TRINITY COLLEGE DUBLIN

MCS DISSERTATION

End-to-end Capacity Reservation in Software Defined Networks

Author:
CIARAN EGAN
11450212

Supervisor:
Dr. Marco Ruffini

A dissertation submitted in fulfillment of the requirements for the degree: ***Integrated Masters In Computer Science***
at the
School Of Computer Science & Statistics

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Trinity Term (May 2016)

Declaration Of Authorship

I, Ciaran Egan, declare that the following dissertation (“End-to-end Capacity Reservation in Software Defined Networks”) except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signed: ____________________________________

Date: ____________________________________

Summary

Software Defined Networking is a recent emerging field of networking. It provides enhanced programmability, cost-effectiveness and increased levels of experimentation to networks. Beginning with the introduction of the OpenFlow protocol in 2008, it has been adopted by many companies including Google, who use it in their data centers. With the growth of Internet traffic levels, guaranteed levels of bandwidth are becoming more of a necessity for online services particularly media streaming platforms. The aim of this dissertation is to provide an insight into the field of Software Defined Networking. It's contribution to the field is an examination of capacity reservation in SDN and what open source SDN controllers can offer networking operators in implementing BoD.

This dissertation outlines the necessary steps to develop a bandwidth reservation system using Ryu as the controller. It provides a listing of results gathered over the course of the implementation which can be used as a foundation for further work in the field or as a point of reference for anyone aiming to implement such a system in their network.

The results gathered over the course of the dissertation indicate that open source SDN controllers can led sufficient bandwidth reservation mechanisms to network operators. However, it has been ascertained that certain functionality is lacking. Aspects such as a lack of online resources and limited topological functions means there is room for future work and improvements. As the field continues to grow in popularity, the shortcomings it currently exhibits are expected to diminish.

TRINITY COLLEGE DUBLIN

Abstract

Computer Science Department

School Of Computer Science & Statistics

Integrated Masters In Computer Science

End-to-end Capacity Reservation in Software Defined Networks

by Ciaran Egan

Software Defined Networking (SDN) is an emerging paradigm in computer networking that enhances network programmability by providing an interface to the control plane of networking devices. Bandwidth is the lifeblood of the Internet and as a result Bandwidth on Demand (BoD) is becoming an increasingly important requirement for service providers with the continuous growth of Internet traffic levels. The aim of this dissertation is to provide an insight into the field of Software Defined Networking. It's contribution to the field is an examination of capacity reservation in SDN and what open source SDN controllers can offer networking operators in implementing BoD. Through describing the steps necessary to implement a bandwidth reservation system using Ryu, an open source SDN controller, this dissertation highlights that open source SDN controllers provide sufficient functionality to carry out capacity reservation. Although lacking functionality in certain areas, the results gathered indicate that open source SDN controller solutions can compete with their proprietary equivalents.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Marco Ruffini, for all the help and resources he provided me throughout the course of this dissertation.

I would like to thank my family for all the support and encouragement they gave me throughout the time I spent on this dissertation.

Finally I would like to thank my fellow classmates who were always there to offer their help for the duration of my time spent in college.

Contents

List Of Figures

List Of Abbreviations

SDN	**S**oftware **D**efined **N**etworking
BoD	**B**andwidth on **D**emand
NOS	**N**etwork **O**perating **S**ystem
QoS	**Q**uality of **S**ervice

Chapter 1

Introduction

1.1 The Subject of Research

Software Defined Networking has become an increasingly popular paradigm of computer networking since the introduction of the OpenFlow protocol[McKeown et al., 2008] in 2008. By providing an interface to the control plane of Ethernet switches (where decisions related to the forwarding of traffic are made), OpenFlow enables a centralised control point for managing traffic flows. Changing traffic routing decision-making from the traditional distributed approach to a centralised solution provides a comprehensive overview of the network in addition to new levels of programmability. SDN provides a cost-effective and extensible infrastructure that has extended it's use cases to managing the data centres of large companies such as Google[Jain et al., 2013]. As the number of networking device vendors supporting OpenFlow continues to grow, continued innovation in the field is expected.

As more networks migrate towards software and virtualized solutions, mechanisms such as Bandwidth on Demand (BoD) are being re-implemented in an SDN context. Implementing BoD in a network allows for a specific level of bandwidth to be requested and reserved over that network for a specific length of time. However, due to the inflexible nature of non-SDN driven networks, reservations can take up to weeks to process. The programmability and flexibility provided by SDNs mean that such bandwidth requests could be granted instantaneously.

This thesis is an examination of bandwidth reservation in SDN and what open-source controllers can lend to network operators in implementing BoD in their networks. It is intended to provide an evaluation of the level of support for bandwidth reservation mechanisms provided by Ryu, an open-source controller. Quality of Service (QoS) is the overall performance of a network and is measured by attributes such as latency, bit rate and throughput. Numerous mechanisms exist to guarantee QoS across a network and this thesis aims to examine how these mechanisms can be implemented and perform in an SDN context. Finally, it examines what system components are required in order to develop an SDN-driven bandwidth reservation management system.

1.2 Motivation for Research Topic

There are a great many benefits provided by SDNs over their traditional counterparts. Their programmable nature provide reduced maintenance costs and improved levels of control over network state. Their ability to transform simple switching devices into an array of middleboxes reduces costs as network operators can focus on a reduced set of device command sets. On-the-fly modifications of traffic flows provide mechanisms for diverting traffic away from damaged nodes, providing networks that are more tolerant to failure.

Internet traffic will exceed a zettabyte (1 trillion gigabytes) in 2016 and is expected to double by 2019[*Cisco Global IP Traffic Forecast* 2015]. The emergence of online media streaming services has created a need for a networking infrastructure where bandwidth can be reserved in a timely manner without requiring large-scale network modifications.

The novelty of the field could potentially lead network operators to use trusted vendor SDN controllers. This dissertation aims to show that open source controllers provide ample functionality and can compete with their proprietary counterparts.

1.3 Dissertation Overview

Section 2 provides background information on SDN, OpenFlow and SDN controllers. It highlights the motivation behind SDN by discussing the shortcomings of modern IP networks.

Section 3 is a description of the implementation steps that were taken throughout the course of the dissertation and it provides an overview of each component of the system.

In section 4, the results of the implementation are discussed. A description of what Ryu can offer in implementing capacity reservation is detailed.

Section 5 contains an overall evaluation of the implementation and the challenges faced, an evaluation of the results gathered and any future work.

Chapter 2

State of the Art

This section will provide an overview of the need, uses and key components of SDNs.

2.1 LAN Switching

LAN switching is a type of packet switching utilised in Local Area Networks (LANs). It is enabled by a series of connected network switches that forward traffic to it's destination device. The main type of switching that is relevant to this dissertation is layer 2 switching.

Layer 2 operates at the data link layer of the OSI model[Zimmermann, 1988] and forwards packets based on their destination MAC address. Each switch has a collection of ports and a MAC address table that maps MAC addresses to the port traffic must take in order to reach that address. In the event of a switch not having an entry for the MAC address in it's table, it floods all of it's ports with a broadcast message requesting the location of the MAC address. Each connected switch relays this broadcast message to all of it's neighbours until eventually a switch replies. This reply is traced back to the original switch that initially requested the location of the MAC address and MAC tables are updated by each switch along the way to reflect the newly discovered MAC address.

In section 2.2.2, OpenFlow is discussed which provides an overview of the type of functionality augmentation it can provide to switches.

