RheaFlow: An Improved Software Defined Network Router

A thesis
submitted in fulfillment
of the requirements for the degree
of
Master of Science
in
Computer Science
at
The University of Waikato

by

Oladimeji Fayomi

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Department of Computer Science
Hamilton, New Zealand
July 27, 2016

Abstract

An Software Defined Networking (SDN) router translates routing information from a standard routing control plane into forwarding decisions on an SDN-enabled device. This thesis presents the design of RheaFlow an SDN router that uses Internet Protocol (IP) routing control logic to modify forwarding behaviour on OpenFlow switches. RheaFlow is implemented as a set of network applications on the Ryu OpenFlow controller platform. It utilises the Ryu Application Programming Interface (API) to build a modular SDN router architecture that is easily extendable and simple to configure.

RheaFlow adopts and improves the IP routing via OpenFlow based on the approach of RouteFlow, an existing SDN router. The RheaFlow design centralises the routing control plane which is distributed in RouteFlow. It also removes RFProtocol, an internal control protocol used by RouteFlow.

Acknowledgements

I would like to acknowledge and thank my supervisor Dr. Richard Nelson for his patience, support and guidance throughout the course of this work. I would also like to thank Brad Cowie and Richard Sanger for their assistance, inputs and advice during this project.

To my mom, thanks for your support and encouragement through the years. It has kept me going during the trying times. I would like to thank my siblings and Omoniyi Alimi for their support during this project.

Finally, I would like to appreciate Stephen Judd and Lola Bamgbose for their support during this project. Thank you.

Contents

List of Figures

List of Tables

List of Acronyms

ACL	Access Control List
API	Application Programming Interface
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
CE	Control Element
CSV	Comma-separated Values
FIFO	First In, First Out
FE	Forwarding Element
FIB	Forwarding Information Base
ForCES	Forwarding and Control Element Separation
ICMP	Internet Control Message Protocol
IPC	Inter-Process Communication
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
JSON	JavaScript Object Notation
MAC	Media Access Control
NDP	Neighbour Discovery Protocol
NE	Network Entity
NETCONF	Network Configuration Protocol
OF-CONFIG	OpenFlow Configuration and Management Protocol

ONOS	Open Network Operating System
OVSDB	Open vSwitch Database Management Protocol
OSPF	Open Shortest Path First
PC	Personal Computer
REST	Representational State Transfer
RIB	Routing Information Base
RIP	Routing Information Protocol
SDN	Software Defined Networking
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VLAN	Virtual LAN
VM	Virtual Machine
WSGI	Web Server Gateway Interface
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

Software Defined Networking (SDN) is a new approach to networking that enables network operators to manage network services and control the forwarding decisions of network devices by using applications that are based on open standard implementations and abstracted from the network devices they control, while still remaining globally aware of the network topology. The SDN paradigm is based on the notion that it is faster, cheaper and more flexible to make configuration changes and scale network resources to meet demands in a network by re-writing applications that control forwarding decisions on network devices than it is to replace or upgrade network equipment. SDN gives network operators fine-grained control of their network and enables them to dynamically provision network devices on the fly in response to changing traffic patterns on the network. Legacy network infrastructure does not provide the same level of flexibility as SDN, and the scaling of network resources in response to demand usually happens at great cost in time and resources.

In spite of the benefits offered by SDN, the adoption of SDN by network operators has been slow. Apart from barriers such as cost, reliability and security concerns associated with the adoption of a relatively new technology like SDN. A major obstacle that has slowed SDN's adoption is the lack of interoperation and integration with existing network infrastructure and equipment. The centralised control of SDN-enabled devices often makes them incompatible and difficult to deploy on an existing network with legacy network devices which use a mixture of protocols and platforms that decentralise control. Network operators could adopt a clean slate approach and build a new SDN tailored to their requirements; however, the costs associated with building a new SDN are prohibitive. According to [16], migrating a large ISP network to SDN will only be feasible using a step-by-step approach, since replacing the whole network at once will be too costly. This step-by-step approach in which SDN-enabled devices are gradually added to an existing network necessitates the need for an interface between legacy network devices in the network and SDN-enabled devices.

An SDN-enabled device is a network device that forwards packets and processes traffic based on forwarding decisions made by a control plane that is separate and remote to the device. The forwarding plane of a network device is responsible for forwarding packets out the interfaces on the device. While the control plane of a network device builds a map of the network and makes decisions about how packets are forwarded by the forwarding plane of the device. In a conventional network device, the forwarding plane and the control plane are tightly coupled together. A conventional network device can only forward packets using forwarding decisions made by its control plane. Because of this, a conventional network device is limited to forwarding packets across the network based on the knowledge its control plane has about the network which may limited or faulty. An SDN-enabled device on the other hand is able to forward packets based on forwarding decisions made by a control plane that is separate from the device. This enables the SDN-enabled device to receive forwarding decisions from other devices and applications that may have a much better knowledge and complete topology of the network than itself.

An SDN router translates routing information from a standard routing control plane into forwarding decisions on an SDN-enabled device. This ensures that network traffic is forwarded correctly between legacy network devices and SDN-enabled devices without the need to define a new protocol. SDN routers allow SDN-enabled devices to integrate into an existing network. They interact with legacy protocols and platforms used by existing network devices to control the network and modify the forwarding behaviour of SDN-enabled devices in the network based on control decisions received from the legacy protocols. This intermediate role played by SDN routers enables SDN-enabled devices to be deployed in legacy networks, interoperate and transparently forward traffic to legacy network devices without the relying on the protocols and platforms used by the legacy devices.

Apart from integrating SDN-enabled devices into legacy networks, SDN routers manage and centralise the routing control plane in a network and delegate packet forwarding functions to SDN-enabled devices they control. In a network, routing is the process of delivering a packet to its destination via the best path in the network. This involves relaying the packet through routers along its selected path until it reaches its destination. A conventional router performs two major functions: it builds a map of the network and forwards packets to their destination; the router uses routing protocols to exchange routing information with remote peers and build a topology of the network. This routing information is maintained in the router's Routing Information Base (RIB), this is stored in the form of routes to destinations in the RIB. Routes specify how the router can reach remote destinations in the network and each route contains at least the network address of the remote destination, the next-hop for that destination where the router can forward packets and the cost associated with sending packets via this route. The entries in Forwarding Information Base (FIB) are constructed by copying routes from the RIB, resolving the next-hop for each route because it requires the next-hop with lowest cost and computing the interface through which packets should be forwarded for that route. The router uses the FIB to perform its second function of forwarding packets. When a router needs to forward a packet, it performs a lookup in the FIB to select the interface closest to the next-hop for the destination of the packet. The FIB is optimised for fast lookup of the destination of a packet. This is usually implemented in special hardware known as Ternary Content Addressable Memory (TCAM) which is similar to that used in switching devices for quickly matching Media Access Control (MAC) addresses and forwarding packet out the connected interfaces. The TCAM is different from the hardware used by RIB in a router. TCAM is an expensive and power-hungry component. The requirement of routers to rapidly forward packets to the their destinations necessitates the inclusion of TCAMs in hardware routers, this increases the cost associated with acquiring and running these routers in a network. SDN-enabled devices often use TCAMs for forwarding packets, however, they may also use a different technology to achieve this. SDN routers can offload packet forwarding functions to multiple SDN-enabled devices, this reduces the number of TCAMs dedicated to routing.

The delegation of packet forwarding functions to SDN-enabled devices by an SDN router also provides a greater level of flexibility and control for the network operator. It enables the operator to rapidly make changes to the network topology or the routing control plane with minimal disruption to the network.

In conventional routers, changes to the RIB require that the FIB is refreshed so that packets are forwarded to their destination via the best path based on changes to the RIB. This usually takes time and packet forwarding on the router is degraded during this period. However, since SDN routers offload packet forwarding to SDN-enabled devices. These SDN-enabled devices will continue to forward packets while the FIB is being refreshed thereby minimising the disruption caused by network changes. Delegation of packet forwarding functions to SDN-enabled devices by SDN routers enhances these devices by giving them the capability to forward packets to remote destinations which is a function normally performed by routers. This reduces the number of routers required in a network thereby simplifying the network and reduces the overhead introduced by the routing control plane that would be generated by routers in a network.

RouteFlow[14] is an open-source SDN router project that provides virtualised Internet Protocol (IP) routing services over SDN-enabled devices. It uses an OpenFlow controller application to translate the FIB generated by a software IP routing engine into OpenFlow flow rules. These rules are installed in the forwarding plane of OpenFlow switches. Subsequent traffic received by the OpenFlow switch which matches the rules are forwarded to their destination by the OpenFlow switch bypassing the routing engine and controller.

An OpenFlow switch is an SDN-enabled device that forwards and processes packets based on instructions received from a controller application via a control channel. An OpenFlow controller modifies the forwarding behaviour of an OpenFlow switch by directly manipulating the forwarding plane of an OpenFlow switch. It does this by inserting flow entries into the flow table of an OpenFlow switch via the OpenFlow control channel. A flow entry is an instruction to the forwarding hardware of an OpenFlow switch to process and forward packets based on a set of header fields in the packets. A flow table is maintained by an OpenFlow switch and it is used to store flow entries received from an OpenFlow controller. Packets are forwarded on an OpenFlow switch by comparing their headers against the flow entries in the switch's flow table until a match is found. OpenFlow is further discussed in Chapter 2.

RouteFlow allows OpenFlow switches to integrate seamlessly into conventional networks by collecting Address Resolution Protocol (ARP) tables and FIB generated by existing routing protocols such as Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP). These route tables are converted into OpenFlow rules that are installed in the forwarding plane of the OpenFlow switches. This also allows a set of OpenFlow switches controlled by RouteFlow to behave as a single logical IP router to the rest of the network because these switches forward packets to their destination based on the FIB generated from the RIB maintained by the IP routing engine in the RouteFlow application. RouteFlow has matured from an experimental project into a stable solution that has been deployed in production networks [20][23] and it is still actively improved and maintained.

While RouteFlow delegates the forwarding of packets to connected OpenFlow switches, it does not fully centralise its routing control plane. This is because RouteFlow is unable to build a map of the network by itself rather it depends on an IP routing engine that is separate and remote to the RouteFlow application. Also the lack of centralisation of the IP routing engine with the rest of the RouteFlow application also prevents the remote routing engine from being notified of changes in the forwarding plane that would require an update of the RIB, this makes RouteFlow unable to respond to changes without disrupting the network. The configuration of RouteFlow is a manual and complicated process which requires the use of multiple configuration files, this hinders the flexibility required to make changes in the network and limits the operator's control over the network. The RouteFlow application also introduces an intermediate control protocol used to translate FIB entries into OpenFlow rules and routing protocol traffic to the routing engine, this intermediate control protocol increases rather than decrease the overhead introduced into the network by the routing control plane.

RheaFlow is an SDN router that provides the same capabilities as RouteFlow. It translates the FIB generated by a software IP routing engine into OpenFlow rules that are installed on connected OpenFlow switches. However, RheaFlow addresses the lack of centralisation of the IP routing engine and OpenFlow controller identified in RouteFlow by implementing a design that situates the IP routing engine and the RheaFlow application on the same operating system. This ensures that the routing engine is able to respond to changes in the forwarding plane with minimal disruption in the network and optimises the translation of FIB to OpenFlow rules installed in the forwarding tables of OpenFlow switches. This is achieved by sending FIB updates received from the IP routing engine directly to the OpenFlow controller and converting them to OpenFlow rules thereby eliminating the need for an intermediate message format as used in RouteFlow. RheaFlow also improves upon the complex and manual configuration process observed in RouteFlow by simplifying the configuration process and dynamically responding to changes in configuration state without user intervention. This thesis presents the design and implementation of the RheaFlow solution and evaluates it against some of the problems identified in RouteFlow.

1.1 Document Structure

The rest of the thesis is laid out as follows:

Chapter 2 identifies and discusses the important features in an SDN router. Different SDN routers are discussed and compared to RouteFlow based on these features.

Chapter 3 describes the RouteFlow architecture and highlights the issues identified with it.

Chapter 4 provides a description of the components of the RheaFlow architecture. The RheaFlow implementation is discussed and RheaFlow's design goals are highlighted. The process of converting a FIB to OpenFlow rules is explained.

Chapter 5 describes the testbed for the RheaFlow prototype and discusses its performance during stress tests. RheaFlow is also evaluated against RouteFlow to determine whether it addressed the issues identified with RouteFlow.

Chapter 6 briefly discusses the RheaFlow design experience and challenges encountered during RheaFlow implementation. RheaFlow is also compared to SDN routers discussed in Chapter 2 based on the SDN router features identified. Areas of future work are also highlighted.

Chapter 7 summarises the work undertaken for this thesis.

Chapter 2

Related Work

The SDN paradigm is relatively new in the network architecture and engineering field; because of the potentials SDN has to offer, it is an area that is being actively researched and innovated upon by both academia and the wider industry. The gap in interoperability with legacy networks has been identified as a major obstacle slowing down the adoption of SDN [24]. This is because network operators usually have an obligation to maintain a high level of availability, reliability and quality of service on their networks and the introduction of new and untested SDN technologies could be disruptive thus putting them in breach of service level agreements with their end users. Hence, the introduction of SDN technologies into an existing network requires a careful and gradual deployment process. One of the functions of an SDN router is to enable a gradual migration of segments of a network to SDN while maintaining backward compatibility with the rest of the network.

An SDN router translates routing control plane outputs and information used to modify forwarding behaviour on legacy devices into forwarding decisions on SDN-enabled devices. This ensures that network traffic is forwarded correctly and transparently between legacy devices and SDN-enabled devices without the need to define a new protocol. Decoupling the control plane from an SDN-enabled device in SDN enables an SDN router to focus solely on centralising the routing control plane and maintaining an up-to-date topology of the network while SDN-enabled devices connected to the SDN router handle the forwarding of packets to their destinations. Apart from forwarding packets, an SDN router also enables the SDN-enabled devices to redirect control plane traffic to itself or deliver control plane traffic to a remote control plane.

There are varieties of protocols and standards that can be used to implement SDN. Some of the SDN routers examined in this chapter were implemented using Forwarding and Control Element Separation (ForCES) protocol, other SDN routers examined were implemented using OpenFlow protocol. However, some of these are more widely implemented more than others. ForCES protocol [5] is an SDN specification that defines a template for separating the control plane from the forwarding plane in a Network Entity (NE) to enable flexible processing and forwarding of packets. ForCES enables an SDN architecture by allowing network operators to delegate packet forwarding and per-packet processing functions to specific SDN-enabled devices in a network. An SDN-enabled device dedicated to packet forwarding and processing is called a Forwarding Element (FE) in ForCES and the underlying hardware in a FE is usually dedicated and optimised for per-packet processing and forwarding. In a ForCES based architecture, control logic is maintained by the Control Element (CE) which instructs the FE on how to process packets. The ForCES protocol provides the standard by which the CE must interact and exchange information with an FE. ForCES enables a network operator to set the numbers and combination of CEs and FEs that will form a NE in a network. The NE appears as single monolithic piece to the rest of devices in the network, it also ensures that packet can arrive via any FE in a NE and exit via another FE transparently.

The OpenFlow protocol [13] is a recent SDN specification that defines an Application Programming Interface (API) and communication interface with which the forwarding plane of an OpenFlow switch can be manipulated. Since SDN is the separation of the control plane from forwarding plane to ensure flexible packet forwarding and greater control over the network. OpenFlow enables a control plane application which is usually called a controller which may be remote to the OpenFlow switch to directly send forwarding instructions to the hardware in an OpenFlow switch. The hardware in the OpenFlow switches is optimised to perform fast lookups on packets using information from their header fields against a flow table to determine the interface via which these packets should be forwarded. OpenFlow enables the controller to directly modify the flow table as may be needed to forward packets. Compared to ForCES, OpenFlow is a low-level SDN specification that gives a controller the ability to directly modify the packet processing pipeline in the hardware of an OpenFlow switch which gives a network operator great flexibility over packet processing and granular control in a network. On the other hand, ForCES approaches SDN from a conceptual level as it focuses more on defining and specifying the functions of the elements in an SDN architecture while leaving the implementation on the devices to the network operator. ForCES could be considered a forerunner to the OpenFlow however, OpenFlow is more widely implemented and adopted than ForCES. This tilts the choice of SDN specification towards OpenFlow because one of the concerns of this work is increasing the adoption of SDN by making it easier to add an SDN router to an existing network. This concern is better addressed by using an SDN technology that is more widely adopted and implemented.

The SDN routers discussed in this chapter were also considered with respect to some of the issues identified with RouteFlow and the features provided by these SDN routers. As mentioned earlier in Chapter 1, an important issue is the centralised management of the routing control plane, because distributed management of the routing control plane often complicates the configuration of an SDN router. It also prevents or delays the SDN router from notifying the routing control plane of changes or failures in the SDN-enabled devices. Another issue that was examined is the complexity of configuring and deploying an SDN router into a network. The rapidly evolving nature of large networks requires that network operators are able to quickly modify and deploy resources in order to meet their obligations. Complex and complicated configurations or deployments take time and are prone to errors. Hence, SDN routers should be simple to configure and deploy. Furthermore, the ability for an SDN router to support multiple SDN-enabled devices is also considered important. Since an SDN router delegates packet forwarding to the SDN-enabled devices, if the SDN router can control multiple devices, it greatly increases scalability and may simplify the network topology for a network operator. Interoperability and backward compatibility with existing routing control protocols was considered in this evaluation of SDN routers. This is because SDN routers are deployed in existing networks and need to exchange routing information using existing technologies and routing protocols. The SDN routers were also evaluated on their ability to dynamically discover new SDN-enabled devices, reconfigure themselves and update their control plane logic so that they can forward packets with the newly discovered SDN-enabled devices.

