dl_type=0x8100	output:1 set_field:66:bb:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:12→eth_dst strip_vlan
	49200	in_port=1 dl_vlan=7 dl_dst=66:bb:bb:bb:bb:bb:21 dl_type=0x8100	output:2 set_field:66:bb:bb:bb:bb:bb:23→eth_src set_field:66:cc:cc:cc:cc:32→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=6 dl_dst=66:bb:bb:bb:bb:bb:23 dl_type=0x8100	output:1 set_field:66:bb:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:12→eth_dst set_field:3→vlan_vid
	16720	ip in_port=3 dl_dst=12:a1:a1:a1:a1:03 nw_dst=172.31.1.2	output:1 set_field:66:bb:bb:bb:bb:bb:21→eth_src set_field:66:aa:aa:aa:aa:12→eth_dst
Switch 3:	16640	ip in_port=1 dl_dst=12:a1:a1:a1:a1:02 nw_dst=10.0.0.0/24	output:3 set_field:66:cc:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:dd:43→eth_dst push_vlan:0x8100 set_field:1→vlan_vid
	49200	in_port=2 dl_vlan=4 dl_dst=66:cc:cc:cc:cc:cc:32 dl_type=0x8100	output:3 set_field:66:cc:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:dd:43→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=5 dl_dst=66:cc:cc:cc:cc:cc:32 dl_type=0x8100	output:3 set_field:66:cc:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:dd:43→eth_dst set_field:1→vlan_vid
	49200	in_port=3 dl_vlan=7 dl_dst=66:cc:cc:cc:cc:cc:34 dl_type=0x8100	output:2 set_field:66:cc:cc:cc:cc:cc:32→eth_src set_field:66:bb:bb:bb:bb:bb:23→eth_dst strip_vlan
	16720	ip in_port=1 dl_dst=12:a1:a1:a1:a1:02 nw_dst=172.31.1.2	output:3 set_field:66:cc:cc:cc:cc:cc:34→eth_src set_field:66:dd:dd:dd:dd:dd:43→eth_dst push_vlan:0x8100 set_field:1→vlan_vid
Switch 4:	49200	in_port=2 dl_vlan=5 dl_dst=66:dd:dd:dd:dd:dd:43 dl_type=0x8100	output:1 set_field:66:dd:dd:dd:dd:dd:41→eth_src set_field:66:aa:aa:aa:aa:14→eth_dst set_field:5→vlan_vid
	49200	in_port=1 dl_vlan=4 dl_dst=66:dd:dd:dd:dd:dd:41 dl_type=0x8100	output:2 set_field:66:dd:dd:dd:dd:dd:43→eth_src set_field:66:cc:cc:cc:cc:cc:34→eth_dst strip_vlan
	49200	in_port=2 dl_vlan=1 dl_dst=66:dd:dd:dd:dd:dd:43 dl_type=0x8100	output:1 set_field:66:dd:dd:dd:dd:dd:41→eth_src set_field:66:aa:aa:aa:aa:14→eth_dst strip_vlan