2.2 Software Defined Networking

2.2.1 Issues with Traditional IP Networks

To illustrate the motivation behind SDN, this section provides an account of some of the difficulties associated with traditional IP networks. Firstly, they are complex and difficult to manage[Benson, Akella, and Maltz, 2009]. Traditional computer networks can be divided into three planes:

- **The data plane**: The networking devices that forward data across the network.
- **The control plane**: The protocols that populate the tables used by the forwarding devices in the data plane such as Open Shortest Path First (OSPF)[Moy, 1998].
- **The management plane**: These are the software tools that monitor and interact with the control planes of networking devices and are generally Command Line Tools (CLIs) or Graphical User Interfaces (GUIs).

Both the control and data plane reside on the networking devices in traditional IP networks. This was so to provide resilience to networks and also provided increased port densities and line-rates. However, this also provides very little flexibility for IP networks as the control and data planes are vertically integrated.

Middleboxes[Jamjoom, Williams, and Sharma, 2014] were added to improve a networks functionality but this increased the complexity of networks as the heterogeneity of the network devices adds increased command sets for network operators. These devices such as loadbalancers and firewalls need to be strategically placed in a network. Reconfiguring of devices to handle failure cases, updates and infrastructure changes can be an arduous process that usually requires taking them offline, leading to downtime of network segments. Innovating, evolving, designing and managing networks is restricted by it's inflexibility. Topologies can't be altered as easily without providing updates to multiple devices. An example of the near-immutability of IP networks can be seen in the upgrading from IPv4 to IPv6 in IP networks. Over 10 years has been spent upgrading the address space and it isn't expected to completely replace IPv4 until May 2148[Prince, 2013] at the current rate.

2.2.2 Towards SDN

SDN is a relatively new paradigm in computer networking. The term first came to prominence with the release of *OpenFlow: Enabling Innovation in Campus Networks*[McKeown et al., 2008] in 2008. However the idea of programmable networks has been around long before the inception of OpenFlow. The idea to separate the control and data planes of devices has been around since the 1980s with projects such as NCP[Sheinbein and Weber, 1982] making use of the global view of the network. In 1997, active networks[Tennenhouse et al., 1997] was an early attempt to enable data plane programmability. It enabled nodes to modify packet attributes and to perform computations on packet values. Another project that influenced the development of OpenFlow are programmable ATM networks[Lazar, Lim, and Marconcini, 1996]. The architecture that was developed provided an open programmable environment that enables the creation of services or network applications.

Building and combining ideas from these projects led to the introduction of the OpenFlow[McKeown et al., 2008] standard: a protocol for interfacing capabilities with flow tables of Ethernet switches. This interface provides functionality to remove the vertically integrated nature of the control and data planes of network devices by moving their control logic to a remote, centralised controller. The aim of OpenFlow was to encourage experimentation in networks. The inflexibility of networks mentioned in section 2.2.1 meant any experimentation would require considerable time and effort due to the static nature of their organisation. Researchers at Stanford found that OpenFlow allowed them to segment sections of their campus network for experimentation without affecting the overall performance of efficiency of the network. Traffic flows were partitioned between production and research flows allowing for new protocols to be tested without having to build an entirely new research networking infrastructure.

As SDN has evolved to it's current state, there are a number of defining characteristics that are used to quantify it:

- The control plane (where decisions about where traffic is sent are made) and data plane (where the forwarding of traffic is performed) of network devices are separated, transforming their function to simple forwarding devices.
- With the decoupling of control and data planes, the control planes of the forwarding devices are managed remotely in a logically centralised controller

called an SDN controller or Network Operating System (NOS).

- Forwarding decisions are flow-based (a sequence of packets between two nodes: source and destination) rather than the existing distributed approach of destination-based.
- As a result of the three characteristics above, SDNs are programmable.

The programmability of SDNs are considered their primary selling point. They enable network operators to dynamically alter flow tables. This provides the ability to modify network policies on-the-fly. Flows can be redirected around nodes that are down. The centralised nature of the networks control plane provides an easy to understand and comprehensive overview of the entire network. This simplifies updating, modifying and building applications for the network. The controller is hosted on commodity server hardware and so various implementations with APIs in high-level languages are available. The flow-based concept means that various networking devices such as routers, switches, loadbalancers and other middleboxes[Jamjoom, Williams, and Sharma, 2014] devices are unified in functionality, providing increased flexibility to network operators.

OpenFlow

OpenFlow is a protocol that provides access to the forwarding planes of network devices. First implemented in a campus network in Stanford University in 2008[McKeown et al., 2008], OpenFlow provides access to a devices control plane allowing for flow-table entries to be added, removed and modified instantly without needing to take the device offline. OpenFlow switches are composed of at least three parts:

- A flow table. A table that performs actions on incoming packets based on filtering criteria.
- A secure channel. This is for communication between the switch and the remote control point for the switches forwarding logic.
- The OpenFlow protocol itself. It provides a way for the controller to interact with a switch and means that network operators don't need to program the switch individually.

Once a switch complies with the three requirements above, it can operate in an SDN context. There is an increasing number of networking device vendors that support

OpenFlow. The design of an OpenFlow switch means that vendors can support OpenFlow without having their internal workings exposed. In addition to vendors such as Cisco[Cisco, 2015] and Pica8[Bill Oliver, 2014] providing support for OpenFlow, there are also open-source and virtual switch solutions such as Open vSwitch[Pfaff et al., 2015] that can operate as OpenFlow devices.

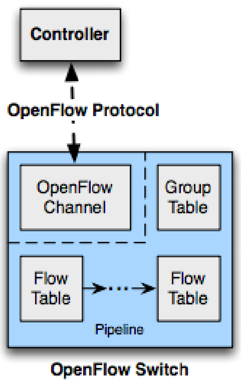


FIGURE 2.1: OpenFlow switch design.
McKeown et al., 2008

OpenFlow-compatible switches contain a flow table. A flow table can easily be substituted for a MAC table but the relationship is not reversible. Each flow table entry contains three fields:

- A rule field. This is a label for identifying (matching) packet headers. The protocol currently supports a multitude of headers such as Ipv4, Ipv6, Ethernet, TCP/UDP and MPLS.
- An action field. This is for defining the action to be taken upon the matching criteria in the rule field being met. Examples include, outputting the packet to a specific port, outputting the packet to the controller for further inspection, drop the packet for QoS or security reasons, or pass the packet to another flow table where further matching and actions can occur.
- A stat field. This monitors the frequency of occurrence of a particular flow entry.

The types of messages provided by OpenFlow can be categorised into two types: switch-to-controller and controller-to-switch messages. Messages from the controller to switch are primarily pertaining to the management or retrieval of the switches state. Examples of such messages include a modify-state message (how flow entries are added, deleted or modified) and a read-state message that returns statistics on the switch. Messages intended for the controller that originate from a switch are for the most part asynchronous. These messages include error messages (how the switch indicates to the controller that it is experiencing problems), packet-in message (the switch forwards an incoming packet to the controller either purposely or die to no matching fields in the flow table).

To demonstrate the OpenFlow protocol in action, a description of the messages passed between the switch and the controller to establish a connection, retrieve stats from the switch and modify a flow are as follows[1]:

1. The switch sends an OFPT_HELLO message to notify the controller of it's existence in the network. This message includes the value of the highest version of OpenFlow that it supports. The controller chooses the version of OpenFlow that it will use based on the clients supported versions and replies with an OFPT_HELLO message containing the version that will be used throughout the session.