Finally, SDN routers were also evaluated based on whether they required new or separate control protocols and Inter-Process Communication (IPC) channel like RouteFlow. This is because separate control protocols often require that they are processed differently which increases the overhead associated with using these SDN routers in a network. Table 2.1 shows the comparison between the SDN routers evaluated in this chapter.

SDN Router	**SDN Specification**	**Multiple SDN-enabled Device Support**	**Centralised Control Plane**	**Dynamic Discovery and Configuration**	**Simple Configuration**	**Backward Compatibility**	**Separate Control Protocol**
FIBIUM	OpenFlow	No	Yes	Partial	No	Yes	No
SoftRouter	ForCES	Yes	Yes	Yes	No	Yes	Yes
DROP	ForCES	Yes	Yes	Yes	No	Yes	Yes
SDN-IP	OpenFlow	Yes	Yes	Partial	No	Limited	No
Atrium	OpenFlow	Yes	Yes	Partial	No	Limited	No
RouteFlow	OpenFlow	Yes	No	No	No	Yes	Yes

Table 2.1: Comparison of SDN Router Features.

The rest of this chapter examines each SDN router and briefly highlights its features. The limitations and strengths of each SDN router are also discussed.

2.1 FIBIUM

FIBIUM [18] centralises the routing control plane on a cheap commodity Personal Computer (PC) and partially delegates packet forwarding to an OpenFlow switch it controls. The FIB used by the OpenFlow switch to forward packets are constructed from the RIB maintained by an IP routing software on the PC. An OpenFlow controller application on the PC monitors for changes to the FIB and selects a few FIB entries to be installed on the OpenFlow switch. The FIB entries installed on the OpenFlow switch in the FIBIUM setup are selected based on traffic to the destinations reachable via these FIB entries. FIBIUM does not install the full forwarding table from the IP routing software because it was designed to be used with low-end OpenFlow switches with limited OpenFlow rule capacity. According to [4, 6, 9] a cheap commodity PC running open source IP routing software does not offer the performance required for a large carrier network because it lacks the port density and specialised hardware for forwarding packets. The FIBIUM architecture consists of three main components which are:

- An OpenFlow switch which provides the port density and packet forwarding performance that is lacking on a commodity PC.
- An open source IP routing software that enables FIBIUM to receive and send RIB updates which are used to build the FIB table.

- RouteVisor, the main component of the FIBIUM architecture. It serves as a controller for the OpenFlow switch and determines which set of forwarding entries from the FIB table populated by the routing protocols from the IP routing software will be installed on the switch. It does this by collecting traffic statistics on the most used FIB entries and installing those on the OpenFlow switch.

Although, FIBIUM aims to achieve similar levels of performance to that obtained from commercial hardware routers by using a low-end OpenFlow switch and a cheap commodity PC running open source IP routing software. As an SDN router, its design does not completely offload packet forwarding to the OpenFlow switch because packets that can not be forwarded using the forwarding entries on the OpenFlow switch are sent to the PC to look up their destination using the full FIB maintained on the PC. This is because the OpenFlow switches targeted by FIBIUM are not capable of storing a large number of FIB entries that could be generated by BGP in a large network. This is a good compromise as it ensures that the packets that can not be forwarded using the forwarding entries on the OpenFlow switch are still delivered to their destination via the forwarding table maintained on the PC. This is not a concern considered in RouteFlow or RheaFlow that is being proposed in this work. This is because the RheaFlow is able to work with a wide range of OpenFlow switches. In a situation where the OpenFlow switches are not capable of storing large flow entries, CacheFlow [8] could be used.

The IP routing software and RouteVisor are installed on the PC. RouteVisor creates virtual interfaces for the interfaces on the OpenFlow switch and uses Virtual LAN (VLAN) tags to map the virtual interfaces on the PC to the physical interfaces on the switch thereby making the interfaces on the OpenFlow switch visible to the IP routing software. Packets are either forwarded through FIBIUM by the OpenFlow switch or by the PC. Packets forwarded by the OpenFlow switch already have their destination prefix installed in the forwarding table of the OpenFlow switch while packets forwarded by the PC do not have any forwarding entries in the OpenFlow switch. These packets are delivered to the PC via the OpenFlow control channel between the OpenFlow switch and RouteVisor on the PC. These packets are checked against the full FIB table maintained on the PC to determine their destination and forwarded by the PC.

As an SDN router, FIBIUM centralises the routing control plane but partially delegates forwarding to the OpenFlow switch. It however, provides backward compatibility with a variety of existing IP routing control protocols by using an IP routing software capable of running multiple legacy IP routing protocols like BGP and OSPF to build and maintain its topology and knowledge of the network. FIBIUM does not require or define a separate control protocol as this only increases the overhead associated with running the router in a network. Another issue identified with FIBIUM is that it is unable to support multiple OpenFlow switches. FIBIUM's inability to delegate packet to multiple OpenFlow switches makes it less scalable and could possibly increase the time spent provisioning and configuring the network. This also relates to the issue of the simplicity of its configuration and dynamic discovery of an OpenFlow switch. While, FIBIUM dynamically configures the OpenFlow switch in response to changes in the FIB maintained on the PC, it is not possible to rapidly expand the topology by dynamically discovering and adding a new port on the OpenFlow switch without disrupting forwarding on other ports on the OpenFlow switch as it requires reloading RouteVisor. Coupled with the lack of multiple OpenFlow switch support, this configuration could be complex and time consuming in a network with multiple instances of FIBIUM. However, the dynamic configuration of virtual interfaces by RouteVisor in FIBIUM was adopted in RheaFlow to improve the static and manual configuration process in RouteFlow.

2.2 SoftRouter Architecture

SoftRouter Architecture [12] design applies a ForCES based approach that enables the centralisation the routing control plane on general purpose server is known as the CE. The CE is capable of running existing routing protocols such as BGP or OSPF and it uses the control logic and topology knowledge to determine forwarding behaviour of SDN-enabled devices associated with it. The SoftRouter archictecture is made of following entities:

- A NE which comprises of CE and any number of FEs. The FE establishes a connection with the nearest CE it discovers in a network and proceeds to forwarding packets on the links to other FEs in the network it has detected based on the logic supplied by the CE.
- A discovery protocol to be used by FE to find and establish a connection

to the nearest CE.

- A FE/CE control protocol for communication between the FE and CE once a connection has been established.

In SoftRouter, an SDN-enabled device associated to a CE is called a FE. This FE is usually a commercial router or switch that has been stripped of its control logic. It forwards packets based on instructions from an associated CE. A CE may be associated with a set of FEs or just a number of ports on a single FE, this association is called a NE. A NE makes a combination of CE and FEs appear as a single monolithic device to the rest of the network. In the SoftRouter architecture, the NE is the SDN router, the routing control plane is centralised in the CE while packet forwarding is handled by the FE which is the SDN-enabled device based on instructions received from the CE. The routing control plane is centralised in the CE and supports multiple IP legacy routing protocols while forwarding is handled by the FE in a NE based on the SoftRouter architecture. This SoftRouter approach ensures a complete separation of the control plane from the forwarding plane as specified for an SDN router. In a SoftRouter NE, multiple FEs may be associated with a CE ensuring multiple SDN-enabled device support. The discovery and configuration required to associate FEs to CE for a NE is dynamic and happens with minimal intervention from the user in the basic configuration. However, association and subsequent communication between a CE and FE requires a separate control protocol which increases the overhead associated with a SoftRouter NE in the network.

2.3 DROP: Distrubuted Software Router Project

DROP [2] uses open source software IP routers to implement a distributed SDN router architecture. The implementation is based on the ForCES standard, the open source IP software router is a version of Quagga modified for the DROP architecture. The DROP architecture is composed of the following building blocks:

- CEs distributed across the network and dedicated to control, management and routing decisions.

- FEs also distributed across the network, dedicated and tuned for forwarding IP packets.

- An internal/private network that connects the CEs and FEs together.
- CE controllers that manage and exchange routing information between the CEs and FE controllers that maintain and manage FIB states and configuration across the FEs.

The routing control function is centralised on a Linux machine running the modified Quagga routing engine. This machine is called the CE and it uses IP routing protocols like BGP or OSPF to build a topology of the network, it modifies the forwarding behaviour of the SDN-enabled devices using Netlink [17]. An SDN-enabled device in DROP is called an FE. It is also a Linux machine running the version of Quagga modified for DROP and it focuses solely on forwarding packets and receives forwarding instruction from the CE via the netlink protocol. A combination of FEs and CE forms a NE in DROP which is seen as a monolithic routing entity by other devices in the network. DROP[2] is a distributed SDN router implemented based on ForCES guidelines using open source IP software routers. DROP architecture enables the distribution and spread of the CE and FEs that form a NE physically and geographically across a network while presenting a single logical view to other devices in the network. This makes DROP a scalable SDN router with a centrally managed routing control plane that offloads packet forwarding to multiple SDN-enabled devices distributed physically and geographically across a network. However, the use of an overlay private network for exchanging forwarding information between the CE and FE would add unnecessary complexities to a network and the overhead associated with the private network and its netlink traffic could degrade the overall performance of the network.

2.4 SDN-IP

SDN-IP [10] forwards packets with OpenFlow switches using a FIB constructed from the RIB generated by the BGP routing protocol. The routing control plane is centralised in the SDN-IP application while the OpenFlow switches forward packets to their destination. SDN-IP is run as an application on Open Network Operating System (ONOS), an OpenFlow controller that provides a high level abstraction on top of OpenFlow. SDN-IP also acts as an iBGP peer to an upstream BGP peer. SDN-IP builds its RIB by receiving BGP route updates from its upstream peer, it then constructs a FIB for the destinations reachable via the routes in the RIB. ONOS modifies the packet processing pipeline of the OpenFlow switches via the OpenFlow control channel using the FIB. This enables the OpenFlow switches to forward packets to their destinations by delivering them to the next-hops of their destination via the ports on the OpenFlow switch directly connected to the next-hops. The OpenFlow rules installed on the OpenFlow switches by ONOS also ensures BGP traffic is delivered to SDN-IP's iBGP peer. This iBGP peer only receives route advertisements it does not have the capability to send route updates to remote peers.

SDN-IP only offers limited backward comptability as it does not support any IP routing control protocol beyond BGP. According to [10], SDN-IP can only generate forwarding entries for 15,000 Internet Protocol version 4 (IPv4) routes and it does not support Internet Protocol version 6 (IPv6). This can be problematic in a large network and makes it unsuitable for a network that aims to carry IPv6 traffic. While it is possible to offload packet forwarding to multiple OpenFlow switches, the application can not dynamically reconfigure the OpenFlow switches in response to network changes such as a change in the address of the next-hop to a destination that is directly connected to a port on an OpenFlow switch. Also, SDN-IP requires an understanding and familiarity with the ONOS configuration process which is not intuitive.

2.5 Atrium

Atrium [21] also forwards packets with OpenFlow switches using a FIB constructed from a RIB generated by the BGP routing protocol. The Atrium design extends the SDN-IP application by the upgrading the BGP routing control plane application used to send and receive route updates. This eliminates the need for a separate upstream BGP peer in a network in which Atrium has been deployed. Atrium is run by ONOS which converts the FIB constructed from routing table into OpenFlow instructions for the OpenFlow switch.

Atrium addresses the issues of dynamic reconfiguration of the OpenFlow switch port in response to network changes and limited translation of the routing table into OpenFlow rules identified with SDN-IP. However, it still only supports the BGP routing protocol. The Atrium configuration process is complicated as it requires that the BGP peers that will be exchanging route updates be included at configuration time. Subsequent BGP peers that are required to exchange routes with Atrium after it has been started requires a restart. This disrupts packet processing on the OpenFlow switches. Atrium like RouteFlow has been deployed in production networks and is still being actively developed. However, it was discovered halfway through the development of RheaFlow.

2.6 Conclusion

While the SDN routers examined in this chapter are not an exhaustive list, they were useful in evaluating features important in an SDN router such as:

- An SDN implementation that is widely supported, to increase its chances of adoption by network operators.
- Ability to delegate forwarding to multiple SDN-enabled devices for scalability.
- A centralised management of the routing control plane in conformance with the SDN paradigm.
- Dynamic discovery of SDN-enabled devices, reconfiguration of SDN-enabled devices in response to network changes and a simplified configuration that allows for quick changes in response to rapidly evolving network demands with minimal disruption.
- Backward compatibility with existing routing control protocols.
- The absence of a separate control protocol which likely increases overhead.

As Table 2.1 indicates, RouteFlow lacks some features important to an SDN router. RouteFlow was chosen for this work and is described in the next chapter.

Chapter 3

Overview of Routeflow

RouteFlow [14] is an open source SDN routing solution that uses the control logic and topology knowledge of IP routing protocols to connect separate IP subnetworks and forward IP traffic over OpenFlow switches. A set of OpenFlow switches connected to RouteFlow are seen as a single logical router. This makes the OpenFlow behaviour transparent to the rest of the network. The new SDN routing solution presented in this thesis was based on improvements made to RouteFlow. RouteFlow was chosen as the target solution for improvement based on these reasons:

- Unlike other SDN routers examined previously, RouteFlow is backwards compatible with existing IP routing protocols like BGP, OSPF and Routing Information Protocol (RIP). It also provides a platform for supporting other routing protocols while others support at most two IP routing protocols. RouteFlow's design ensures it does not depend on any specific open source routing software unlike other SDN routing solutions.
- RouteFlow is a stable and widely recognised project in the SDN area and it has been successfully deployed in production networks [20, 23].
- RouteFlow is implemented with OpenFlow, the most recognised specification of SDN. Improving solutions that enable OpenFlow switches to interoperate with legacy network devices would bolster the case for SDN adoption by reaching a larger audience.

This chapter provides an overview of the RouteFlow architecture, a brief discussion of the various components used in the design, how the control logic and knowledge of the network topology obtained from IP routing protocols are used to modify forwarding behaviour on OpenFlow switches. Finally, components and operations in the RouteFlow architecture that could be improved are discussed.

3.1 RouteFlow Architecture

RouteFlow is designed to collect routing and topology information from multiple IP routing engines, and modify the forwarding behaviour of multiple OpenFlow switches. It achieves this by running the multiple routing engines in Virtual Machine (VM)s and multiplexing the interfaces used for forwarding on the multiple OpenFlow switches into virtual interfaces on a single virtual switch. The virtual switch presents the interfaces from different OpenFlow switches via the virtual interfaces as being from a single OpenFlow switch to the controller. In RouteFlow, the RFClient component collects the FIB generated by the virtualised IP routing engines and sends these to RFServer which determines how the forwarding entries should be installed on the OpenFlow switches based on the mappings between the virtualised IP routing engines and the OpenFlow switches. The forwarding entries are sent to RFProxy, the OpenFlow controller application that converts the FIB sent by RFServer into OpenFlow rules and installs them on the OpenFlow switches. RouteFlow components communicate with using RFProtocol, an internal RouteFlow message format over an IPC channel. Figure 3.1 shows the relationship and interaction between the components in RouteFlow. The components of the RouteFlow architecture are further outlined in the sections below.

3.1.1 RFClient

RFClient is the RouteFlow component that runs in the virtualised routing engine which is a Linux VM running an open source IP routing software such as BIRD or Quagga. The IP routing software uses routing protocols like BGP and OSPF to exchange routing information with other routers and builds a RIB from this information. The FIB table is generated from this RIB by the Linux network stack. RFClient which is implemented in C++, listens and collects the ARP and FIB tables maintained by the Linux kernel from the VM using the Netlink Linux API. The FIB table and subsequent updates to the table are sent via an established IPC channel to RFServer.