2. The controller sends an OFPT_FEATURES_REQUEST message to the switch. This message contains just a header and queries the switch for it's capabilities. The switch responds with a OFPT_FEATURES_REPLY message which returns information such as the number of tables present, the switches datapath ID, the list of it's ports, it's list of available OpenFlow actions (examples include outputting to specific port or modifying packet headers) and it's list of OpenFlow capabilities (examples include monitoring flow and table statistics).

3. To retrieve statistics on a switch, the controller sends an OFPT_SET_CONFIG message to the switch outlining configuration settings such as the maximum bytes of packet it should send to the controller and the set of configuration flags to be used. At this point, a successful connection has been established between the switch and controller.

4. The controller can request the state from the switch using the OFTP_MULTIPART_REQUEST message. The types of data that can be returned include statistics on various flows, tables or ports amongst others. In addition to these statistics, information such as port descriptions and table features can be requested in this message. Multiple request types can be chained together in a single multipart request. The switch replies with a

[1]Assume a standard TCP, SCTP or TLS/SSL has been established between the switch and controller.

OFPT_MULTIPART_REPLY with an attached list of replies to the controllers requests.

5. To add, delete or modify a flow entry on a switch, the controller sends a FLOW_MOD request. Examples of flow-modification messages include OFPFC_ADD, OFPFC_DELETE and OFPFC_MODIFY. In the event of a successful flow modification, the switch doesn't reply. In the event of an error occurring, the switch responds asynchronously to the controller with a OFPET_FLOW_MOD_FAILED message.

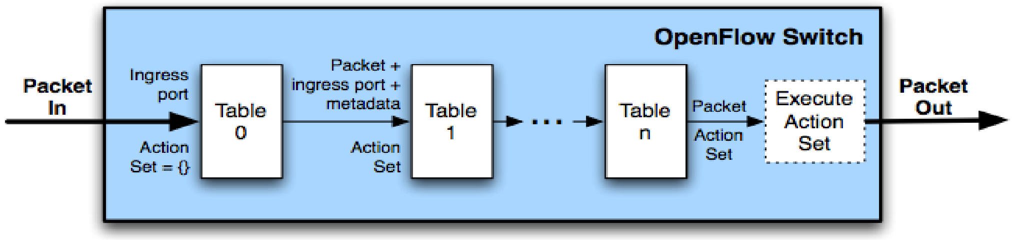


FIGURE 2.2: OpenFlow 1.3 switch detailing the chaining of tables to create a packet pipeline processing architecture. [*OpenFlow Switch Specification* 2012]

An OpenFlow-enabled switch needs to have at least one flow table. Flow tables are sequentially numbered beginning with 0. The creation and chaining of flow tables creates a processing pipeline for incoming packets. This pipeline is visualised in Figure 2.2. Packets can only be forwarded to tables in ascending numerical order meaning a packet cannot be processed by the same table twice thus preventing cycles. As a packet is processed by tables, it builds up it's action set and does not execute the series of actions until it has either reached the end of the pipeline, or a particular table has decided not to forward it to the next.

SDN Controller

The benefits provided by OpenFlow (or an alternative protocol) aren't scalable without an orchestrator component that provides a level of abstraction above the protocols command set. Having to add and modify flows on a per-switch basis every time a device is started up, although an improvement on non-OpenFlow compatible devices, isn't scalable. To alleviate this, an SDN controller is required. Also known as a Network Operating System (NOS), it acts as the "brains" of the network. It's a centralised point where all forwarding decisions for an SDN are made. It interacts with switches over what's called a southbound API or interface and communicates with business logic and applications via a northbound API where high-level programming languages can provide directives to the controller.

SDN controllers typically handle repetitive tasks such as the initial handshake and the reporting of OpenFlow errors. As mentioned above, using OpenFlow commands on their own isn't scalable. They provide an abstraction from the 'low-level' OpenFlow commands provided by the southbound interface into an easier-to-use package. The NOS facilitates in making network operations simpler[Gude et al., 2008a] and provides a comprehensive overview of the network state.

There are both centralised and decentralised controllers available. Centralised controllers provide a unified interface to the network but are also a single point of failure. These controllers are suited to smaller networks. As networks grow, more controllers need to be added and so decentralised controllers are favoured due to their interoperability support. Decentralised controllers, although more complex, can be scaled easier and are more tolerant to failure. Due to its dominance in the southbound API market, most controllers only support OpenFlow. Controllers also contain east and westbound interfaces which are what enables multiple controllers to communicate and coordinate between SDNs.

2.2.3 SDN Architecture

In Figure 2.3, the layers of abstraction provided by the SDN architecture are shown.

The lowest level of abstraction is the network infrastructure. It corresponds to the hardware network forwarding devices (switches, routers, loadbalancers etc.). As is a characteristic of SDNs, these devices only manage their data plane, the control logic has been removed. These devices support an interface such as OpenFlow which allows for the remote control of their control logic. Networking device vendors are now producing networking devices that support the OpenFlow standard. In addition, software switches such as Open vSwitch[*Open vSwitch - http://vswitch.org* 2013] that support OpenFlow are becoming more prominent in virtualized networks and data centers[Weissberger, 2013].

The southbound interface is an interface for the SDN controller to interact with the forwarding device. Open vSwitch Database Management Protocol (OVSDB)[Pfaff

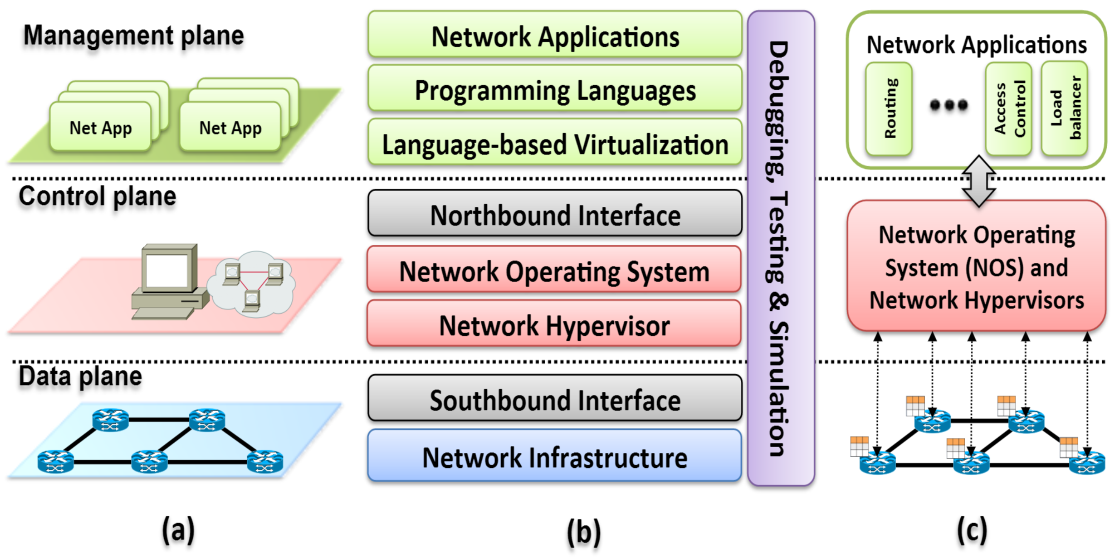


FIGURE 2.3: SDN architecture and it's fundamental abstractions as (a) planes, (b) layers and (c) system design architecture. [Kreutz et al., 2014]

and Davie, 2013a] and OpenFlow are examples of such interfaces. OpenFlow (developed by the Open Networking Foundation[ONF, 2014]) is the most widely received southbound interface and network device vendors such as CISCO and Juniper support it, providing further room for expansion for SDNs.

A network hypervisor allows for multiple virtual servers to be hosted on a single hardware device, sharing CPU, disk memory and other computing resources. This idea, called virtualization, is an already defined paradigm in computer science[Chowdhury and Boutaba, 2010]. Virtual servers or machines can be easily ported to different networking devices as needed providing flexibility and failover in the event of a physical server going down.

The fourth layer of the SDN abstraction stack is the network operating system or the controller. This is where the control logic of the forwarding devices is managed. It has been covered in section 2.2.2.

The northbound interface is an interface between the management and control planes. Where the southbound interface is a hardware interface between layers, the northbound interface is a software implementation. A lot of SDN controllers contain their own northbound interface[Ferro, 2012] such as NOX and OpenDaylight. However the variance in these APIs means that there is no de facto API, unlike how OpenFlow is widely accepted as being the primary southbound API. The emergence of a single northbound interface is difficult due to the nature of varying network applications built on top of this interface.

Language-based virtualization provides modularity to networks[Messaoud et al., 2014] and abstracts the complexity and dynamic nature of the physical infrastructure.

Programming languages provide the simplest level of abstraction for the development of SDNs. Here, programmers can write scripts in their preferred language of choice to modify the network state, which is more favourable than writing 'low-level' OpenFlow (or another southbound API) commands. This high level of abstraction provides functionality to easily modify the flow tables of forwarding devices. High level programming languages enable configurations to be modularised and reused. Management becomes easier and so more time is available to network operators to further develop and further innovate.

2.2.4 Use Cases and Benefits

The types of applications that can be developed using SDN are numerous. They can be grouped into (but not limited to) 5 main categories[Kreutz et al., 2014]:

- **Traffic engineering**: Examples include load balancing, router configuration and traffic optimisation applications.
- **Mobility and wireless**: Example is Light virtual access points (LVAPs)[Suresh et al., 2012] and AeroFlux, a hierarchical WLAN control plane.
- **Measurement and monitoring**: These applications are for supervising a network.
- **Security and dependability**: Examples include firewalls, access controls and middlebox implementations.
- **Data centre networking**: The flexibility and modularity of virtualized networks provide a perfect framework to build data centres upon. Combining the previous four application categories, responsive data centres can be built.

SDNs are cost effective. Middleboxes such as load balancers or routers can be implemented on switches using combinations of OpenFlow commands and features. Assuming the switches meet the correct criteria (such as a specific version of the southbound interface), combinations of flow entries can be used to emulate various types of middlebox. It is worth noting that switches cannot be a direct replacement for all middleboxes in every situation. Specialised hardware such as an SSL accelerator in a load balancer cannot be emulated on switches with the same level of performance. SDN controllers don't require specialised hardware to operate so can be hosted on commodity server hardware.

2.3 Open Source SDN Controllers

2.3.1 Overview

With the introduction of OpenFlow in 2008, a number of open source SDN controllers emerged that aimed to augment the newfound functionality provided by the protocol. They are typically organised as a collection of applications or modules that can be selectively executed. Out-of-the-box modules typically include APIs that provide information about the network topology or install router or almost any middlebox settings onto a switch to modify its behaviour. In addition to the array of modules offered by controllers, new ones or customised versions of existing ones may also be developed. ONOS[Berde et al., 2014], NOX[Gude et al., 2008b] and Beacon[Erickson, 2013] are examples of open source controllers that have been built on top of the functionality provided by OpenFlow.

2.3.2 Ryu

Ryu[Nippon Telegraph and Telephone Corporation, 2012] is a component-based software defined networking framework. Supported by NTT labs in Japan, Ryu is built entirely in Python. It is managed and sustained by the Ryu community. It provides SDN components or apps with APIs that are both configurable and easy to deploy. It has been certified to work with a multitude of switches including Open vSwitch.

Ryu is composed of a number of components or SDN apps. These provide a range of functionality including:

- Firewall API. This REST API provides an interface to configuring switches to behave as firewalls. Ryu also provides a similar app for implementing router functionality in a switch.

- QoS API. Ryu provides APIs to modify switch behaviour so that bandwidth can be allocated to flows. It supports mechanisms such as meter tables or Differentiated Services.
- Topology API. This is a REST interface that provides topological information such as switches and their associated ports, links and hosts.
- Switch flow-table API. This provides a REST API where flow entries for a switch can be retrieved, added, deleted or modified. Statistics on flows, flow tables, queues, meter tables and ports amongst many more can be obtained.

2.4 Capacity Reservation

To illustrate the motivation behind Bandwidth on Demand (BoD)[Bernstein, 2006] or capacity reservation in networking, consider two types of file transfer. The first is the transfer of a video file from one host to another across an IP network. Packets are forwarded through the network, some are dropped. The packets are sharing the same bandwidth as other streams of packets. As the bandwidth between two switches reaches maximum capacity, the switch drops[2] any packets it cannot send and they are re-sent by the host. The dropping and re-sending of packets in this instance is relatively acceptable because, assuming no major network failures, the video file will eventually reach its destination. Once the transfer is complete, the video can be viewed at the exact same quality as it was sent at.

Now consider the scenario where a video is being streamed live. Here, the rate at which packets are sent is important. In this case if packets are dropped, the quality of the video is affected. Services such as Netflix and Sky Go depend on a reliable end-to-end connection between hosts to provide their service to customers at an acceptable standard. They require the reservation of bandwidth in order to guarantee their service.

As mentioned before, IP networks are inflexible. The increasing amount of traffic passing through them[Korotky, 2013] would suggest that the likelihood of packet dropping is also increasing. The need for bandwidth reservation is becoming more prominent for companies that depend on reliable connections. As services request

[2]Switches have a small amount of memory dedicated to queueing packets it cannot forward at the time. For this example we will assume the buffer is full.

capacity, the current implementation would involve re-configuring individual switches to alter their data-planes to provide more capacity.

A software defined implementation of BoD could dynamically alter the bandwidth allocations of clients instantaneously upon request, while automatically scaling capacity back once their allowance expires. This would remove the laborious configuration changes required at present in IP networks. It can take weeks for service providers to allocate bandwidth in modern IP networks. An SDN-driven solution could complete such an allocation in a matter of seconds.