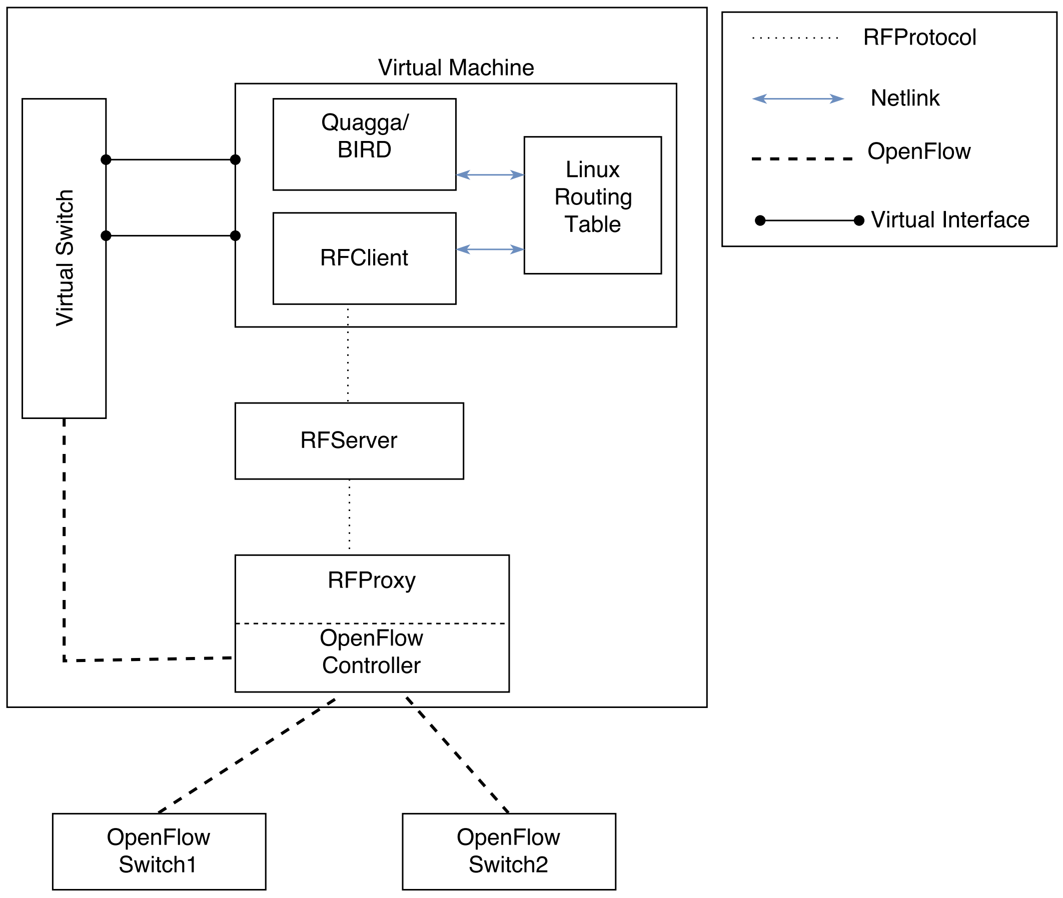

Figure 3.1: The RouteFlow Architecture

3.1.2 RFServer

RFServer is where the intelligence of the RouteFlow architecture resides. It manages the mapping of an OpenFlow switch to a virtualised routing engine. Based on this mapping, it determines the OpenFlow switch that will use the FIB entries from a virtualised IP routing engine to forward packets. It maps the virtualised IP routing engine running in RouteFlow to interfaces of the connected OpenFlow switches using the virtual bridge in the architecture. RFServer receives FIB updates sent by RFClient. It may modify these FIB updates if necessary before sending them to RFProxy via an established IPC channel. RFServer is also responsible for setting the default OpenFlow rule-set to be installed on the OpenFlow switches. These rules ensure that OpenFlow switches forward routing protocol and other network control traffic to the RouteFlow virtual switch via RFProxy. RouteFlow virtual switch then forwards these out its virtual interfaces to the virtualised IP routing engine.

3.1.3 RFProxy

RFProxy is the OpenFlow controller application that modifies forwarding behaviour on the OpenFlow switches. It receives FIB updates from RFServer, converts them to OpenFlow rules and installs them on the required OpenFlow switches. It notifies RFServer when OpenFlow switches are added to the topology and configures them using the default OpenFlow rule-set received from RFServer. The RFProxy component was designed to be replaceable with different OpenFlow controller platforms. This was to ensure that RouteFlow is not tied to a particular OpenFlow controller platform. However, this adds complexity to RouteFlow's configuration as it requires an internal representation of OpenFlow rules in the form of the RFProtocol message format, which does not depend on the OpenFlow controller platform being used. This thesis focuses on a fork of RouteFlow called Vandervecken [1]. The RFProxy component in Vandervecken is run on Ryu, a Python based OpenFlow controller platform.

3.1.4 RouteFlow Virtual Switch

The virtual switch is used to virtualise the physical interfaces of OpenFlow switches that will be forwarding IP traffic to the IP routing engine on the VMs. It is an Open vSwitch instance [15] that is connected to the virtualised IP routing engine with virtual interfaces. The virtualised IP routing engine forward and receive control traffic through these virtual interfaces. The virtual switch is configured with OpenFlow rules to forward any control traffic it receives from its virtual interfaces to the corresponding OpenFlow switch and vice versa. This makes the OpenFlow protocol traffic transparent to the virtualised IP routing engine on one hand and routing protocol traffic transparent to the OpenFlow switches on the other hand. From the perspective of the virtualised IP routing engine, the OpenFlow switches are directly connected.

3.1.5 RFProtocol

RFProtocol is an internal RouteFlow message format used to exchange topology change and configuration information between the components of RouteFlow. RFProtocol messages can be divided into two broad categories. The first category of RFProtocol messages are used to configure RouteFlow and notify RFServer of changes to the topology and state of the OpenFlow switches managed by the OpenFlow controller. Some of these RFProtocol messages are:

- PortRegister: sent by RFClient to RFServer. It is used to notify RF-Server that a port on the virtualised IP routing engine has been detected and is ready for use.
- PortConfig: used to modify the configuration of a port on the virtualised IP routing engine. It is usually sent from RFServer to RFClient.
- DatapathPortRegister: used by RFProxy to notify RFServer that an interface on an OpenFlow switch is ready for use.
- DatapathDown: used by RFProxy to notify RFServer that an OpenFlow switch has lost connection with the controller.
- VirtualPlaneMap: notifies RFClient of the mapping between the virtual switch ports and the ports of the OpenFlow switches.
- DataPlaneMap: notifies RFProxy of the mapping between the ports of the connected OpenFlow switches and the ports on the virtual switch.

The second category of RFProtocol messages are called RouteMod messages. These are used to deliver updates and changes to the FIB table from RFClient to RFServer. The OpenFlow rules to forward IP traffic over the OpenFlow switches are converted from the RouteMod messages received by RFServer.

3.1.6 OpenFlow

OpenFlow is an API used to modify the forwarding behaviour on OpenFlow switches. It also establishes a communication channel over Transmission Control Protocol (TCP) between the controller and OpenFlow switches. In RouteFlow, RouteMod messages received by RFProxy are converted to OpenFlow messages. These OpenFlow messages are constructed as a set of matches and actions. The matches are a combination of fields in the header of packets used to group packets into a flow. The actions are the operations to be performed on a packet if it matches. Some of the header field matches and actions performed on packets required to enable OpenFlow switches forward IP traffic are listed in Table 3.1.

Matches	Actions	Type
IPv4 Dst= 192.168.0.0/24	Output: Port 1	Forwarding IPv4
IPv6 Dst= 2001:0db8:0:f101::/64	Output: Port 2	Forwarding IPv6
Ingress Port=3, IPv4 Dst=192.168.0.1, MAC Dst=0c:84:dc:54:ab:c6	Set MAC Dst to: 0c:84:dc:54:eb:d6, Output: Port 1	Routing IPv4
Ingress Port=2, MPLS Label=14	Pop MPLS label, Output: Port 1	MPLS forwarding

Table 3.1: Example of Matches and Actions Performed by OpenFlow to Forward Packets.

3.2 RouteFlow Configuration and Operation

3.2.1 Starting RouteFlow

A basic RouteFlow setup requires at least one configuration file. This file is used to specify the mapping between the virtualised IP routing engines and the physical OpenFlow switches. The format of the configuration file is shown below:

```
vm_id,vm_port,ct_id,dp_id,dp_port
12A0A0A0A0A0,1,0,99,1
12A0A0A0A0A0,2,0,99,2
```

Each row maps the port (`vm_port`) on a virtualised IP routing engine identified by `vm_id` to port (`dp_port`) of an OpenFlow switch identified by `dp_id` that is connected to a controller with `ct_id`. To start RouteFlow, each of the components are started separately. The startup sequence is described below:

1. RFServer is started first. It loads the configuration file into memory and waits for other components to establish a connection via the IPC channel. The Vandervecken fork has been modified to start the other components after RFServer is started.

2. RFProxy is started and it establishes a connection to RFServer via the IPC channel. Once an OpenFlow switch has connected to the controller, RFProxy notifies RFServer of the ports on the OpenFlow switch with a RFProtocol DatapathPortRegister message. RFServer responds with configurations and OpenFlow default rule-sets if a map exists for that OpenFlow switch port in the configuration file.

3. RFClient is the last component to be started, this is completed alongside

configuration of the virtual switch. RFClient establishes a connection to RFServer and notifies it of the ports available on the VM by sending RFProtocol PortRegister messages. This enables RFServer to complete the rest of the configuration by notifying RFClient of the OpenFlow switch ports to RFClient association. Once this is done, RouteFlow is ready to forward IP traffic.

3.2.2 Installing a New Network Route

Traffic between IP subnets is forwarded on the OpenFlow switches using the OpenFlow rules installed on them. These OpenFlow rules are generated from updates to the FIB table in the kernel of the VM by the open source IP routing software of choice. The process by which a network route is converted to OpenFlow rules is enumerated below:

1. A remote legacy router sends a route update to the IP routing engine in the VM using the configured routing protocol of choice.
2. The OpenFlow switch recognises the routing protocol traffic carrying the route update and forwards the packets to the controller.
3. The route update is sent to the IP routing engine via RFClient. The IP routing engine makes a decision on whether the new route should be installed in the routing table of the VM. If the new route is installed in the VM's kernel, RFClient sends a RouteMod message containing the address, prefix, gateway address and MAC address of the gateway to RFServer.
4. RFServer determines the OpenFlow switch that the route update should be installed on and forwards the RouteMod message to RFProxy. The RouteMod message is converted to OpenFlow rules by RFProxy and installed on the required OpenFlow switch. Subsequent traffic arriving at the OpenFlow switch that matches the installed route are forwarded directly to their destination.

3.2.3 RouteFlow Operation Mode

RouteFlow has two modes of operation, namely *slow path* and *fast path*. In *slow path* mode, all network control traffic received by the OpenFlow switches for the IP routing engine in the VM are encapsulated as OpenFlow packet-in messages and sent to RFProxy. These are sent by RFProxy as OpenFlow packet-out messages to the virtual switch which forwards them to the IP routing engine via the virtual interfaces. This adds some latency to the delivery time of control traffic transiting through the OpenFlow switches. *Fast path* mode often requires an extra interface on the machine running RouteFlow and a port on the OpenFlow switch. An ethernet link is established between the OpenFlow switch and the machine using extra ports. Essentially, *fast path* is a faster way of delivering packets to and from the virtualised IP routing engine by using the physical link between the OpenFlow switch and the machine running RouteFlow. Two extra configuration files are required for *fast path*, these are *rffastpath.csv* and *dp0links.csv. Rffastpath.csv* maps the OpenFlow switch port that will be used for *fast path* to the interface designated for *fast path* on the RouteFlow machine. While *dp0links.csv* defines the OpenFlow port number to be assigned to the interface designated for *fast path* on the RouteFlow machine. RFServer uses this configuration file to configure the OpenFlow switch and virtual switch to forward network control traffic to the virtualised IP routing engine bypassing RFProxy. *Fast path* reduces latency introduced by network control traffic and increases throughput in the network.

3.3 Identified Flaws in RouteFlow and Possible Optimisations to the RouteFlow Design

The use of multiple virtualised IP routing engines allows RouteFlow to gather topology and control plane information from the perspective of many IP routing engines. This gives RouteFlow a more accurate view of the network. It also reduces the amount of hardware used in a network. However, the use of multiple IP routing engines complicates RouteFlow's configuration and adds complexity to the network configuration. Complex configuration processes are error prone and time consuming. This discourages customisations and slows down network changes and this would be undesirable in a large network with dynamic network traffic. The use of virtualised IP routing engines in RouteFlow leads to nested virtualisation. If RouteFlow itself were to be virtualised, the nested virtualisation could be complex to manage. Nested virtualisation also introduces potential performance and security issues. The solution proposed to this issue by RheaFlow is to reduce the number of components involved and the amount of configuration required. This is further discussed in the next chapter.

Another design flaw that complicates the configuration process in RouteFlow is the use of multiple configuration files for *fast path* mode and a single file for *slow path* mode. These configuration files are created in Comma-separated Values (CSV) format, which is not easily readable for humans and increases the chances of misconfiguration. The virtual interfaces in RouteFlow are also manually configured and statically mapped during the configuration process. This prevents failure events on the OpenFlow switch ports from being replicated to the virtualised IP routing engine. This prevents the IP routing engine from updating its RIB in response to an issue on its forwarding plane which is the OpenFlow switch port. Also new ports on the OpenFlow switch can not be added without restarting RouteFlow and configuring a virtual interface for it. These are addressed in this project by letting RheaFlow configure the virtual interfaces and dynamically map them to the OpenFlow switch ports. RheaFlow also adopts the use of YAML Ain't Markup Language (YAML) configuration file format to make the configuration readable for humans and parsable for the application.

The VM identifier `vm_id` required in RouteFlow's configuration is the MAC address of the interface used for RFProtocol IPC between RFClient and RFServer. The user has to confirm which interface is being used for RFProtocol IPC on the virtualised routing engine before starting RouteFlow. This process is error prone and could be removed from the configuration process. Also, `vm_id` has to be included in the configuration file before RouteFlow is started. It can not changed be while RouteFlow is still in operation as RFClient will be unable to communicate with RFServer. The RFProtocol IPC channel between RFClient and RFServer is important for RouteFlow's proper functioning. A failure of the IPC channel would render RouteFlow non-functional and cause disruption in the network.

RFProtocol and the IPC channel used to communicate between the various RouteFlow components, adds some overhead while performing certain operations or forwarding control traffic. RFProtocol is an important component of the RouteFlow design as it connects RFClient which is running in a VM to RFServer on the host machine running RouteFlow. As mentioned earlier, a design that moves the IP routing software out of the VM into the same machine as RFServer makes it easier to eliminate RFProtocol and reduce control traffic overhead. This design approach is explored further in RheaFlow.

Fast path mode in RouteFlow improves latency and minimises the overhead of OpenFlow control channel by sending network control packets to the virtualised IP routing engine directly from the OpenFlow switches. This reduces the number of packets handled by RFProxy and minimises the risk of overwhelming RFProxy with many packets. In the proposed RheaFlow design, network control traffic is delivered directly to the IP routing engine without any intervention from the controller. An OpenFlow controller platform receives the route updates from the IP routing engine and listens for network changes using the Netlink Linux API. These are then converted to OpenFlow modifications without any intermediate protocol.

Chapter 4

RheaFlow Design and Implementation

RheaFlow is an SDN routing solution that connects separate IP subnetworks and forwards IP traffic over OpenFlow switches. This is achieved by converting the FIB obtained via routing protocols on an open source software IP routing engine, and ARP tables from the Linux kernel into OpenFlow rules that specify forwarding actions on OpenFlow switches. The functionalities and capabilities provided by RheaFlow are similar to those provided by RouteFlow. Also, RheaFlow's design, architecture and operations are influenced by RouteFlow. However, RheaFlow seeks to improve the routing and forwarding of IP traffic over OpenFlow switches by providing solutions to some of the flaws identified in RouteFlow in the previous chapter and [16].

The components used in RheaFlow's design and the architecture are described in this chapter. Some of the considerations and constraints considered during the design process are also discussed. Finally, the process of converting IP route information to OpenFlow rules and modifying OpenFlow rules in reaction to network changes are discussed.

4.1 RheaFlow Design Goals

Some of the considerations and constraints guiding the RheaFlow design are listed below:

- The configuration and deployment process should be simple and intuitive: This is achieved by reducing the number of configuration steps and processes involved in starting the SDN router. This involves hiding the complexities of the application from the user for simple configurations. Also, using configuration file formats that are intuitive and readable for humans on one hand and easily parsable for machines on the other hand.

- Simplifying the architecture: In RouteFlow's architecture, the IP routing software that provides the IP control logic used to modify forwarding behaviour on the OpenFlow switches is configured in a VM that is a guest on the machine running the RouteFlow application. The information provided by IP routing software in the VM could be provided by an IP routing software running on the same machine as the OpenFlow controller. This simplifies the architecture.
- Reduce overheads associated with the conversion of the FIB into OpenFlow rules: In RouteFlow, FIB updates are converted into an intermediate message protocol on RFClient and sent via an IPC channel before they are converted to OpenFlow by RFServer. This adds unnecessary overhead to the conversion process.
- Reduce the number of OpenFlow messages sent to the controller by using *fast path.*
- An application and platform independent data-interchange format and interface that can be easily modified and extended to receive other network control information such as Access Control List (ACL) and firewall rules from network devices other than routers.
- An SDN router architecture with modular components that are easily customisable and extended to provide new features and functionalities.

4.2 RheaFlow Components

The tools and components used in the RheaFlow design and how they fit into the architecture are discussed in this section.

4.2.1 Ryu

Ryu [22] is an open source component-based SDN application development framework written in Python. It is an OpenFlow controller platform that provides robust APIs and libraries that can be used to write network control applications. These applications are used to control and manage a network together with the devices in it. The design of the Ryu framework provides a platform on which separate SDN applications performing different functions can cooperate and communicate with each other seamlessly. These applications can also be integrated with existing components in Ryu. Existing components can also be modified to cooperate or interact with the applications. The components of Ryu relevant to RheaFlow include the OpenFlow controller which manages the OpenFlow connection to the OpenFlow switches. It sends OpenFlow messages to the OpenFlow switches and parses the OpenFlow messages sent by the OpenFlow switches. Ryu-manager is the main executable within Ryu's architecture and is used to start and run the needed applications. By default Ryu-manager sets up the OpenFlow Controller to listen on the specified address and port for connection from OpenFlow switches when it is executed. Ryu-manager uses the core processes component to manage the messaging between the running applications and event notification.

Ryu has multiple northbound protocol libraries for parsing and processing existing network protocols. It also has multiple southbound protocol libraries for managing other SDN specifications including OpenFlow. Ryu also has a Web Server Gateway Interface (WSGI) component and Representational State Transfer (REST) interfaces for running applications written in Python in server mode. This functionality provided by the WSGI and REST component of Ryu was utilised in RheaFlow's design. The diagram in Figure 4.1 further describes the Ryu Archictecture. The main component in RheaFlow's architecture is the Ryu controller. Other components used in the design of RheaFlow are totally dependent on Ryu. Ryu is used as the base platform for RheaFlow's architecture for these reasons:

- The Ryu framework ensures that new southbound protocols or SDN specifications are interoperable with existing network technologies. It supports multiple southbound protocols like Network Configuration Protocol (NETCONF) and OpenFlow Configuration and Management Protocol (OF-CONFIG) for managing SDN-enabled devices. It also supports multiple northbound protocols like BGP out of the box. The versatility offered by the Ryu framework makes it easy to modify RheaFlow to support other SDN specifications apart from OpenFlow should the need arise.