Chapter 3

Design & Implementation

3.1 System Components

3.1.1 Bandwidth Reservation System Architecture

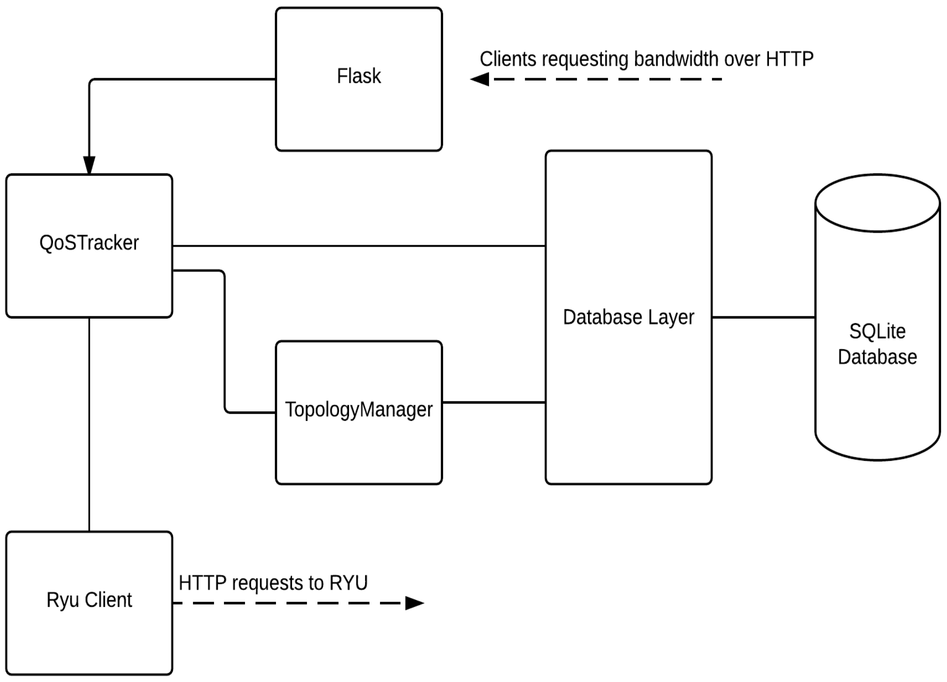


FIGURE 3.1: Architecture of the bandwidth reservation system.

Database Models

To maintain track of the network state, persistent storage of the relevant information was needed. The database schema is outlined in Figure 3.2.

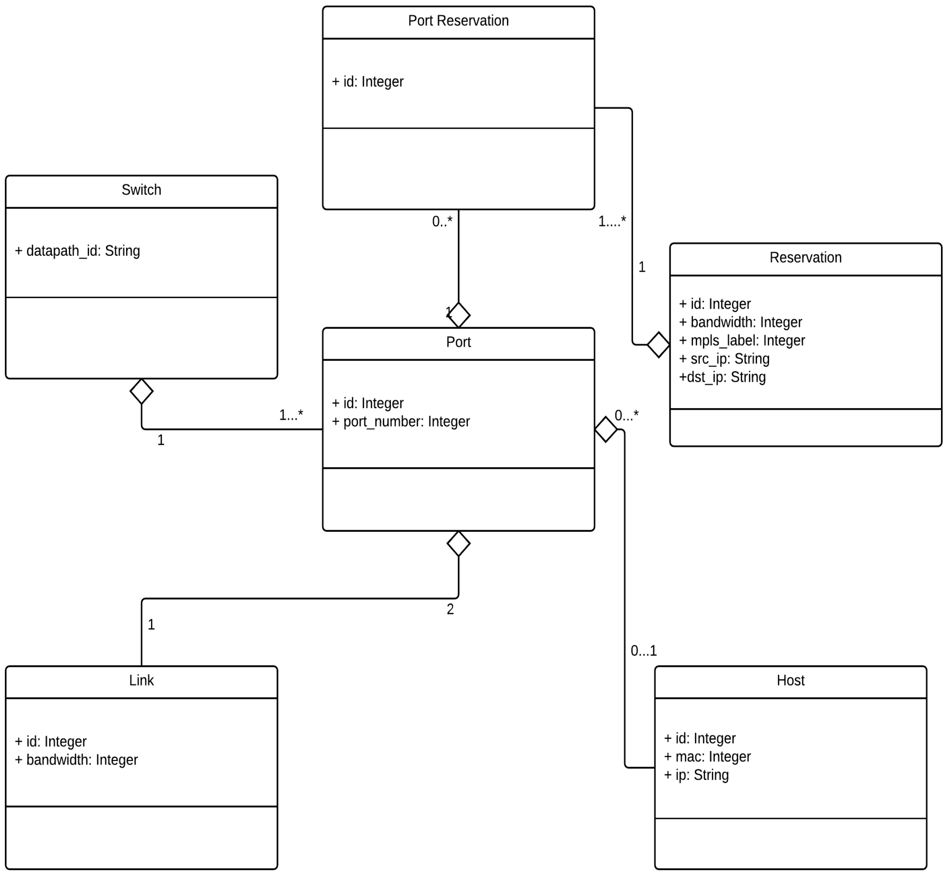


FIGURE 3.2: UML class diagram representing the database models.

An entry to the port table represents a port on a switch. It maintains a reference to it's switch by storing it's id. It is identified from other ports for a given switch by it's port_no column which is a representation of it's port number.

```
CREATE TABLE port (
        id INTEGER NOT NULL,
        switch INTEGER,
        port_no INTEGER,
        PRIMARY KEY (id),
        FOREIGN KEY(switch) REFERENCES switch (dpid)
);
```

An entry to the host table represents a host that is connected to the network. It keeps a reference to the ID of the port with which it first makes contact with when sending traffic through the network. This supports hosts that are connected directly to the network and hosts that have more than one hop between itself and the network. A copy of it's IP address and MAC address are also stored.

```
CREATE TABLE host (
        id INTEGER NOT NULL,
        mac VARCHAR,
        ip VARCHAR,
        port INTEGER,
        PRIMARY KEY (id),
        FOREIGN KEY(port) REFERENCES port (id)
);
```

An entry to the link table represents an inter-switch link. It stores the IDs of the two ports that it is connected to which can then be traced back to the switches via the port's 'switch' column. The bandwidth is stored as an integer with the value represented in bits.

```
CREATE TABLE link (
        id INTEGER NOT NULL,
        src INTEGER,
        dst INTEGER,
        bandwidth INTEGER,
        PRIMARY KEY (id),
        FOREIGN KEY(src) REFERENCES port (id),
        FOREIGN KEY(dst) REFERENCES port (id)
);
```

An entry to the switch table represents a switch in the network. The only value it maintains is it's datapath ID as it's primary key.

```
CREATE TABLE switch (
        dpid INTEGER NOT NULL,
        PRIMARY KEY (dpid)
);
```

A reservation row represents a bandwidth reservation. It maintains the IP addresses of a source and destination host along with the ingress port (the port the source host's traffic will first make contact with in the network) and the egress port (the port at which the reserved traffic leaves the network as it travels towards the desitnation host) of the flow. The bandwidth to be requested is an integer that represents the capacity to be reserved in bits. The mpls_label column was used as an initial implementation where traffic was identified on the network by means of allocating a value to each reserved flow to be set in the MPLS header. Future work may still avail of this functionality so the decision was made to leave it in the schema.

```
CREATE TABLE reservation (
        id INTEGER NOT NULL,
        src VARCHAR,
        dst VARCHAR,
        bw INTEGER,
        mpls_label INTEGER,
        in_port INTEGER,
        out_port INTEGER,
        PRIMARY KEY (id),
        FOREIGN KEY(in_port) REFERENCES port (id),
        FOREIGN KEY(out_port) REFERENCES port (id)
);
```

An entry to the port_reservation table represents a node (switch) along a reserved flow. It maintains a reference to a reservation table entry. By retrieving all port-reservations for a particular reservation, the path the flow takes through the network can be determined. It stores the ID of the ingress port to each switch along a flows path. Due to the deterministic nature of the function that maps a route from source and destination host, the flow for a particular reservation can be determined by re-running it. The reasoning behind including this table was to accommodate future work that would enable more dynamic flows that change their path in response to node failure or to make the best use out of the available bandwidth.

```
CREATE TABLE port_reservation (
        id INTEGER NOT NULL,
        port INTEGER,
        reservation INTEGER,
        PRIMARY KEY (id),
        FOREIGN KEY(port) REFERENCES port (id),
        FOREIGN KEY(reservation) REFERENCES reservation (id)
);
```

Database Layer

SQLAlchemy is an Object Relational Mapper (ORM) Python library that converts Python classes and their instances to database tables and rows. This removed the necessity to write raw database-engine specific queries. An example of the port table represented using SQLAlchemy is shown below.

```
class QoSHost(Base):
    """
    Class to represent a host.
    """
    __tablename__ = "host"

    id = Column(Integer, primary_key=True, autoincrement=True)
    mac = Column(String)
    ip = Column(String, nullable=True)
    port = Column(ForeignKey("port.id"))
```

Interactions with the database are made using SQLAlchemy's session object. This provides an interface to modify rows of a table that is similar to modifying attributes of a Python object. Attributes are either added, modified or deleted and the commit() function of the session class is called which takes any changes made to the SQLAlchemy objects and persists those changes in the database. This meant that database interactions didn't have to be made using a database-specific language.