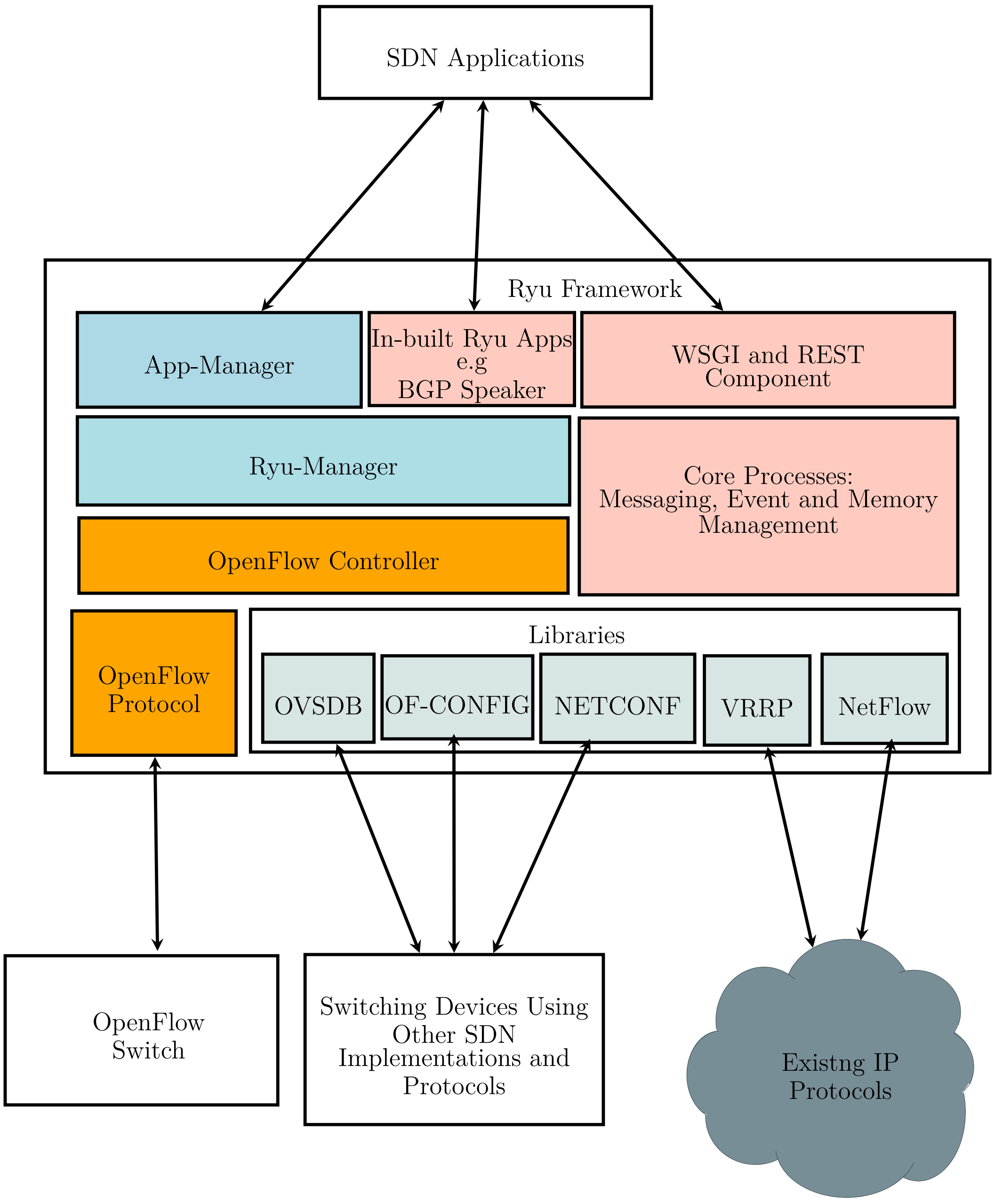

Figure 4.1: Ryu Framework Architecture

- The Ryu OpenFlow controller has been tested with OpenFlow switches from different vendors and it supports multiple OpenFlow versions. This is necessary to prevent a mismatch between the OpenFlow features used by RheaFlow and those supported by the OpenFlow switches. This ensures that OpenFlow instructions sent from the controller are interpreted uniformly on OpenFlow switches from different vendors.
- Ryu maintains backward compatibility for deprecated features in newer

releases. This ensures RheaFlow is not pinned to a specific version of the Ryu framework.

- The Ryu framework is written entirely in Python. It will also run applications and scripts written in Python. The use of a single language greatly increases the ease and speed of development.

Ryu Applications

Applications are used to implement the network control logic and operations performed by Ryu. They are Python modules that Ryu executes to manage and control network behaviour. A Ryu application is a Python class object that is a subclass of the *ryu.base.app_manager.RyuApp* class specified in the Ryu API. Applications are run as single-threaded entities in Ryu and multiple applications communicate with each other using events. When applications are started by Ryu-manager, internal Ryu applications like the OpenFlow controller are also started. Each application is launched in a single thread with an event queue. Each application responds to events in its queue in First In, First Out (FIFO) order and performs the necessary action. The Ryu application model in Figure 4.2 describes how events are handled by an application. The event driven programming model used in Ryu makes it possible for applications to send messages to each other without blocking. The ability to process messages asynchronously between applications is essential in networking programming because it ensures applications are not starved of of system resources. The Ryu API provides an event class which can be used by applications to message other applications. The Ryu API also provides *event_handler* objects for applications to receive and process events asynchronously. The RheaFlow design leveraged the ability to run multiple applications performing different functions on a single Ryu instance. The event related classes provided in the Ryu API were extended to fit the requirements of RheaFlow. These are further discussed and explored in the RheaFlow implementation section.

4.2.2 Netlink

Netlink [17] is a Linux network configuration and management API. It is used for configuring the network control plane in user space on a Linux operating system. The netlink API defines a set of protocols that enables the forwarding plane, which is managed by the kernel in the Linux network stack, to interact with control plane components, which usually operate in the user space of

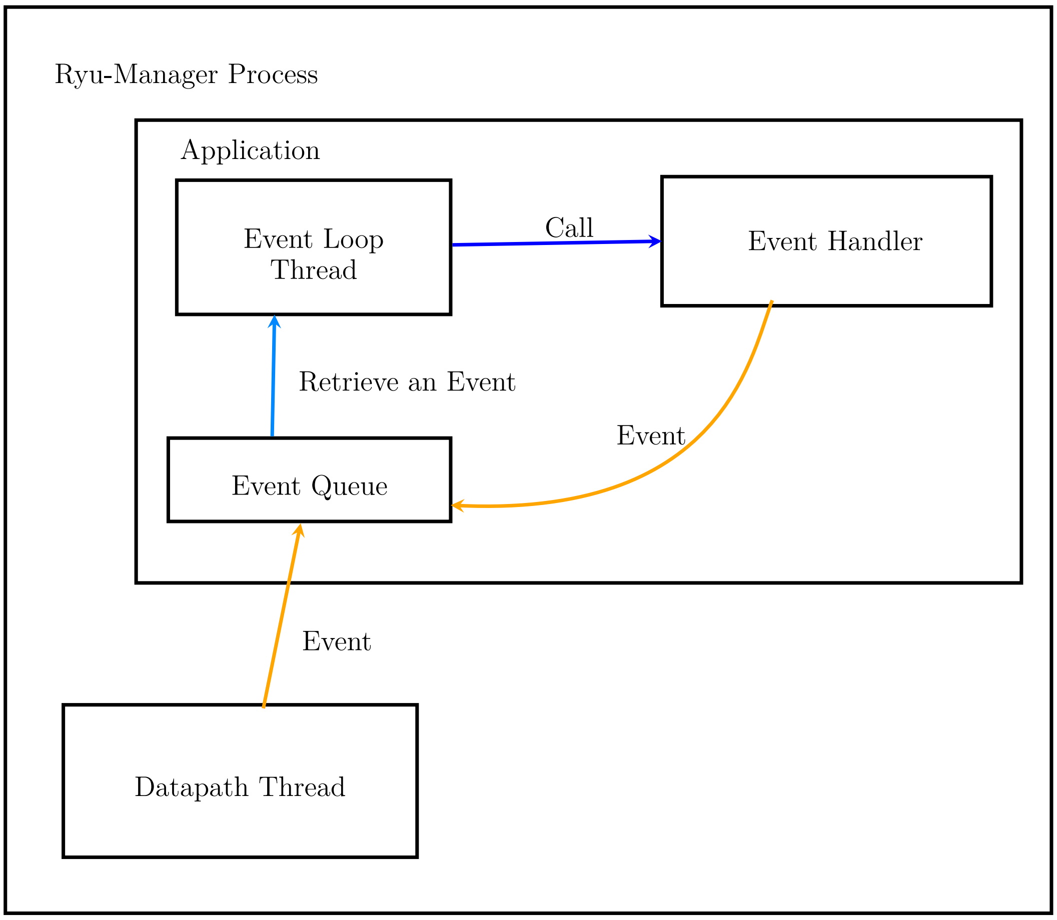

Figure 4.2: Ryu Application Model

the Linux operating system. Apart from providing an interface with which network related applications running in the user space of a Linux operating system can receive configurations and communicate with the forwarding plane in the kernel. Netlink is also used as an IPC mechanism between network applications running in user space in the Linux operating system.

While there are at least nine netlink protocols according to [7], two of these are relevant to RheaFlow. These two protocols are the NETLINK_ROUTE and NETLINK_ROUTE6 protocols. Each of these protocols have different message families; however, only the link layer, address and neighbour message families of both protocols were used in this project. In RheaFlow these messages were used to obtain neighbour table entries, configure and monitor the interfaces on the Linux machine. The netlink messages were processed and recieived by RheaFlow using Pyroute2 [19], a netlink and Linux network configuration library written in Python. In RheaFlow's architecture a Ryu application keeps the link layer, address and neighbour information for the interfaces on the system. These are updated by another application written using the Pyroute2 library. The interaction between netlink and the rest of RheaFlow are further explained in Section 4.3 of this chapter.

4.2.3 YAML

YAML is a human readable data serialisation language. It enables data structures to be stored or transmitted across a network in a format which can be reconstructed by a computer without modifying the data structure. YAML's syntax easily maps to the common data types used in Python. It also presents the data in a format that is familiar and understandable for humans. YAML was used in RheaFlow to provide a configuration system which is intuitive for the operator and also provides the configuration data to RheaFlow using the data types available in the Python language. The configuration process in RheaFlow is further discussed in Section 4.3.1.

4.2.4 JSON

JavaScript Object Notation (JSON) is an open standard light-weight interchange data format that is easy for computers to parse and generate. It is used in RheaFlow to exchange route information between legacy IP routing software and RheaFlow without having to specify new protocols and message format. The details are further discussed in Section 4.3.2.

4.2.5 IP Routing Daemon

BIRD[11] is an open source IP routing engine daemon for Unix like systems. It is used to run the IP routing protocols that exchange routing and topology information with other legacy routers. The routing daemon provides the FIB updates and information used in RheaFlow to modify forwarding behaviour on OpenFlow switches.

The BIRD used in RheaFlow was modified to send route updates to the rest of the RheaFlow application as JSON objects over a TCP socket. This is a modification that could be easily made to other open source routing software. RheaFlow could also be easily extended to receive other types of network control logic such as firewall and ACL rules in JSON without the need for an intermediate protocol.

4.2.6 Virtual Switch

The virtual switch is used to replicate the interfaces of the OpenFlow switches connected to RheaFlow. The virtual switch is an Open vSwitch instance running on the same machine as the rest of RheaFlow. The interfaces on the virtual switch are virtual ethernet pairs with one end of each pair connected to the switch while the other end is not connected. The free end of the virtual ethernet pair makes packets received by the virtual switch visible to the rest of the Linux network stack on the machine running RheaFlow. This ensures network control traffic received by the virtual switch is visible to the rest of the Linux network stack which will send it to the IP routing engine and other network applications via netlink. The virtual ethernet pairs and virtual switch are set up by RheaFlow without any intervention from the user.

4.3 RheaFlow Implementation

RheaFlow's architecture is implemented on a single Linux machine that has the components described in Section 4.2 installed. Implementing RheaFlow in a single Linux machine reduces the complexity that may be introduced by adding SDN-enabled devices to a legacy IP network. It also gives RheaFlow a simple architecture which is one of the design goals outlined earlier in this chapter. This implementation makes RheaFlow a physical as well as logical router in the network. It can be connected to the other legacy devices in a network and forward traffic to them. Also, implementing RheaFlow on a single machine ensures the control plane is logically centralised. This was an issue identified with RouteFlow by [16] because of the use of distributed routing protocols running on multiple VMs.

The functionalities in RheaFlow are provided by three main Ryu applications and auxiliary Python classes. The main applications are RheaController, RheaRouteReceiver and RheaNLSocket. RheaController is RheaFlow's core application. It responds to events from RheaRouteReceiver and RheaNLSocket by installing OpenFlow rules on the OpenFlow switches using RheaFlowProcessor. RheaFlowProcessor is a Python auxiliary class that generates OpenFlow messages and sends them to the OpenFlow switches. RheaRouteReceiver receives route information from the IP routing engine and sends an event to notify RheaController. RheaNLSocket on the other hand keeps RheaController updated about interfaces and neighbour tables on the Linux machine running

RheaFlow. RheaNLSocket uses NetlinkProcessor, an auxiliary Python netlink application to listen for netlink messages and push relevant messages to itself. The netlink messages pushed to RheaNLSocket could be sent as events to RheaController or used to update interface and neighbour tables maintained for RheaController. RheaFlow applications are modular, each application is dedicated to providing a set of functionalities and interacting with specific components of the architecture. The modular design of RheaFlow applications makes it easy to extend their functionalities or customise them in the future without compromising its stability. RheaFlow applications communicate with each other using the event notification and handling system provided by Ryu API. The event classes in the API were extended to enable RheaFlow applications to notify and handle event messages not supported in the Ryu API. These events are described later in this section.

RheaFlow is designed to support a single IP routing engine, which is on the same machine as RheaFlow to multiple OpenFlow switches. However, it is also able to receive route information from additional remote routers in the network. In RheaFlow, the ports on the connected OpenFlow switches represent a separate IP subnet within the network. These ports are made visible to the BIRD routing engine on the machine running RheaFlow via virtual ethernet interface pairs. One end of the virtual ethernet interface pair is connected to the virtual switch; the other end of the pair is configured with an IP address in the IP subnet connected to the OpenFlow switch port. This enables the BIRD IP routing engine to send and receive IP traffic from the OpenFlow switches via RheaFlow. IP packets destined for the BIRD IP routing engine from an OpenFlow switch port are sent to RheaFlow, and RheaFlow sends it to the virtual switch which forwards the packet out the virtual ethernet interface pair. This makes the packets visible to the Linux network stack which hands them over to the BIRD IP routing engine. This approach makes it possible to connect and route multiple IP subnets on a single OpenFlow switch with the machine running the RheaFlow solution acting as the gateway for each subnet. Figure 4.3 shows the interaction between RheaFlow applications and other components used in the architecture. RheaFlow applications and their functionalities are further discussed below.

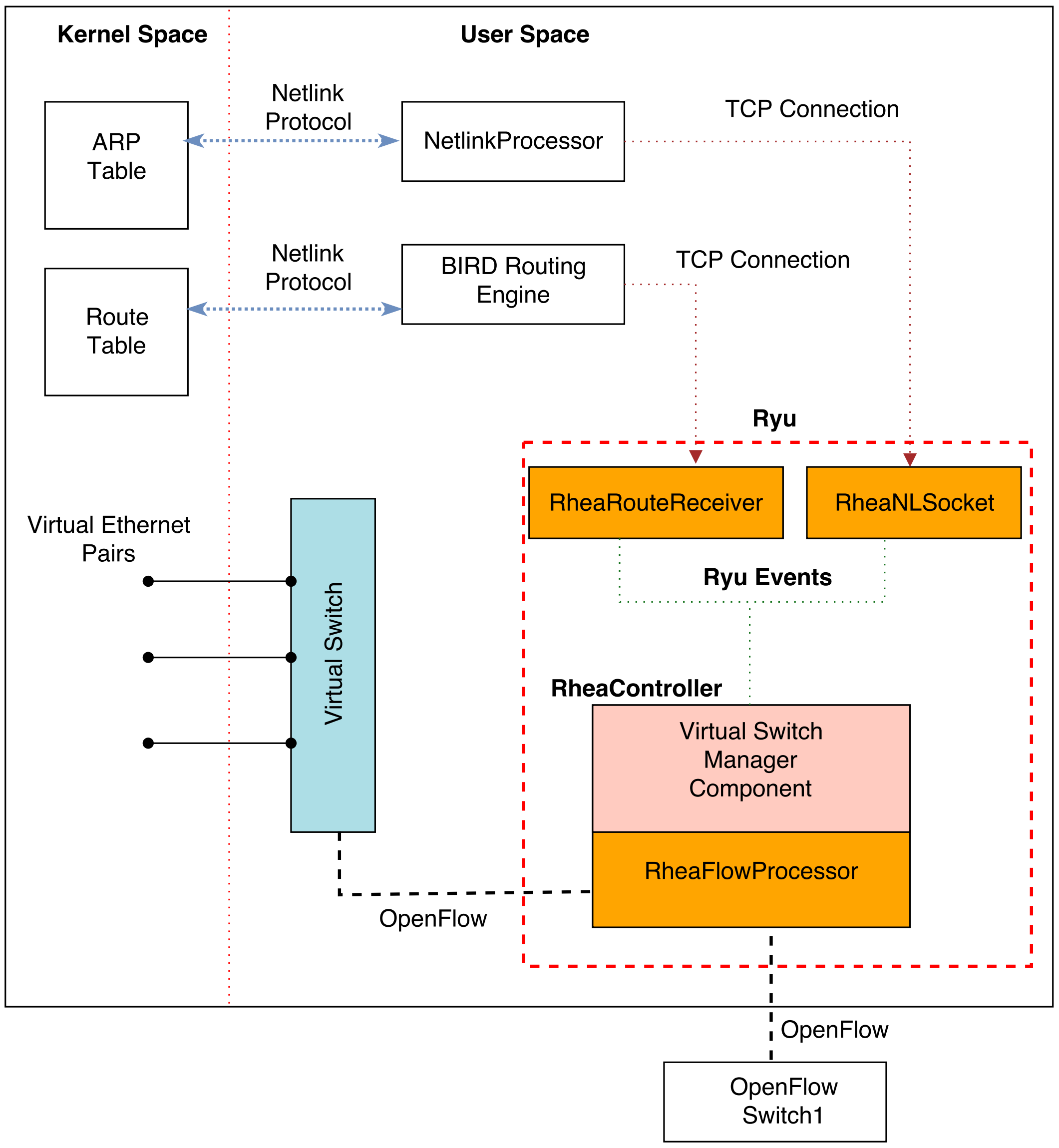

Figure 4.3: RheaFlow Architecture

4.3.1 RheaController

RheaController is the central application within RheaFlow's architecture. The overall logic of RheaFlow is implemented in the RheaController application. RheaController is reactive to changes in the state of network, and it is notified about network changes by the events received from OpenFlow controller in Ryu and other RheaFlow applications. The Ryu API provides Python decorators which are used to turn methods into event handlers for different types of events that can be generated by the OpenFlow controller and OpenFlow switches. The events sent by other RheaFlow applications to RheaController were created from base event classes and decorators provided by the Ryu API.

The RheaController application is a subclass of the *ryu.base.app_manager .RyuApp* class provided by the Ryu API. Network control is achieved by means of the methods defined in the RheaController class. These methods are event handlers for changes in the state of the network and connected OpenFlow switches that require a control decision from RheaFlow. Events are handled asynchronously by RheaController, and the Ryu API enforces an asynchronous programming style so that the application does not block while responding to an event. The events monitored and handled by the methods in the RheaController application are:

- event.EventSwitchEnter, an internal Ryu OpenFlow related event that notifies RheaController that an OpenFlow switch has connected to the controller. This event provides information about the connected OpenFlow switch such as the unique identifier for the OpenFlow switch called datapath identifier *dpid*, the numbers of ports on the device which the decorated method uses to perform the initial configuration of the OpenFlow switch and install the initial set of OpenFlow rules required by the OpenFlow switch to forward traffic from the ports specified in the configuration file.

- ofp_event.EventOFPPacketIn is an OpenFlow event that notifies RheaController that the controller has received a packet-in message from an OpenFlow switch. The event contains the *dpid* of the OpenFlow switch, the port from which it was sent and the packet encapuslated in the event. This event is sent to the controller when an OpenFlow switch does not know how to forward a packet to its destination. The RheaController method handling packet-in events examines the packet header and responds to the OpenFlow switch with an action to be performed on the packet.

- event.EventPortDelete notifies RheaController that a port on a connected OpenFlow switch has been deleted. The RheaController event handler for a port deleted event removes all OpenFlow entries related to that port from the OpenFlow switch. It also modifies the configuration of the virtual switch accordingly and notifies the required RheaFlow applications.

- event.EventPortAdd is sent by the OpenFlow Controller. It notifies RheaController that a port on a connected OpenFlow switch has been detected or added and is ready to forward packets. This triggers the RheaController event handler handling newly detected ports to install the necessary OpenFlow rules required to forward traffic through that port and modify the virtual switch configuration accordingly.

- event.EventSwitchLeave is an internal Ryu OpenFlow event that notifies RheaController that an OpenFlow switch has been disconnected. The RheaController method handling OpenFlow switch disconnection initiates a shutdown process if the disconnected OpenFlow switch is the virtual switch. Otherwise it modifies the virtual switch configuration and notifies other RheaFlow applications.
- RheaFlowEvents.EventRouterConnect is sent by RheaRouteReceiver application to notify RheaController that an IP routing engine has connected and is ready to send route update to RheaFlow.
- RheaFlowEvents.EventRouteDisconnect is used by RheaRouteReceiver to notify RheaController that an IP routing engine has disconnected and will no longer send route updates.
- RheaFlowEvents.EventRouteReceived is a RheaFlow specific event that notifies RheaController that the route information for a remote subnet has been received. The route information received for a remote subnet consists of the network address, network mask and next-hop for the network. The RheaController method processing route information enlists auxilliary methods in RheaNLSocket to resolve the next-hop address. It also installs OpenFlow rules on the necessary OpenFlow switch. This is further discussed in the RheaFlow operations.
- RheaFlowEvents.EventRouteDeleted is sent by RheaRouteReceiver when the BIRD IP routing engine sends a message withdrawing the route information for a remote subnet. It informs RheaController that the route for a remote subnet has been deleted and the network is no longer reachable via the next-hop specified in the route information. The method handling this event calls on RheaFlowProcessor methods to generate and send OpenFlow messages to remove OpenFlow rules relevant to the remote subnet on the necessary OpenFlow switch.
- RheaFlowEvents.EventNeighbourNotify is a RheaFlow specific event that notifies RheaController of changes to the neighbour or ARP table main-

tained by the forwarding table which resides in the kernel of the Linux machine running RheaFlow. This triggers the neighbour handler method in RheaController to send the necessary OpenFlow modifications to the connected OpenFlow switch if required.

RheaController provides other functionalities in RheaFlow apart from modifying forwarding behaviour in OpenFlow switches in reaction to changes in the network state or topology. It handles the creation and configuration of the virtual switch on the machine running RheaFlow. When RheaFlow is started, RheaController queries the Open vSwitch management database maintained by Open vSwitch database server to determine if there is an existing instance of the virtual switch. If the virtual switch instance exists, RheaController deletes the interfaces attached to it and deletes existing OpenFlow rules; if the virtual switch does not exist, RheaController creates the virtual switch, creates the virtual interface pair based on the configuration file and installs OpenFlow rules on the virtual switch. The virtual switch configuration and management is handled by an auxiliary Python class called VSManager which is discussed later in this section. RheaController also creates and maintains a table that maps the association between the ports used for forwarding traffic by RheaFlow on the connected OpenFlow switch and virtual interface ports connected to the virtual switch. These mappings ensure packets are delivered to the BIRD IP routing engine from ports on the OpenFlow switch via the right virtual interface pair and vice versa; this prevents the router from the dropping the packets due to a mismatch. RheaController application also depends on auxiliary python classes and applications to provide other functionalities. These Python classes are described below:

RheaFlowProcessor

RheaFlowProcessor is a Python class used by RheaController to interact with OpenFlow switches. It uses the packet libraries provided by the Ryu API to provide methods that send OpenFlow messages to the OpenFlow switches. The functions performed by the methods in the RheaController class can be categorised into three main groups:

1. Initial OpenFlow switch configuration methods: the methods in these groups are used to send initial OpenFlow configuration messages to the OpenFlow switches when they connect to RheaFlow. These methods are usually invoked by RheaController when it recieves an event.EventSwitch

Enter. They send OpenFlow messages to initialise the flow table on the OpenFlow switches and install OpenFlow entries that would enable the OpenFlow switches to send packets to the BIRD IP routing engine through *fast path* or the controller (*slow path*). These methods usually require minimal information from RheaController to install these rules since little is known about the network state when these methods are called. At most, RheaController provides the *dpid* of the OpenFlow switch, MAC address and OpenFlow port number of the virtual interface on the virtual switch that is mapped to a port on the OpenFlow switch.

2. Flow Modification methods: These methods are called by RheaController in response to network change events. They send OpenFlow messages that specify how traffic should be forwarded between the ports on the OpenFlow switches in response to network events notification received by RheaController. These methods install OpenFlow rules on the OpenFlow switch in response to route addition and deletion events, next-hop address resolution and neighbour discovery events. The OpenFlow entries generated by these methods are based on the network context information available at the time of generation. RheaController provides most of the information required to install these while some is obtained from the OpenFlow switch themselves. RheaController sends all of the association tables between the virtual switch and the OpenFlow switch, all interface details on the machine running RheaFlow, the neighbour tables and the event information received by RheaController.

3. Datapath response methods: These methods are called by RheaController's ofp_event.EventOFPPacketIn handler to process OpenFlow messages received from the OpenFlow switch. These methods use the packet libraries provided by the Ryu API to analyse the packet and generate an appropriate response to the sending OpenFlow switch. The responses generated by these packets could be OpenFlow rules specifying how the OpenFlow switch should handle subsequent packets. RheaFlow may also deliver the packet to its destination in cases where the packet is sent from the OpenFlow switch to the virtual switch or vice versa.

VSManager

VSManager is a Python class used by RheaController to create and configure the virtual switch. It abstracts the virtual switch configuration away from setup tasks that should be performed by the operator when configuring RheaFlow. This simplifies the RheaFlow configuration and minimises the chances of misconfiguration while deploying RheaFlow into a network. VSManager initialises the virtual switch and contains methods to create and connect virtual ethernet pairs to the virtual switch. It also configures the IP addresses and OpenFlow port number for the interfaces connected to the virtual switch based on the configuration file. VSManager also manages the tear down of the virtual switch configuration and makes changes to the virtual switch in response to network changes. VSManager performs these tasks by issuing subprocess calls to the Linux netlink subsystem for interface configuration and Open vSwitch's Open vSwitch Database Management Protocol (OVSDB) for virtual switch configuration.

Association Table

RheaController maintains association tables to map the virtual interfaces on the virtual switch to ports on the OpenFlow switch. It achieves this using a table class which stores the tables as Python dictionary objects. The table class provides methods that are used to modify the tables or add new entries to the tables. The tables are populated during the initial configuration of the OpenFlow switches with ports to be used for RheaFlow. The virtual ethernet pair representing each port is also created and configured. The tables enable RheaFlow to respond to changes in the port configuration without having to restart the application. For example, if RheaFlow notices that the MAC address or OpenFlow port number of a port has changed, it will update the association tables accordingly using the methods provided in the table class and make changes to the network if necessary. The tables also enable RheaFlow to intelligently handle the configuration of a port that is added to an OpenFlow switch long after the initial configuration of the OpenFlow switch has been completed without disrupting the OpenFlow switch's operations. The table class maintains three association tables.

- The OpenFlow switch to virtual switch map table. The entries in this table are used to map OpenFlow switch ports to virtual switch ports. The *dpid*, OpenFlow port number, port name and port hardware address for an OpenFlow switch port is mapped to a port name, OpenFlow port number and port hardware address of the virtual switch port. The OpenFlow switch details are grouped as a tuple and used as a key to identify a tuple containing the virtual switch details in the Python dictionary.

- *Fastpaths* table is a dictionary object that is used in *fast path* operations. It enables RheaController to associate a *fast path* label with a particular OpenFlow switch port to virtual switch port association.

- Inter-switch link table is a dictionary object that is used to install OpenFlow rules that will handle traffic forwarding on links between multiple OpenFlow switches without involving the controller. The inter-switch link table enables RheaController to map an inter-switch label to the association between ports on different OpenFlow switches.

RheaFlowEvents

The RheaFlowEvents class extends the base event classes provided in the Ryu API to define new events for RheaFlow. These events are sent to RheaController by other RheaFlow applications. The events contain messages about network events that event handler methods in RheaController use to generate OpenFlow messages. RheaFlowEvents class eliminates the need for RFProtocol message format as used in RouteFlow which increases the control traffic overhead. The events defined in this class and observed by RheaController methods are:

- EventRouterConnect is used to notify RheaController that an IP routing engine has connected. It provides the IP address and TCP port number of the IP routing engine. It is currently only used for logging and notifications in RheaController.

- EventRouterDisconnect is used to notify RheaController that an IP routing engine has disconnected. It provides the IP address and TCP port number of the disconnected IP routing engine.

- EventRouteReceived notifies RheaController that a route to a remote subnet has been received. The route sent by the event is a Python tuple that contains the address of the remote network, the subnet mask and the IP of the next-hop for that network.

- EventRouteDeleted notifies RheaController that a remote network is no longer reachable via a next-hop. The unreachable route is a tuple that contains the address of the network, the mask of the network and the next-hop for that network.

- EventNeighbourNotify notifies RheaController of changes to the neighbour table maintained by the Linux network stack of the machine running RheaFlow. This event indicates that a neighbour has been added or deleted from the neighbour table. The action performed on the neighbour and the neighbour's details are sent in the event.

RheaYAML

RheaYAML is a Python class used by RheaController to parse the configuration file which is written in YAML. YAML is a simple annotation language format used to store data. It is suitable for configuration files and was chosen because it is human-readable. A configuration file that is both human-readable and easily parsed by the machine minimises misconfigurations. The RheaYAML class utilises Python YAML libraries to convert the documents in the configuration file into Python data types that are used by RheaController to perform initial configuration of the OpenFlow switches and the virtual interfaces on the virtual switch. The configuration file consists of two block collections. One collection is used to specify the information needed to configure the OpenFlow switches and the other collection specifies the configuration of the virtual switch for *fast path* operation. Each node in the OpenFlow switch collection specifies the details for each OpenFlow switch that will be connected to RheaFlow. In each OpenFlow switch node, mappings are used to provide the information required to configure each OpenFlow switch. A sample of RheaFlow's configuration file is provided in Appendix A. The mappings in the node for each OpenFlow switch are used to describe the OpenFlow switch. The entries in the nodes for each OpenFlow switch are:

- *name* which is the human-readable name assigned to the OpenFlow switch.
- *type* is used to identify the vendor of the OpenFlow switch.
- *dp_id* is the unique identifier for the OpenFlow switch.
- *vs_port_prefix* is used to set the interface name for the virtual interfaces created by RheaController for replicating the OpenFlow switch ports to the IP routing engine.
- *ports* specifies the OpenFlow ports on the OpenFlow swith that will be used by RheaFlow. It also specifies the IP addresses to be assigned to

be virtual interfaces mapped to the OpenFlow switch ports. The IP address assigned to the virtual interface mapped to an OpenFlow switch port must belong to the IP subnet connected to the OpenFlow switch port. This is required so that the Linux network stack of the RheaFlow machine can perform MAC to IP address resolution.

- *fastpath_port* is used to specify the OpenFlow switch port that will be used for *fast path*.
- *fastpath_vs* is used to specify the OpenFlow port number of the *fast path* interface on the RheaFlow machine.
- *isl_port* is used to specify the designated OpenFlow switch port for inter-switch link.
- *isl_rem_port* is used to specify the OpenFlow port to be used for an inter-switch link with another OpenFlow switch connected to RheaFlow.
- *isl_rem_dp_id* is used to specify the unique identifier for another OpenFlow switch undertaking inter-switch connections.
- *rem_port* is a list used to specify the OpenFlow ports on the remote OpenFlow switch that will forward traffic over the inter-switch link.

Three of the described mappings in the OpenFlow switch nodes are required to configure RheaFlow while the others are optional. RheaFlow will fail to start if any of these three mappings: *name*, *dp_id*, *a*nd *ports* are missing. The *fast path* related mappings enable *fast path* mode in RheaFlow if values are specified for them. The inter-switch related mappings enable inter-switch operations. The *fastpath_interface* and *fastpath_port* mappings in the virtual switch collection are used to specify the name of the interface on the RheaFlow machine that will be used for *fast path* and the OpenFlow port number that should be assigned to it. Specifying values for the virtual switch connection are required if RheaFlow is to be operated in *fast path* mode.

When the RheaFlow application is started, the configuration file is loaded by RheaController and OpenFlow switch nodes are converted into Python data types in the application. When an OpenFlow switch connects to RheaFlow, the OpenFlow switch *dpid* provided by the switch is checked against the identifier in the configuration file; if there is a match, RheaController creates and configures a virtual interface for each of the ports specified in the configuration with the IP addresses. If the *dpid* supplied by the OpenFlow switch does not match any of the those provided in the configuration, RheaController does not complete the rest of the configuration process for the OpenFlow switch.

4.3.2 RheaRouteReceiver

RheaRouteReceiver is a Ryu application that receives route information encoded in JSON from the BIRD IP routing engine. It decodes the JSON message received, verifies the decoded route information and sends the route information to RheaController as an EventRouteReceived event. RheaRouteReceiver application is a *ryu.base.app_manager.RyuApp* subclass that utilises the WSGI interfaces provided by Ryu's API to create a client-server model between RheaController and the BIRD IP routing engine running in RheaFlow.

RheaRouteReceiver uses the Python JSON processing library to decode messages received from the IP routing engine. It listens on TCP port 55650 for inbound connections from an IP routing engine. Once the IP routing engine connects to RheaRouteReceiver it sends an EventRouterConnect event which contains the IP address and TCP port number of the IP routing engine to RheaController. When the IP routing engine disconnects it sends an EventRouterDisconnect to notify RheaController. When a JSON encoded message from the connected IP routing engine is received by the application, it calls methods in the JsonHandler class to verify that the message received is in JSON format, decodes the message and verifies that information received is complete. The JsonHandler class contains methods that can be used to fix up incomplete decoded route information received by the application. However, if any of the verification steps fail and JsonHandler cannot fix the route information, it will send an error notification to the IP routing engine that the route information can not be translated into OpenFlow rules. If the message received from the IP routing engine is successfully verified and decoded, an EventRouteReceived event is generated and sent to RheaController. The event handler processing the event returns a result depending on the outcome of the operation performed. This is also delivered to the IP routing engine over the TCP connection.