The ability to abstract away from a specific database engine such as MySQL or SQLite is provided by SQLAlchemy's database connection object that handles multiple database engines. Although SQLite was chosen as the engine of choice during the implementation stage, SQLAlchemy's flexibility would allow for a MySQL or PostgreSQL database to be connected instead with minimal configuration changes.

Ryu-Interface

The RyuManager component of the system is dedicated to communicating with the Ryu APIs. Using Python's requests library, it takes the parameters required by the Ryu API from the QoSTracker component and correctly packages them into JSON format. The request is sent to the appropriate Ryu API endpoint and returns the returned response to QoSTracker.

The types of requests that are handled by the RyuManager are as follows:

- Configure the OVSDB address on a switch. This is so the switch knows where to access OVSDB from in the event of any needed configuration changes. It communicates with Ryu's rest_conf_switch API. It accepts a single parameter, the location of a running OVSDB process in the format <protocol>:<host>:<port> to the url 'http:<host>:<port>/v1.0/conf/switches/<switch_id>/ovsdb_addr'.
- Add a packet-marking flow. This request configures a particular switch to mark any incoming packets that are recognised as reserved traffic. The node that this is rule is added to will mark packets based on their priority by assigning a value to the DSCP header field in the IP packet. It determines whether traffic is to be given a higher priority by matching on the source and destination IP addresses in the packet header. This request is sent to Ryu's QoS rules API which pertains to adding rules for prioritising traffic. The QoS rules url is /qos/rule/switch-id and it accepts arguments in the following JSON format:

```
{
    "match": {
        "nw_dst": "10.0.0.1",
        "nw_src": "10.0.0.4",
        "nw_proto": "UDP"
    },
    "actions": {
        "mark": 26,
        "queue": 1
    }
}
```

  Upon the successful addition of the packet-marking rule, Ryu returns the following success message:

```
[
    {
        "switch_id":"0000000000000020",
        "command_result":[
            {
```

```
                        "result":"success",
                        "details":"QoS added. : qos_id=1"
                    }
                ]
            }
        ]
```

- Add a packet-checking flow. This adds a rule to check for a specific DSCP header value. If the desired DSCP value is present in the packet, it is added to the switch in questions higher priority queue. This request is made to Ryu's QoS rules API and accepts arguments in the following JSON format:

```
        {
            "match": {
                "ip_dscp": 26
            },
            "actions": {
                "queue": 1
            }
        }
```

  Upon the successful addition of the packet-checking flow, the following is returned in Ryu's response:

```
        [
            {
                "switch_id":"0000000000000030",
                "command_result":[
                    {
                        "result":"success",
                        "details":"QoS added. : qos_id=1"
                    }
                ]
            }
        ]
```

- Add QoS queue configurations. This request configures the maximum rates allocated to traffic flows. The maximum rate at which best-effort (non-reserved) traffic is defined along with the minimum rate guaranteed to reserved traffic.

It interacts with Ryu's QoS queues API at the /qos/queue/switch-id url and accepts the following parameters in JSON format:

```
{
    "port_name":"s0-eth1",
    "type":"linux-htb",
    "queues":[
        {
            "max_rate":"1000"
        },
        {
            "min_rate":"750"
        }
    ]
}
```

Upon the successful configuration of queues, Ryu's QoS queues API returns:

```
[
    {
        "switch_id":"0000000000000010",
        "command_result":{
            "result":"success",
            "details":{
                "0":{
                    "config":{
                        "max-rate":"1000"
                    }
                },
                "1":{
                    "config":{
                        "min-rate":"750"
                    }
                }
            }
        }
    }
]
```

Flask Server

The ability for clients (hosts) to request bandwidth was a necessary requirement for this system so an API hosted on an HTTP server to accept information about bandwidth reservations was required. Only one endpoint was necessary for the intended implementation: the ability to post a request to the server to request bandwidth. The only clients of this service would be hosts connected to the net so it's frequency of use and traffic volume would be very small.

Flask[Grinberg, 2014], a web micro-framework for Python, was chosen because it's lightweight and easy-to-use web-server provided the required functionality. It accepts a POST request to the 'add_reservation' endpoint and requires the following parameters in JSON format:

```
{
    "src": "10.0.0.4",
    "dst": "10.0.0.1",
    "bw": 75
}
```

Topology Layer

The TopologyManager component contains topological functions and acts as a go-between for the QoSTracker and the database layer. Not added until late into the implementation phase, this layer was required as QoSTracker was handling a lot of these functions and the codebase was becoming messy and unmanageable.

The first function housed by the topology layer is the retrieval of the maximum bandwidth for a particular path (sequence of switches). A path in this instance is a sequence of switches that represents a flow across the network. It is represented as an array of switches with the first entry being the source host and the last represents the destination host. It sequences through the switches, querying the database layer for a link entry between each switch. The bandwidth of each link is stored. At the end, the minimum value is returned as the maximum bandwidth as the flow's bandwidth is only as large as the smallest link.

The topology layer contains a function to get the available bandwidth for a particular path. This follows the same process as outlined in the get_max_bandwidth_for_path() description above. The only difference is that for each link it examines, it also checks for any existing bandwidth reservations on each link and subtracts them from the total bandwidth for each link. Once the available bandwidth has been calculated for each hop along the path, the smallest value is returned as the available bandwidth for the path.

Another function provided by the topology layer is one to find a route from source host to destination host. This function is a naive solution to discover a sequence of nodes across a network that join two hosts. It is a recursive function that operates on a single switch at a time, maintaining a reference to the previous switch so to prevent backtracking. The function accepts three arguments: the destination hosts IP address, the current switch that is currently being examined (set to the switch that the source host is connected to) and the switch it was previously examining (set to null for the first iteration).

```
def get_route_to_host(self, dst_ip, switch, prev_switch=None)
```

The currently implemented algorithm works as follows:

1. Query the database layer for any hosts attached to the switch.
2. If there are hosts connected to the switch, the function checks their IPs to see if they match the 'dst_ip' it is looking for. If it finds it, it returns the switch in it's own array. If there are no hosts connected or no matching host connected the function continues.
3. Get all of the switches that are connected to the current switch. If 'prev_switch' is not null, it is excluded from the results returned.
4. If there are no neighbours, return null as this is the equivalent of the function reaching a 'dead-end'. Otherwise, iterate over the neighbours, re-calling itself with a neighbouring node as the node to search. If the function returns a path (i.e. if it has found the host), the function returns this path, and adds the current switch to the path (array of switches) as the first entry of the array, pushing the other path elements to the left of the array by one. This is so the ordering of the array represents the ordering of the nodes the traffic would traverse across the path.
5. Once a path has been determined, it is returned.