The are two events that can be generated based on the message received from the IP routing engine. EventRouteReceived is generated by RheaRouteReceiver when the route information received from the IP routing engine indicates a route to a remote subnet should be added. EventRouteDeleted is generated by RheaRouteReceiver when the route information received from the IP routing engine indicates a route to a remote subnet has been withdrawn. In both cases, the route information sent by the IP routing engine contains the address of the remote subnet, the address mask and the next-hop address for reaching the network.

4.3.3 RheaNLSocket

RheaNLSocket is the third Ryu application. It maintains information about the interfaces and neighbour tables on the Linux machine running RheaFlow applications by listening to netlink messages. It also generates EventNeighbourNotify events which are sent to RheaController when a neighbour is added or deleted from the neighbour table.

RheaNLSocket also utilises the WSGI interfaces provided by Ryu's API to create a server application that receives updates about interfaces and neighbour tables from a Python application called NetlinkProcessor. The NetlinkProcessor is an auxiliary application that uses the Pyroute2 netlink library to create a netlink socket to receive and process netlink messages asychronously. It includes a callback subroutine that sends a message to RheaNLSocket when link layer, address and neighbour netlink messages are received. The netlink messages that trigger a callback from the NetlinkProcessor application include: RTM_NEWNEIGH, RTM_DELNEIGH, RTM_NEWLINK, RTM_DELLINK, RTM_NEWADDR and RTM_DELADDR.

RheaNLSocket listens on TCP port 55651 for messages generated by the callback subroutine in NetlinkProcessor. For RTM_NEWNEIGH and RTM_DEL NEIGH netlink messages received by NetlinkProcessor. The callback subroutine updates the neighbour table maintained by RheaNLSocket. This also triggers RheaNLSocket to send an EventNeighbourNotify event to RheaController. The neighbour entries in the neighbour table contain the MAC and IP address of the neighbour and the index of the interface on which the neighbour was discovered. RTM_NEWNEIGH netlink messages contain the neighbour entries and are used to update the neighbour table. The corresponding event sent to the RheaController contains the neighbour entry and a RTM_NEWNEIGH action to signify a new neighbour. RheaController adds OpenFlow entries for forwarding traffic to the neighbour on the OpenFlow switches if the neighbour was discovered on one of the virtual interfaces connected to the virtual switch. RTM_DELNEIGH netlink messages contain the same details as an RTM_NEWNEIGH except it indicates a neighbour is no longer reachable and the neighbour entry is removed for that neighbour. The corresponding event sent to RheaController contains the neighbour entry and a RTM_DELNEIGH action which signifies the neighbour is unreachable. This causes RheaController to send OpenFlow modification messages to remove flow rules that have been added for the neighbour. The rest of the netlink messages that trigger a callback from the NetlinkProcessor generates an interface table containing all details about the interfaces on the Linux machine running RheaFlow. The interface table is sent to RheaNLSocket, which replaces its existing interface table with the newly received table. These messages do not cause an event to be generated; however, they are part of the network state information used by RheaController to generate OpenFlow rules.

RheaNLSocket also includes methods that can be called from RheaController to initiate a neighbour discovery process for a host. This is usually used when there is no neighbour entry for the next-hop to a remote network.

4.4 Intercepting Network Control Traffic

RheaFlow controls the forwarding behaviour of OpenFlow switches connected to it by installing OpenFlow rules on these OpenFlow switches in response to changes in the network. Some of these changes occur on the connected OpenFlow switches, some are triggered by traffic forwarded by the OpenFlow switches which requires that the OpenFlow switches exchange packets with the IP routing engine and other applications in the network stack of the machine running RheaFlow. As such OpenFlow switches connected to RheaFlow should forward routing protocol traffic between a remote router and the IP routing engine in the RheaFlow environment. This includes control traffic such as Internet Control Message Protocol (ICMP), ARP and IPv6's Neighbour Discovery Protocol (NDP) because they are required by the network stack of the RheaFlow machine to perform diagnostic and neighbour discovery operations. Apart from that control traffic, the OpenFlow switches should also forward IP routing protocol traffic like OSPF and BGP between the machine running RheaFlow and remote routers.

RheaFlow can be operated in two modes. These modes specify the OpenFlow rules that should be installed on the OpenFlow switches and the virtual switch to enable the forwarding of packets between the RheaFlow machine and remote hosts connected to the OpenFlow switches. These modes are described in the following two subsections.

- *Slow path*: the default operating mode for RheaFlow. The application is operated in this mode if no valid values are set for the *fastpath_port* and *fastpath_vs* options for an OpenFlow switch or the *fastpath_interface* and *fastpath_port* options for the virtual switch in the RheaFlow configuration file. When RheaFlow is operated in *slow path* mode, the virtual switch is configured with OpenFlow rules that instruct it to send all incoming packets received from the Linux network stack or IP routing engine on the virtual interfaces to RheaController via the OpenFlow control channel. The OpenFlow switches are configured with OpenFlow rules that instruct them to send all incoming packets received on their interfaces to be delivered to RheaController via the OpenFlow control channel. RheaController receives inbound packets sent by the OpenFlow switches via the OpenFlow control channel as ofp_event.EventOFPPacketIn events and then checks the destination addresses of the packets. If the destination addresses for the packets match the addresses of the virtual interfaces on the RheaFlow machine, RheaController sends the packets via the OpenFlow control channel to the virtual switch which delivers them out the right interface for the applications listening for the packets. Likewise, incoming packets from the network stack of the RheaFlow machine received by the virtual interfaces are also sent to RheaController via the OpenFlow control channel which are received as ofp_event.EventOFPPacketIn events. These events are checked to determine which OpenFlow switch ports the packets should be delivered to.

  The use of the OpenFlow control channel to deliver traffic between the OpenFlow switches and the RheaFlow machine adds considerable overhead to the RheaFlow operation. The forwarding of traffic between the RheaFlow machine and remote devices over the OpenFlow control channel is slower because of the additional encapsulation that has to be done by the OpenFlow switches and RheaFlow to deliver the packets. RheaFlow may also be susceptible to denial-of-service attacks while operating in *slow path* mode. This may happen if the connected OpenFlow switches flood the OpenFlow control channel with packets thereby causing RheaFlow to dedicate available resources to responding and processing packet-ins while neglecting other network events. RheaFlow should be operated in *slow path* mode only if low amounts of traffic is expected between the OpenFlow switches and RheaFlow machine.

- *Fast path*: the recommended mode for operating the RheaFlow application. The application is operated in *fast path* mode if valid values are set for the *fastpath_port* and *fastpath_vs* options for an OpenFlow switch and the *fastpath_interface* and *fastpath_port* options for the virtual switch in the RheaFlow configuration file. *Fast path* configuration mode requires that a designated port on the OpenFlow switch specified by the *fastpath_port* option in the configuration file be connected to an interface on the RheaFlow machine specified by the *fastpath_interface*, *fastpath_port* and *fastpath_vs* options in the configuration file to create a *fast path* link. The *fast path* link is configured as a VLAN trunk port between the virtual switch and the OpenFlow switches connected to RheaFlow. VLAN tags are then created for each virtual interface mapped to an OpenFlow switch port. Packets are exchanged between the virtual interfaces of RheaFlow and the OpenFlow switch ports by tagging the packets with the assigned VLAN tag for the virtual interface to OpenFlow switch port mapping. They are forwarded over the *fast path* link bypassing the RheaFlow application.

  During initial configuration the *fast path* interface on the RheaFlow machine is added to the virtual switch. A *fast path* label is assigned to the mapping between a configured virtual interface on the virtual switch and OpenFlow switch port. These details are stored in the *Fastpaths* table maintained by RheaController. The *fast path* labels are the VLAN tags added to the packets transiting through the *fast path* link. The OpenFlow rules installed on the virtual switch instruct it to tag any outbound packet on a virtual interface with the VLAN tag assigned to the virtual interface and forward it via the *fast path* link to the OpenFlow switch of the destination port. The OpenFlow rules on the virtual switch also instruct it to strip inbound packets received over the *fast path* link of their VLAN tags and forward the packets out of the virtual interface assigned to the VLAN tags. The OpenFlow rules installed on the OpenFlow switches for exchanging packets with the virtual switch instruct them to tag packets from a port bound for the mapped virtual interface with the assigned VLAN tag for that ingress port and forward it over the *fast path* link while packets received from the *fast path* link are stripped of their VLAN tags and forwarded out the port the stripped VLAN tags were assigned to.

RheaFlow operation in *fast path* mode is efficient as the virtual switch and the connected OpenFlow switches do not have to send packets to each other via the slow OpenFlow control channel. This enables RheaFlow to focus on installing OpenFlow rules in response to network events and reduce the number of OpenFlow messages exchanged between the OpenFlow switches and RheaFlow. It also reduces load on the OpenFlow switch processor as it does not have to encapsulate control packets as OpenFlow packet-in messages. However, *fast path* requires a link between all connected OpenFlow switches and the machine running RheaFlow. This could be problematic in a RheaFlow configuration with multiple OpenFlow switches connected as the machine may not have the port density required to establish a *fast path* link with the connected OpenFlow switches. Inter-switch links have been proposed to address this issue. This requires setting up links between the participating OpenFlow switches using the *isl_port*, *isl_rem_port*, *isl_rem_dp_id* and *rem_port* options specified in the configuration file. One or more of the participating OpenFlow switches depending on the interfaces available on the RheaFlow machine would have *fast path* links to the RheaFlow machine and inter-switch links will be used to connect the OpenFlow switches together. VLAN tags are assigned for the inter-switch link configuration. OpenFlow rules are installed on all OpenFlow switches to ensure OpenFlow switches that do not have *fast path* links are able to forward traffic to the RheaFlow machine via the inter-switch links with OpenFlow switches that have *fast path* links. The inter-switch link configuration has not been implemented for this thesis.

4.5 OpenFlow Rules

OpenFlow switches connected to RheaFlow forward packets by matching flow entries in their flow tables. Flow entries are inserted or removed from the OpenFlow switches' flow tables when the OpenFlow switches receive *FlowMod* messages from the controller. *FlowMod* messages consist of a command which is used to indicate if the flow entry should be added, deleted or modified, the number of the flow table where the entry should be added, a cookie which is used to identify an entry in the flow table, a priority number used to enforce the order in which packets are matched against entries in the table, the idle_timeout and hard_timeout which represents the number of seconds since a packet has hit the flow entry and the number of seconds before the nety expires and is deleted by the OpenFlow switch, the match which specifies the headers of the packet and the ingress port, the action which specifies what operations should be performed on matched packets and out_port which is used to specify the egress port for the match packet after the actions have been performed.

FlowMod messages are sent by the RheaFlow application to configure the OpenFlow switches. They add or delete flow entries in response to network events such as route addition or deletion and neighbour discovery on the OpenFlow switch ports. Some OpenFlow switches have multiple flow tables with multiple flow entries in each table. However, flow entries added by RheaFlow are limited to a single table. This design choice enables RheaFlow to work with a range of OpenFlow switches with the lowest common denominator being a single flow table OpenFlow switch.

The flow entries added to an OpenFlow switch's flow table by RheaFlow can be divided into two sets. The first set of flow entries are added by RheaFlow during the initial configuration of the OpenFlow switch. The matches defined for these entries are used to catch packets that should be forwarded to the interfaces on the machine running the RheaFlow application. The actions performed on packets that match these entries depend on the configuration mode of RheaFlow. Packets that match flow entries added during the initial OpenFlow switch configuration in *slow path* mode are sent to the controller while packets that match flow entries added during the initial OpenFlow switch configuration in *fast path* mode have a VLAN tag added based on their ingress port and are forwarded out the *fast path* port on the OpenFlow switch. The second set of flow entries are added by RheaFlow when it is notified of new routes or neighbours. The matches defined in these entries catch packets that should be delivered to the network reachable via the route just received by RheaFlow or the neighbour that was just discovered. The actions performed on the packets enable the OpenFlow switches to forward the packets to their destination without requiring the BIRD IP routing engine on the RheaFlow machine to perform expensive and time-consuming route lookup operations.

The flow entries added to the OpenFlow switch's flow tables for routes are the same in most cases in the *fast path* and *slow path* mode except when the next-hop for a route is an interface on the RheaFlow machine.

4.5.1 OpenFlow Entry Creation Process

This section outlines the process of inserting flow entries in the OpenFlow switch's flow table by the RheaFlow application. It also describes the matches and actions performed.

Slow Path Configuration Flow Entries

Flow entries added to the OpenFlow switch's flow table during configuration match packets that should be forwarded to the interfaces on the RheaFlow machine. As mentioned in section 4.3, each port on an OpenFlow switch is connected to a single IP subnet and carries traffic for that subnet. The virtual interface created on the RheaFlow machine for each OpenFlow switch port is assigned an IP address within the subnet connected to the OpenFlow switch port. Because the BIRD IP routing engine is on the RheaFlow machine, the IP address assigned to the virtual interface is a gateway address for the subnet connected to the OpenFlow switch port. Since the virtual interface IP address is part of a subnet connected to the OpenFlow switch port, the virtual interface should be able to receive and send ARP requests and responses if the subnet is an IPv4 network, and receive and send NDP packets for an IPv6 subnet. It should also be able to receive and send ICMP packets and normal data packets. Table 4.1 shows an example of the default OpenFlow rules installed for *slow path.*

For the OpenFlow switch to match and send most of the packets described above to the RheaController which then sends it to the virtual switch for forwarding out the right virtual interface, it needs to know the MAC address of the virtual interface. The MAC address of the virtual interface is supplied by RheaNLSocket to RheaController during the configuration process. The RheaFlowProcessor methods then generate flow entries to catch packets whose destination MAC header field match the MAC address of the virtual interface for both IPv4 and IPv6 packet types. Entries are generated for the IPv6 link-local address of the virtual interface which is created from the MAC address. Entries are generated for IPv6 link-local multicast addresses and ARP broadcasts so that these may be sent to the virtual interfaces on the RheaFlow machine. The action performed on packets that match the *slow path* flow entries is to send them to RheaController as packet-in messages. All *slow path* flow entries are assigned a lower priority than flow entries added for routes. Flow entries for ARP packets have a slightly higher priority than other *slow path* entries. However, flow entries for ARP still have a lower priority than flow entries for routes.

Matches	Actions
ARP, Ethernet dst=ff:ff:ff:ff:ff:ff	Send to controller
in_port=1, IPv4 type, Ethernet dst=0c:84:dc:54:eb:d6	Send to controller
in_port=1, ICMP type, Ethernet dst=0c:84:dc:54:eb:d6	Send to controller
in_port=1, IPv6 type, Ethernet dst=0c:84:dc:54:eb:d6	Send to controller
in_port=1, IPv6 dst=2001:0db8:0:f101::/64, Ethernet dst=0c:84:dc:54:eb:d6	Send to controller
in_port=1, ICMPv6	Send to controller
in_port=1, ICMPv6, Ethernet dst=0c:84:dc:54:eb:d6	Send to controller

Table 4.1: Example of Default *Slow Path* Entries

Fast Path Configuration Flow Entries

Flow entries installed on OpenFlow switches for *fast path* mode perform the same functions as those installed on the OpenFlow switches for *slow path.* They match packets that should be forwarded to the interfaces on the RheaFlow machine. However, this is done by tagging packets with VLANs that correspond to the virtual interface to OpenFlow switch port mappings available for the OpenFlow switch and then forwarding these tagged packets via the *fast path* link to the RheaFlow machine. Additionally, some of the *fast path* flow entries match the VLAN tagged to packets received via the *fast path* link from the RheaFlow machine, strip the packets of the VLAN tags and forward the packets out the right OpenFlow switch ports based on the VLAN. This ensures the virtual interfaces on the RheaFlow machine are able to send and receive traffic from subnets connected to the OpenFlow switch ports without using the slow OpenFlow control channel between the virtual switch and OpenFlow switches. Table 4.2 shows an example of the default OpenFlow rules installed for *fast path.*

Most of the *fast path* flow entries for an OpenFlow switch port match packets whose destination MAC address and IPv6 link-local address are the same as the MAC and IPv6 link local address of the virtual interface mapped to the port. All the *fast path* flow entries require the VLAN tag and the OpenFlow port number of the designated *fast path* port on the OpenFlow switch. The MAC address of the virtual interface is provided RheaNLSocket and VLAN tag is created during the configuration process by RheaController and stored in the *Fastpaths* table. The flow entries match for ARP broadcasts, IPv4, IPv6, ICMP packets with destination MAC addresses that are the same as the MAC addresses of the virtual interfaces on the RheaFlow machine. These flow entries also match for IPv6 packets with the link-local address of the virtual interface as the destination IPv6 address and IPv6 multicasts packets received on the OpenFlow switch port. Packets that matches these flow entries then have VLANs representing OpenFlow switch port to virtual interface map pushed on their headers and forwarded via the *fast path* link as actions. The flow entries for the *fast path* port match against packets with the VLANs assigned to that port mapping. These VLANs are popped from the packets and forwarded the out the OpenFlow switch port. All *fast path* flow entries have a higher priority than *slow path* flow entries. However, these flow entries still have a lower priority than entries installed for routes. This is mostly to help an operator differentiate between *slow path* and *fast path* flow entries.

Matches	Actions
ARP, in_port=1	Push VLAN label, output *fast path* port
in_port=1, Ethernet dst=0c:84:dc:54:eb:d6	Push VLAN label, output *fast path* port
in_port=1, IPv4 type, Ethernet dst=0c:84:dc:54:eb:d6	Push VLAN label, output *fast path* port
in_port=1, ICMP type, Ethernet dst=0c:84:dc:54:eb:d6	Push VLAN label, output *fast path* port
in_port=1, IPv6 type, Ethernet dst=0c:84:dc:54:eb:d6	Push VLAN label, output *fast path* port
in_port=1, IPv6 dst=2001:0db8:0:f101::/64, Ethernet dst=0c:84:dc:54:eb:d6	Push VLAN label, output *fast path* port
in_port=1, ICMPv6	Push VLAN label, output *fast path* port
in_port=1, ICMPv6, Ethernet dst=0c:84:dc:54:eb:d6	Push VLAN label, output *fast path* port
in *fast path* port, VLAN label	Pop VLAN label, output port

Table 4.2: Example of Default *Fast Path* Entries

Neighbour Discovery and Route Flow Entries

The BIRD IP routing engine on the RheaFlow machine maintains the routing table for the network. The BIRD IP routing engine populates this table by exchanging routing protocol information with other routers in the network. Conventional network switches forward packets out their ports based on data link layer addresses hence; they can only forward packets between hosts on the same subnet. Packets to be forwarded between hosts on different subnets require an IP router, which performs a lookup in its routing table to find a next-hop to the subnet in which the destination host resides and forward the packets via the next-hop address. The packet forwarding between different subnets performed by a router is similar to the forwarding performed by switches. However, hardware IP routers are usually limited by the number of interfaces they have. This is because TCAMs which are used to perform fast lookups on the forwarding tables to determine the interface via which packets should be forwarded, are expensive.

The flow entries added to an OpenFlow switch's flow table by RheaFlow are based on routes received from the BIRD IP routing engine or neighbours discovered on the virtual interfaces. These flow entries enable the OpenFlow switch to forward packets between different subnets connected to the OpenFlow switch ports without requiring the IP routing engine. This speeds up the delivery of packets to their destination as packets between different subnets that would have been sent to the virtual interface on the RheaFlow machine for the IP routing engine to perform a lookup and forward the packets out the right interface just have certain headers modified and forwarded between different ports on the OpenFlow switch.

A neighbour discovered on the virtual interface of the RheaFlow machine has to be a host directly connected or on the subnet connected to the OpenFlow switch port mapped to that virtual interface. This requires that flow entries are added for other ports on the OpenFlow switch so that packets sent by other hosts from different subnets connected to other ports on the OpenFlow switch to the discovered host are forwarded to the OpenFlow switch port mapped to the virtual interface on which the host was discovered. When a neighbour is discovered on a virtual interface, RheaNLSocket notifies RheaController and provides the IP and MAC address of the neighbour along with details of the interface on which it was discovered. The RheaController neighbour-notify event handler generates flow entries to match for packets that are to be sent to the neighbour from other OpenFlow switch ports. The fields matched in the flow entries are:

- Port number of the ingress port on which the packet is received.
- IP version. IPv4 or IPv6.
- IP destination address of the neighbour.
- Destination MAC address which is the address of the virtual interface mapped to the ingress port as this virtual interface is the gateway of the subnet connected to the ingress port.

The actions performed on packets that match the flow entries are:

- Source MAC address of the packet is replaced by the MAC address of the virtual interface on which the neighbour was discovered because it is the next-hop for that neighbour and MAC addresses can not be used for forwarding packets between different subnets. The original source MAC address was from the sender.

- Destination MAC address of the packet is set to the MAC address of the neighbour.
- And, finally, output the packet via the port mapped to the virtual interface on which the neighbour was discovered.

This process is the same for both IPv4 and IPv6 neighbours. These flow entries have a higher priority than flow entries for sending packets to RheaFlow via *slow path* or *fast path*.

Flow entries for routes received from the IP routing engine are generated by RheaController in a pattern similar to neighbour flow entries. When a RouteReceived event is received by RheaController from RheaRouteReceiver, RheaController checks the neigbour table maintained by RheaNLSocket to find the next-hop. If the next-hop for the received route is an address reachable via a port on an OpenFlow switch, the next-hop would be a neighbour discovered on the virtual interface mapped to the port; hence the MAC address for the next-hop would be available in the neighbour table maintained by RheaNLSocket. The fields matched by the flow entries to forward packets that addressed to the network of the received route are:

- Port number of the ingress port on the OpenFlow switch on which the packet is received.
- IP version.
- Destination MAC address which is the address of the virtual interface mapped to the ingress port as this virtual interface is the gateway of the subnet connected to the ingress port.
- the network address which could be either IPv4 or IPv6.

The action performed on packets that matches these entries are:

- Source MAC address of the packet is replaced by the MAC address of the virtual interface on which the next-hop host was discovered.
- Destination MAC address of the packet is set to the MAC address of the

next-hop.

- And, finally, output the packet via the port mapped to the virtual interface on which the next-hop was discovered.

It is possible for the next-hop for the received route to be the address of a physical interface on RheaFlow machine or a neighbour connected to a physical interface on the RheaFlow machine. In that case, the fields matched by these flow entries are the same as flow entries for received routes with next-hops reachable via a port on an OpenFlow switch. Since the packets are to be forwarded to their destination via an interface on the RheaFlow machine. The packets are forwarded to the virtual interfaces on the RheaFlow machine using the same actions in the either *fast path* or *slow path* configuration flow entries. The Linux network stack then forwards the packets out the right physical interface.

If the MAC address for a next-hop to a route is not found, RheaFlow initiates a neighbour discovery process which involves sending User Datagram Protocol (UDP) packets on port 6666 to the next-hop. This would force the Linux network stack to perform address resolution. If the MAC for the next-hop is not resolved, the route is kept in a pending route table maintained by RheaController. RheaController checks the pending route table every ten minutes to see if the next-hop MAC address for the pending routes are available. However, this is unlikely to happen because the Linux network stack which maintains the forwarding table for routes maintained by the BIRD IP routing engine in RheaFlow would notify the IP routing engine if a next-hop is unreachable. This would force the IP routing engine to withdraw the route or find another to the network.

Removing Flow Entries

When routes are deleted or neighbours are removed, the corresponding flow entries generated for these events are deleted from the OpenFlow switches by RheaFlow. The flow entries to be removed are generated by the RheaController just as they would be if they were being installed, and the *FlowMod* messages sent to the OpenFlow switch indicates that the specified entries be deleted from the OpenFlow switch. If the flow entries exist on the OpenFlow switch, they will be removed; however, if they don't exist, the OpenFlow switch ignores the *FlowMod* message. The network information needed to delete flow entries for routes and neighbours is the same as the ones need to add them therefore, the flow entries deleted on OpenFlow switches are for the specific neighbours and routes.

Chapter 5

Evaluation

In this chapter, the testing methodology used to validate RheaFlow is described and its performance as a hybrid router is discussed. RheaFlow is also compared against its predecessor RouteFlow to examine its progression towards the design goals identified in Chapter 4.

5.1 RheaFlow Testbed

A network was setup to test the RheaFlow prototype. Three IPv4 subnets were connected to three ports on an OpenFlow switch which was managed by RheaFlow. Additionally, there were routers in these subnets that exchanged routing protocol information with the BIRD routing software that provides route information for RheaFlow. Due to time constraints, it was not possible to test RheaFlow with hardware OpenFlow switches. RheaFlow was tested with an Open vSwitch instance accelerated with Intel's Data Plane Development Kit (DPDK). This enabled the Open vSwitch to provide similar packet forwarding performance to that obtained on hardware OpenFlow switches. The Open vSwitch instance is called *br-dpdk*. It is a Dell PowerEdge R530 server with two Intel Xeon E5-2630 v3 family processors and four quad port Intel I350 network interface cards running a Debian 8 operating system with Linux kernel version 3.16.

The RheaFlow prototype was deployed on a Linux machine in the test network that has the modified BIRD IP routing engine daemon, Ryu and Open vSwitch installed. The machine running RheaFlow is a Dell OptiPlex 755 PC with Intel Core 2 Duo E6750 processor, four gigabytes of memory and three ethernet ports. It uses an Ubuntu 14.04 operating system with Linux kernel version 3.16. The RheaFlow application was executed with Ryu version 4.0. Each of the three *br-dpdk* ports used in this setup were multiplexed by a conventional switch device downstream so that multiple hosts could be connected to each port. The configuration file shown in Appendix A was used to configure RheaFlow in *fast path* mode to manage *br-dpdk*. Table 5.1 shows subnets connected to the ports, and the addresses and names of the virtual interface created for each port based on the machine running RheaFlow. Figure 5.1 show a diagram of the test network.

br-dpdk port number	IP subnet connected	Virtual interface name	Virtual interface address
5	20.0.0.0/24	br-dpdk-p5	20.0.0.254/24
6	30.0.0.0/24	br-dpdk-p6	30.0.0.254/24
7	40.0.0.0/24	br-dpdk-p7	40.0.0.254/24

Table 5.1: IP subnets Connected to *br-dpdk* in the Test Network

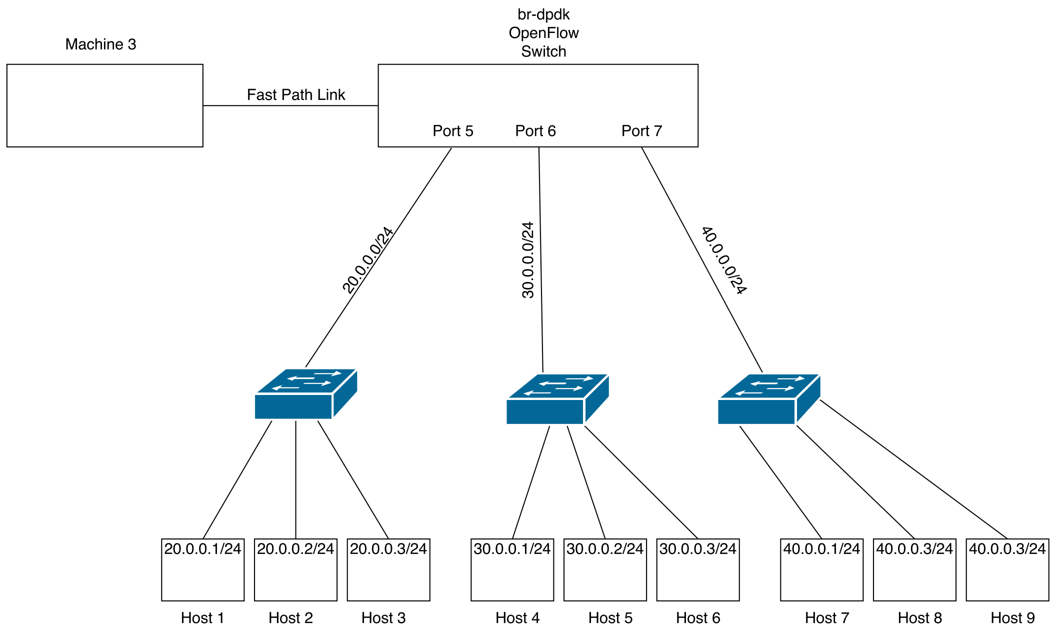

Figure 5.1: Diagram of the RheaFlow Test Network

The hosts in the subnets connected to *br-dpdk* are configured as BGP routers that exchange route information via *br-dpdk* with the BIRD IP routing engine in RheaFlow. These routers are next-hops to remote subnets. Appendix B shows the configuration file used by the BIRD IP routing engine on the RheaFlow machine to setup BGP peering sessions with routers in the connected subnets. The BIRD configuration file also specifies static routes to remote subnets reachable via the routers in the connected subnets. The static routes in the BIRD configuration file are the initial set of routes that were sent to RheaFlow and converted to OpenFlow entries when the BIRD IP routing engine is started. Subsequent routes sent to RheaFlow by the BIRD router are received from the BGP peers. 210,215 routes from the internet's BGP routing table were advertised from routers in the connected subnets to the BIRD routing daemon on the RheaFlow machine. This was a stress test operation to assess RheaFlow's performance under load. The BGP routing table used for this test was the BGP table of the RouteView's sydney router[3] on the first of march 2016. This resulted in over 400,000 flow entries being installed on *br-dpdk* for the routes. The routes were advertised from a router connected to *br-dpdk* port 5. This meant that each route only required two flow entries for *br-dpdk* ports 5 and 6. The average execution time for RheaFlow's RouteReceived event handler to generate and install flow entries for a single route was $8.74ms$. The execution time of the RouteReceived event handler was used to evaluate the speed of the RheaFlow code because more time was spent by the RouteReceived event handler code processing the received routes than other RheaFlow components. The time spent by the RheaRouteReceiver application code on the routes before it raised a RouteReceived event was insignificant compared to that spent by the RouteReceived event handler.

Figure 5.2 shows a flame graph visualisation generated from the CPU profile of RheaFlow application after it had received and deleted 210,215 routes. Each box in the graph represents a function in the RheaFlow code which is called a stack frame. The y-axis of the graph is used to describe the relationship between the functions visualised in the graph; the function beneath a function is its parent. The x-axis of the graph spans all the functions in the application and the width of a box represents the total time the function indicated by the box was executed by the CPU. The graph shows the RheaFlow route event handler functions took the most CPU time.

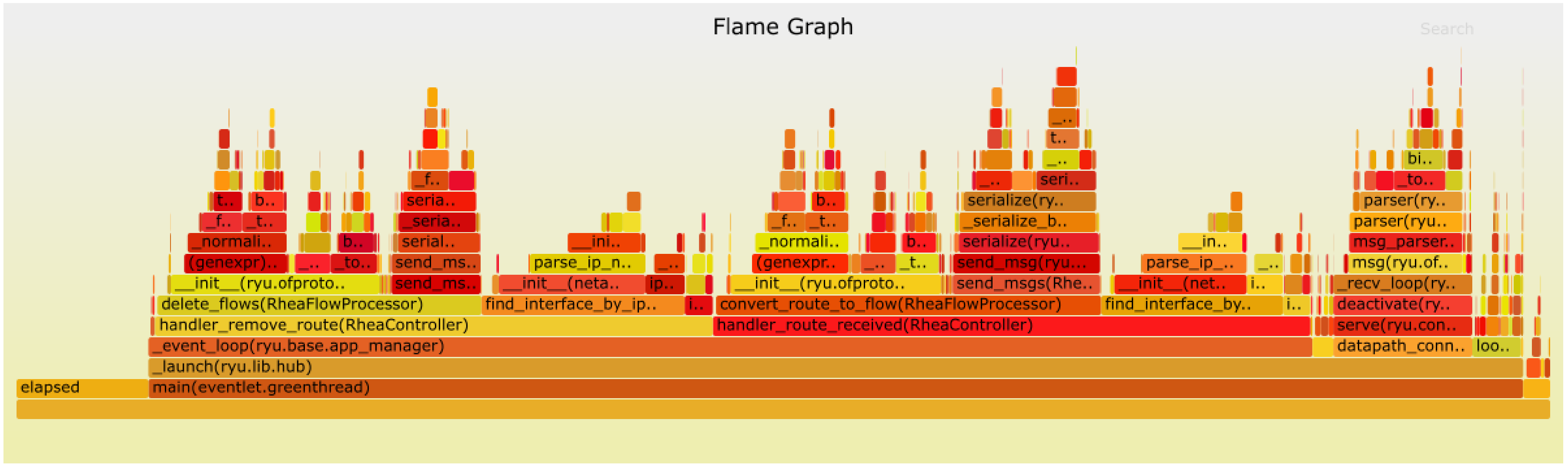

Figure 5.2: A Flame graph visualisation for the CPU profile of the RheaFlow code

5.2 Evaluation against RouteFlow

In this section, RheaFlow is evaluated against RouteFlow based on issues identified with RouteFlow in Section 3.3 and design goals discussed in section 4.1.

- ***Static Configuration Versus Dynamic Configuration***

  The virtual switch and interfaces connected to RouteFlow are manually configured and the mapping between the virtual switch interfaces and the OpenFlow switch ports have to be included in the RouteFlow configuration file before the application is started. This makes it difficult to make changes to the network without restarting RouteFlow. The static mapping of the virtual interfaces to OpenFlow switch port in RouteFlow means failure events on the OpenFlow switch ports are not replicated to the VMs connected to the mapped virtual interfaces, which would put the network in an inconsistent state. Furthermore, as stated in Section 3.3, RouteFlow requires two configuration files for *fast path* apart the default configuration file which is used for *slow path.* Apart from that, RouteFlow configuration requires extra tasks such as determining the identifier of the virtualised routing engine before hand. This identifier is required in the configuration file because it is used to setup the RFProtocol IPC channel between RFClient and RFServer. This identifier can not be changed once RouteFlow is in operation. RheaFlow on the other hand, uses a single configuration file for both *fast path* and *slow path.* It also does not require any extra tasks apart from starting the applications. As mentioned in Section 4.3.1, RheaFlow does the heavy lifting during the configuration by managing the virtual switch and maintaining an association table to keep track of OpenFlow switch port to virtual interface maps. This is something RouteFlow does not do.

  RheaFlow dynamically configures the virtual switch and interfaces connected to it using a combination of the ports' details specified in the configuration file and port state information reported by the OpenFlow switches when they connect to the controller. RheaFlow also performs the mappings between virtual interfaces and the OpenFlow switch ports without user intervention. These mappings are only completed if a port detail exists in the configuration file for each port reported to the controller by the OpenFlow switch when it connects to RheaFlow. The dynamic configuration used in RheaFlow ensures that it is able to respond to changes in the state of OpenFlow switch ports included in the configuration file and modify the mappings, the virtual interfaces and flow entries on the OpenFlow switch accordingly without restarting the application. However, additional ports that are enabled on the OpenFlow switch that should be mapped to virtual interfaces after RheaFlow has been started requires that the configuration file be updated with details for the new OpenFlow switch port and RheaFlow restarted.