This aspect of the system requires more work. This currently returns the first viable route to the host it discovers and not the best. The option to return multiple paths would be an improvement and taking the bandwidths of links into consideration would also yield a far more efficient path discovery function.

Capacity Reservation Manager (QoSTracker)

The QoSTracker component of the system acts as an orchestrator for the various components and handles all of the business logic for the system. It's main tasks include:

- Upon starting the system, it initiates the RyuManager, TopologyManager and DBConnection components.
- It handles incoming requests from the Flask server.
- Determines what QoS rules and queue configurations are necessary to enable a particular bandwidth reservation and passes the relevent information to the RyuManager to be sent to Ryu's northbound API's.

3.1.2 Network Topology

Mininet[Lantz, Heller, and McKeown, 2010] is a software emulator for prototyping realistic virtual networks on a single machine. Testing the bandwidth reservation system against a set of real hardware switches would have been time-consuming (to set up) and inflexible (in it's abilities to change topologically). Mininet offered an easier way to test the bandwidth reservation system against. Rather than having to wire up, configure and turn on a set of physical switches, the network topology can instead be ran from within the same computer that the other system components are hosted.

Mininet can create a network topology via a terminal command or programmatically by calling the Mininet API from within a Python script. The latter option was chosen for this project due to the increased levels of control provided by the API over the command line interface. The Mininet API allows for the creation and naming of hosts and switches, the definition of links between them, the types of firmware to be installed on the switches, the type and location of a controller and the ability to set link bandwidth.

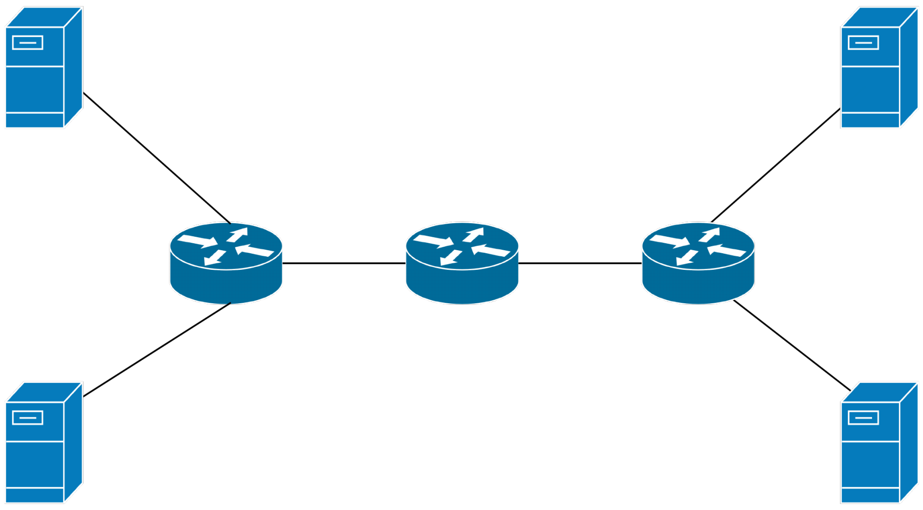


FIGURE 3.3: Network topology diagram.

Throughout the course of development, one particular topology was used to test more than others. It can be seen in Figure 3.3. This topology was the topology that was chosen to test against for the following reasons:

- For any traffic to pass between two hosts on opposite sides of the topology, at least two hops are needed. This meant that to add a reservation, multiple switches needed to be configured with different rules (ingress nodes contain the packet-marking rule while the others contain the packet-checking flow).
- It is linear in nature for simplicity sake. The route-finding algorithm does not account for cycles or multiple path choices.
- An old design of the project involved marking reserved traffic with MPLS labels. Having three switches meant that there was one node to test that the MPLS label was being added correctly, one switch to assert the MPLS label carried throughout the network and the egress node was used to validate that the MPLS label was being correctly removed as it leaves the network.

```
host0 = self.addHost('h0')
host1 = self.addHost('h1')
host2 = self.addHost('h2')
host3 = self.addHost('h3')

switch0 = self.addSwitch('s0', dpid='0000000000000010')
switch1 = self.addSwitch('s1', dpid='0000000000000020')
```

```
switch2 = self.addSwitch('s2', dpid='0000000000000030')

# 100 Mbps
self.addLink(host0, switch0, bw=100) # s0.port_1: host0
self.addLink(host1, switch0, bw=100) # s0.port_2: host1

self.addLink(host2, switch1, bw=100) # s1.port_1: host0
self.addLink(host3, switch1, bw=100) # s1.port_1: host0

# 1 Mbps bandwidth
self.addLink(switch0, switch2, bw=1) # s0.port_3: s2.port_1
self.addLink(switch1, switch2, bw=1) # s1.port_3: s2.port_2
```

The above code snippet from the script that generates the Mininet topology showcases how switches and hosts are connected along with configuring the bandwidth between them.

3.1.3 Open vSwitch & OVSDB

Open vSwitch (OVS)[Pfaff et al., 2015] is a production quality, multilayer virtual switch. Initially designed for virtual networking environments, it is widely utilised and installable to hardware devices. It supports the OpenFlow protocol and is supported by both the Mininet API and Ryu.

Open vSwitch Database (OVSDB)[Pfaff and Davie, 2013b] management protocol is a configuration protocol that is designed to manage and configure OVS implementations.

3.1.4 Ryu

Ryu is the SDN controller for the system. It takes requests about switch configurations and forwards the correct configuration changes down to the switches. Three Ryu apps are required to implement bandwidth reservation: rest_conf_switch, a slightly customised version of simple_switch_13 and rest_qos.

The rest_conf_switch app contains an API for setting configuration values in switches. This is used to set the 'ovsdb_addr' configuration value on the switches so they know at what address to access OVSDB. The simple_switch_13 app is an app that provides capabilities to learn the MAC addresses of nearby hosts. It has been modified for this implementation to report any OpenFlow errors for debugging purposes as Ryu does not report them by default. The rest_qos app accepts requests about QoS-related queue and rule configurations and forwards them to the switches. The functionality provided by these API's has been covered in section 3.1.1.

3.2 System Functionality

3.2.1 System Architecture

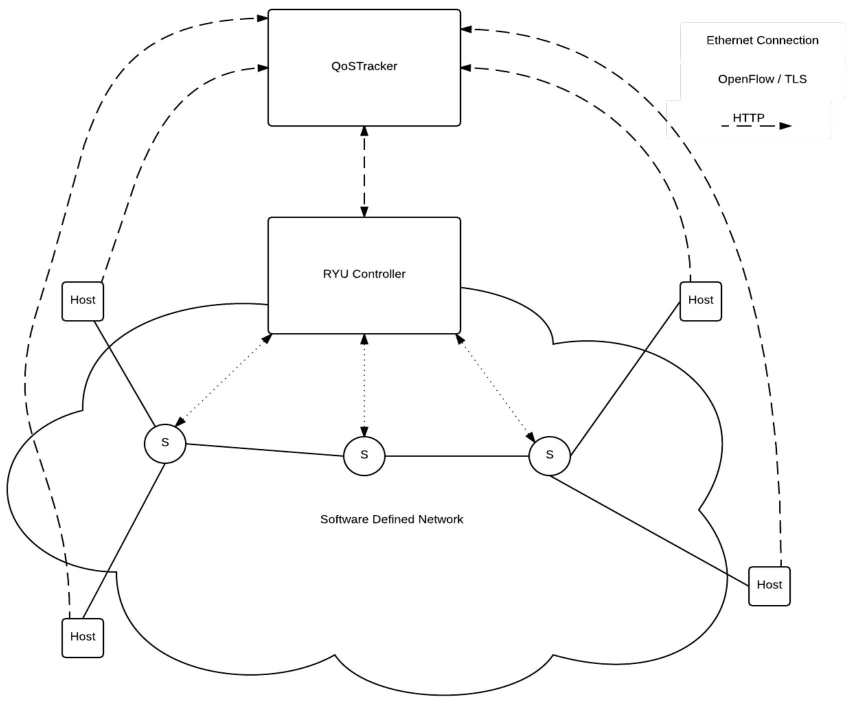


FIGURE 3.4: System Architecture.