- ***Ease of Virtualisation***

  The IP routing engines used in RouteFlow are run in VM instances on the same machine. While this may reduce the cost of the hardware used in the network and the size of the physical topology, it complicates the virtualisation of the RouteFlow application. This is because virtualising RouteFlow would require nested virtualisation in order to run the virtualised IP routing engines that would be mapped to OpenFlow switches. The use of nested virtualisation would increase the complexity of the configuration required for RouteFlow and introduce a single point of failure in the network.

  The RheaFlow architecture on the other hand can be easily virtualised without increasing the complexity of the network or introducing a single point of failure. The BIRD IP routing engine providing route information to RheaFlow can be on the same machine as the rest of the RheaFlow application or on a separate machine, this ensures that the logical and physical topology between the router and OpenFlow switches are the same. It also simplifies the deployment of RheaFlow in a network.

- ***Intermediate Protocol***

  An intermediate RFProtocol and IPC channel is required for exchanging messages between RouteFlow components. This is needed because of the distributed design of the RouteFlow architecture where RFClient is inside the VM hosting the IP routing engine in the RouteFlow setup. This increases the overhead to the network because control information being exchanged between the IP routing engines and the OpenFlow switches must be converted into RFProtocol.

  RheaFlow design completely eliminates the need for an intermediate protocol to exchange messages between its component. It provides a JSON based interface to receive control information from IP routing engines and uses the events provided by the Ryu API to exchange information between its various components. This design, however, makes the RheaFlow application dependent on the stability of the Ryu API which as noted earlier has a good track record.

- ***Ease of Customisation***

  The complexity of the RouteFlow architecture makes modification difficult. For example, in its default configuration, RouteFlow only delivers BGP protocol messages from the OpenFlow switches to the virtualised IP routing engines. To support additional routing protocols, new RFProtocol messages would need to be created. Also, the flow entries generated by RFServer would require modification to forward the new routing protocols. RheaFlow is agnostic about the routing protocols supported by the IP routing engine as it ensures all packets are delivered to the network stack of the machine running RheaFlow. Also, the RheaFlow design makes it easy to add new features. For example, adding new features requires creating an event handler in RheaController to process the events generated as a result of the new feature added.

- ***Fast Path Operations***

  As mentioned earlier, the virtual switch in RouteFlow is manually configured and the mappings are done statically based on the configuration file loaded. This requires the virtual switch to still send VirtualPlaneMap RFProtocol packets to the controller in order to map a virtual router to an OpenFlow switch for *fast path*. This increases configuration overhead for the application. The RheaFlow fastpath configuration does not require any packets to be forwarded to the controller in *fast path*.

- ***Lines of Code*** Compared to RouteFlow, RheaFlow provided the same set of functionalities available in RouteFlow with fewer lines of code. RheaFlow was written entirely in Python with 3540 lines of code. RouteFlow on the other hand was written in C++ and Python, the RouteFlow Vandervecken branch examined in this project contained 4527 lines of C++ code and 3422 lines of Python code. While RheaFlow's smaller code base may make it faster than RouteFlow, an analysis comparing the performance of both applications was not undertaken in this project. One reason identified for the large size of RouteFlow's code is the distributed nature of the architecture in which RFClient is located on a separate machine from RFServer. Communication between these RouteFlow components required a separate message format (RFProtocol) and IPC channel which had to be included in the code. Although RFProtocol and IPC channel increased RouteFlow's code base, it also enabled RouteFlow to be agnostic of the OpenFlow controller platform used to communicate with the OpenFlow switches. This ensures RouteFlow is not tied to a single OpenFlow controller platform. RheaFlow on other hand is tied to the Ryu OpenFlow controller platform because its applications are based on Ryu's API. However, Ryu is an open source controller platform that a good track record in terms of stability and backward compatibility. Utilising the Ryu API in the RheaFlow code ensured a simple modular SDN router architecture that can be easily customised in future.

Chapter 6

Discussion

In this chapter, the experience and the challenges encountered during the RheaFlow implementation are briefly discussed. RheaFlow is compared to other SDN routers in Chapter 2 based on the features highlighted earlier. Some areas of RheaFlow identified for future work are also discussed.

6.1 Design Experience and Challenges

The Ryu API enforces an event driven programming approach, as asynchronous events are the means with which Ryu applications communicate with each other. This event driven programming approach complicated the receipt and processing of netlink messages by pyroute2, the Python netlink library used in this project. The netlink events relevant to the RheaFlow application can not be processed asynchronously like other events in the RheaFlow application. They needed to be processed and responded to immediately. Because of this, the applications handling netlink messages were given higher priority. It was, however, discovered during the testing phase that this design approach exhausted CPU resources on the machine running RheaFlow. Other applications in RheaFlow were blocked or starved because of the high priority given to netlink processing applications. This issue was resolved by separating the processing of netlink messages into a separate Python application using the pyroute2 libraries. This application receives the relevant messages and sends them via a callback to the RheaFlow application that generates events for netlink messages. This challenge provided a good understanding of developing applications on the Ryu platform.

During the stress test operation to assess RheaFlow's performance under load, it was discovered that the test OpenFlow switch disconnects from RheaFlow after the application has been sending lots of *FlowMod* messages to the OpenFlow switch. This disconnection from RheaFlow disrupts traffic forwarding on the OpenFlow switch. Upon further investigation, the cause of the disconnection was identified to be the lack of response to OpenFlow echo request messages sent by RheaFlow to the OpenFlow switch. This causes RheaFlow to assume the OpenFlow switch is dead and close the TCP socket used to send OpenFlow messages to the OpenFlow switch. OpenFlow echo request and reply messages are used to verify that the OpenFlow control channel between the controller and the OpenFlow switch is still open and usable. Further analysis of this failure showed that the OpenFlow switch disconnects from RheaFlow when routes are being withdrawn by the IP routing engine which causes RheaFlow to send *FlowMod* messages to delete flow entries for the routes. This happens because the software of the OpenFlow switch used in the test network is not optimised for finding a specific flow entry in a large flow table with over 100,000 flow entries. The OpenFlow switch was overwhelmed with the search for flow entries that RheaFlow had requested to be deleted and did not send an OpenFlow echo reply to RheaFlow. This problem was addressed by increasing the number of unreplied echo requests that should be sent be RheaFlow before the OpenFlow switch is disconnected.

6.2 SDN Router Features in RheaFlow

In Chapter 2, the following features were identified as important for an SDN router to have:

- An SDN implementation that is widely supported, to increase its chances of adoption by network operators.
- Ability to delegate forwarding to multiple SDN-enabled devices for scalability.
- A centralised management of the routing control plane in conformance with the SDN paradigm.
- Dynamic discovery of SDN-enabled devices, reconfiguration of SDN-enabled devices in response to network changes and a simplified configuration that allows for quick changes in response to rapidly evolving network demands with minimal disruption.

- Backward compatibility with existing routing control protocols.
- The absence of a separate control protocol which likely increases overhead.

RheaFlow is briefly evaluated against these features. Table 6.1 is an updated version of Table 2.1, it shows the features supported by RheaFlow compared to the other SDN routers examined. RheaFlow is based on OpenFlow which

SDN Router	SDN Specification	Multiple SDN-enabled Device Support	Centralised Control Plane	Dynamic Discovery and Configuration	Simple Configuration	Backward Compatibility	Separate Control Protocol
FIBIUM	OpenFlow	No	Yes	Partial	No	Yes	No
SoftRouter	ForCES	Yes	Yes	Yes	No	Yes	Yes
DROP	ForCES	Yes	Yes	Yes	No	Yes	Yes
SDN-IP	OpenFlow	Yes	Yes	Partial	No	Limited	No
Atrium	OpenFlow	Yes	Yes	Partial	No	Limited	No
RouteFlow	OpenFlow	Yes	No	No	No	Yes	Yes
RheaFlow	OpenFlow	Yes	Yes	Partial	Yes	Yes	Yes

Table 6.1: An Updated Comparison of SDN Router Features.

increases its chance of adoption by network operators. It is also able to offload forwarding to multiple OpenFlow switches to ensure scalability and it fully centralises its routing control plane. RheaFlow uses a YAML configuration file for simplicity and does not introduce a separate control protocol. RheaFlow is agnostic about the routing control protocol used to provide routing information, this makes it compatible with existing routing control protocol. RheaFlow is able to dynamically configure the virtual switch and OpenFlow switch ports included in its configuration file when it was started, it is able to respond to changes an OpenFlow switch included in the configuration file long after RheaFlow has been started. However, it is not able not configure OpenFlow switches or OpenFlow switch ports not included in its configuration file and discovered after it has been started.

6.3 Future Work

These are some of the areas of RheaFlow identified for future work:

- Complete the inter-switch configuration and forwarding between multiple OpenFlow switches connected to RheaFlow.
- Conduct further tests on RheaFlow with hardware OpenFlow switches.
- Ensure new ports on OpenFlow switches that were not included in the

configuration file can be configured and mapped to virtual interfaces without restarting RheaFlow.

- Conduct further research into failover and redundancy options for RheaFlow application.

Chapter 7

Conclusion

This thesis presents the design and implementation of RheaFlow, an SDN router that uses IP routing control logic to modify the forwarding behaviour on OpenFlow switches. It converts IP routing control logic into OpenFlow entries that are installed on OpenFlow switches enabling them to forward IP packets to other devices in the network that are not OpenFlow capable. RheaFlow improves upon RouteFlow, an existing SDN routing solution which also enables OpenFlow switches to forward IP traffic by translating IP control logic into OpenFlow rules. RheaFlow improves the process of translating IP control logic into OpenFlow rules by addressing issues identified in RouteFlow design such as:

- The manual configuration and static mapping of virtual interfaces to OpenFlow switch ports.
- The intermediate RFProtocol for exchanging messages between RouteFlow components.
- The distribution of routing applications across multiple VMs which can neither be notified nor respond to failure events on the OpenFlow switches.

The RheaFlow design utilises the Ryu controller platform and API to implement a modular RheaFlow SDN routing application that can be easily customised and extended. Building RheaFlow on the Ryu OpenFlow controller platform allows the components of the RheaFlow architecture to communicate with each without defining a separate control protocol. RheaFlow components communicate with each asynchronously using events classes provided in the Ryu API.

In RheaFlow, the routing control plane is centrally managed. This is achieved by moving the IP routing engine that provides routing information in to the same Linux machine with the rest of the RheaFlow application. This ensures that changes in the state of the port of OpenFlow switches connected to RheaFlow are replicated to the IP routing engine. The RheaFlow design also provides an application and platform independed data-interchange format to receive routing information from an IP routing engine. This approach enables RheaFlow to receive routing information directly from an IP routing engine using JSON. This makes it easy for RheaFlow to receive routing information from remote IP routing engines. It also makes it easy to modify RheaFlow to process other types of network control information.

The RheaFlow configuration and deployment process is simple and intuitive. It uses a single YAML configuration file that is easy to read and understand for the user. RheaFlow also hides the complexities of the configuration from the user. It handles the virtual switch configuration, it also dynamically configures and maps the virtual interfaces on the virtual switch to OpenFlow switch ports without user intervention.

References

[1] Josh Bailey. Vandervecken. [Online]. Available: https://docs.google.com/document/d/1r2QbRRTbq9ilpmPQSwg4MhbBTx3F5I1Jx0E4osqnsro/mobilebasic?pli=1. [Accessed 20 July 2016].

[2] R. Bolla, R. Bruschi, G. Lamanna, and A. Ranieri. Drop: An open-source project towards distributed sw router architectures. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, pages 1–6, Nov 2009.

[3] Advanced Network Technology Center. Routeviews project. [Online]. Available: http://archive.routeviews.org/route-views.sydney/bgpdata/2016.03/RIBS/rib.20160301.0000.bz2. [Accessed 27 July 2016].

[4] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[5] A. Doria, J. Hadi Salim, R. Haas, W. Wang, L. Dong, and R. Gopal. Forwarding and control element separation (ForCES) protocol specification, 2010.

[6] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 20:1–20:12, New York, NY, USA, 2008. ACM.

[7] Neil Horman. Understanding and programming with netlink sockets. [Online]. Available: https://people.redhat.com/nhorman/papers/netlink.pdf. [Accessed 23 June 2016].

[8] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. In-

finite cacheflow in software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 175–180, New York, NY, USA, 2014. ACM.

[9] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[10] Ayaka Koshibe. SDN-IP architecture. [Online]. Available: https://wiki.onosproject.org/display/ONOS/SDN-IP+Architecture, 2015. [Accessed 8 June 2016].

[11] CZ.NIC Labs. The BIRD internet routing daemon. [Online]. Available: http://bird.network.cz/. [Accessed 24 June 2016].

[12] T. V. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo. The SoftRouter architecture. In *In ACM HOTNETS*, November 2004. [Accessed 5 June 2016].

[13] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[14] Marcelo R. Nascimento, Christian E. Rothenberg, Marcos R. Salvador, Carlos N. A. Corrêa, Sidney C. de Lucena, and Maurício F. Magalhães. Virtual routers as a service: The routeflow approach leveraging software-defined networks. In *Proceedings of the 6th International Conference on Future Internet Technologies*, CFI '11, pages 34–37, New York, NY, USA, 2011. ACM.

[15] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII).*, New York, NY, USA, October 2009.

[16] Syed Naveed Rizvi, Daniel Raumer, Florian Wohlfart, and Georg Carle. Towards carrier grade SDNs. *Comput. Netw.*, 92(P2):218–226, December 2015.

[17] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux netlink as an IP services protocol, 2003.

[18] Nadi Sarrar, Anja Feldmann, Steve Uhlig, Rob Sherwood, and Xin Huang. Fibium-towards hardware accelerated software routers. *EuroView 2010 (poster session)*, 9:1–17.

[19] Peter V. Saveliev. Pyroute2. [Online]. Available: http://docs.pyroute2.org/general.html. [Accessed 23 June 2016].

[20] Jonathan Philip Stringer, Qiang Fu, Christopher Lorier, Richard Nelson, and Christian Esteve Rothenberg. Cardigan: Deploying a distributed routing fabric. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 169–170, New York, NY, USA, 2013. ACM.

[21] Open Network Operating system Project. Atrium: A complete SDN distribution from onf. [Online]. Available: http://onosproject.org/wp-content/uploads/2015/06/PoC_atrium.pdf, 2015. [Accessed 6 June 2016].

[22] Nippon Telegraph and Telephone Corporation. [Online]. Available: http://osrg.github.io/ryu/index.html. [Accessed 2 June 2016].

[23] Allan Vidal, Fábio Verdi, Eder Leão Fernandes, Christian Esteve Rothenberg, and Marco Rogério Salvador. Building upon RouteFlow: a SDN development experience. In *31 Brazilian Symposium on Computer Networks and Distributed Systems*, SBRC 2013, pages 879–892, May 2013. [Accessed 02 June 2016].

[24] Stefano Vissicchio, Laurent Vanbever, and Olivier Bonaventure. Opportunities and research challenges of hybrid software defined networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):70–75, April 2014.

Appendix A

Configuration File Sample: config.yaml

```
---
# validate file at http://www.yamllint.com/
datapaths:
   - name: br-dpdk
     type: OpenVswitch
     dp_id: 0000a0369f5d0a2c
     vs_port_prefix: br-dpdk-p
     ports: { 5: ['20.0.0.254/24'],
              6: ['30.0.0.254/24'],
              7: ['40.0.0.254/24'] }
     fastpath_port: 8
     fastpath_vs: 1002
     isl_port:
     isl_rem_port:
     isl_rem_dp_id:
     rem_port: [5,6,7]

# Settings/Configuration for
# the virtual switch dp0
# change fastpath interface to eth2
# on prod
Virtual-switch: {
  fastpath_interface: eth2,
  fastpath_port: 1002
}
```

Appendix B

BIRD Configuration File: bird.conf

```
/*
 *  RheaFlow BIRD configuration file.
 */


# Configure logging
log syslog { info,trace,remote,warning, error, auth,
   fatal, bug };

# Override router ID
router id 172.16.0.45;

# This pseudo-protocol watches all interface up/down
   events.
protocol device {
  scan time 10;   # Scan interfaces every 10 seconds
}

protocol static {
    route 20.0.0.0/24 via 20.0.0.254;
    route 30.0.0.0/24 via 30.0.0.254;
    route 40.0.0.0/24 via 40.0.0.254;
    route 46.0.0.0/24 via 20.0.0.2;
    route 47.0.0.0/24 via 30.0.0.1;
    route 48.0.0.0/24 via 40.0.0.1;
}

```

```
protocol sdn MySDN {
}


protocol bgp {
  local as 2;
  neighbor 192.168.2.1 as 1;
  multihop;
  import all;
  source address 192.168.2.103;
}

protocol bgp machine20 {
  local as 2;
  neighbor 20.0.0.1 as 3;
  direct;
  import all;
  source address 20.0.0.254;
}

protocol bgp machine21 {
  local as 2;
  neighbor 20.0.0.2 as 6;
  direct;
  import all;
  source address 20.0.0.254;
}

protocol bgp machine30 {
  local as 2;
  neighbor 30.0.0.1 as 4;
  direct;
  import all;
  source address 30.0.0.254;
}

protocol bgp machine40 {
  local as 2;
  neighbor 40.0.0.1 as 5;
  direct;
  import all;
  source address 40.0.0.254;
}
```