There are three main components to the system: the QoSTracker (the bandwidth reservation system), the selected Ryu apps and the network topology containing the hosts and switches. In addition to these, a running instance of OVSDB on the same host as the topology is required to be running. How the components interact with each other is illustrated in Figure 3.4.

Although not restricted to a single host, the entire system was hosted on a single machine to make development easier. Throughout the course of development, a

Macintosh Macbook Pro 2012 model was used. VirtualBox, an open source hypervisor, hosts a virtual machine (VM) running Linux that contains an installed version of Mininet. Both the Ryu apps and the QoSTracker elements were hosted on the laptops primary operating system and accessed the topological elements within the virtual network by means of port-forwarding.

3.2.2 Bandwidth Reservation Mechanisms

The QoS features supported by Ryu meant that there were a number of approaches available to implement bandwidth reservation. At this point it is worth explaining how switch queues work. There are two types of switching queues, ingress queues for incoming traffic and egress queues for outgoing traffic. For this project, egress queues are the primary focus. Each queueing mechanism is composed of a collection of smaller queues. These queues are ordered and emptied based on a priority they are assigned. As unreserved traffic is passed through a network, it is placed in the lowest priority queue at each egress port. If none of the higher priority queues are populated at a given time, the switch forwards any packets in that queue. If a burst of reserved traffic was to pass through the switch, it would be placed in a higher priority queue and the switch would forward those packets before the lower priority queues. How packets are assigned to queues is done via packet marking. This marking continues with the packet throughout the network so it is continually guaranteed a higher priority queue. In the event that a reserved traffic flow exceeds it's reserved bandwidth level, any traffic above the guaranteed value is placed in lower queues and is processed throughout the network at a rate known as 'best-effort'. All unreserved traffic is processed at a best-effort rate.

The first option considered was a per-flow reservation approach using Multiprotocol Label Switching (MPLS)[Rosen, Viswanathan, and Callon, 2001] to identify reserved traffic. MPLS is a protocol for shaping and speeding up traffic flows. It works by marking packets via assigning them an MPLS label at the ingress routers to networks. It is continually identified by this label as it moves through the network without networking devices having to examine the IP header information. The general process for the MPLS is:

1. Determine the path the reserved traffic will take through the network.
2. On the first switch of the path (from source to destination), install a QoS rule that checks the packet's source and destination IP address for the expected

values. The action upon a packet matching the criteria is to assign an MPLS label to the packet.

3. For all nodes along the flow, define queue settings for the switch that defines two types of traffic:

   - Best-effort traffic: Any unreserved traffic is passed through the network at the 'best-effort' rate.
   - Priority/Reserved traffic: Any traffic that is reserved.

   For best-effort traffic, a queue is configured that gives it a maximum rate value that corresponds to the maximum available bandwidth for the link connected to the port. For reserved traffic, a queue is configured that defines a minimum rate of traffic that is guaranteed. This corresponds to the amount of bandwidth a client has requested. Traffic that arrives within the requested rate is given a priority of '1' (or another number if more than one reservation is ongoing across the network).

4. For all nodes along the path bar the ingress and egress, install a rule that checks for the presence of the specified MPLS label, and assigns it to the higher priority queue.

5. For the egress node of the traffic flow, install a flow that looks for the specified MPLS label and removes it from the packet.

Development on this approach was initially worked on but was abandoned due to the realisation that although Ryu provides functions for removing the MPLS labels on packets, it hadn't actually been implemented yet. This particular solution would not scale particularly well in larger networks as the amount of rule and queue setting changes per-flow would become difficult to manage as the number of reservations increases. Instead of marking packets using MPLS, the decision to use Differentiated Services (DiffServ)[Grossman, 2013] to categorise and prioritise traffic was made. DiffServ is a networking architecture that provides mechanisms for classifying and managing network traffic. It used a differentiated services code point (DSCP) value in the DS field of IP packets to classify packet priority. Queues are defined on each switch along the reserved traffic's path with queues defined for each type of traffic. Reserved traffic is assigned a minimum-rate queue corresponding to the amount of bandwidth. Upon the entry to the network, reserved packets are assigned an appropriate DSCP value with unreserved traffic assigned a DSCP value of '0'. At each switch, queues are populated based on their DSCP value.

Chapter 4

Results

4.1 Research Questions Results

4.1.1 Topology Discovery

Ryu provides a topology API that returns information on switches, ports, and links.

For switches, it provides their datapath ID to uniquely identify each switch in the network and a list of the ports connected. Each port is uniquely identifiable by it's hardware address which is returned with the relevent port information by Ryu. A HTTP GET request to the /v1.0/topology/switches/<dpid> endpoint returns:

```
[
    {   "ports":[
                 {
                    "hw_addr":"9e:24:2a:05:a2:9c",
                    "name":"s0-eth1",
                    "port_no":"00000001",
                    "dpid":"0000000000000010"
                 },
                 {
                    "hw_addr":"3a:a7:8a:a5:83:b5",
                    "name":"s0-eth2",
                    "port_no":"00000002",
                    "dpid":"0000000000000010"
                 }
              ],
              "dpid":"0000000000000010"
    }
]
```

For links, Ryu returns information about the source and destination port and switch which, when combined, uniquely identify it from the other links. A HTTP GET request to the v1.0/topology/links endpoint returns:

```
   {
      "src":{
         "hw_addr":"82:b4:8e:79:2a:5a",
         "name":"s0-eth3",
         "port_no":"00000003",
         "dpid":"0000000000000010"
      },
      "dst":{
         "hw_addr":"0a:e1:fe:8a:12:69",
         "name":"s2-eth1",
         "port_no":"00000001",
         "dpid":"0000000000000030"
      }
   }
```

The information returned by the API is sufficient to accurately determine the topology of the network. It provides enough information to uniquely identify each individual topology component along with adequate data to derive the connections between components.

One aspect of Ryu's topology API that was found to be lacking was link bandwidth discovery. A bandwidth reservation system cannot operate correctly without knowledge of the links connecting the networking devices. This required having to manually enter the bandwidths for each link in the topology for every topology tested with. Although it is not a major stumbling block, it does hamper the systems ability to simply plug into different topologies without making significant configuration changes.

4.1.2 Bandwidth Reservation Mechanisms

Ryu supports a variety of bandwidth reservation mechanisms meaning that network operators are not limited in how they implement BoD in their networks.

4.1.3 Queues & Rules Interface

Ryu provides a comprehensive API to interface with the queueing infrastructure of switches. Utilising Ryu's rest_qos module, network operators can use combinations of rules and queue configurations to build a functional capacity reservation system.

The types of requests that Ryu supports have been outlined in section 3.1.1.

4.1.4 Meter Tables Interface

Ryu supports the use of meter tables. Meters were introduced in OpenFlow version 1.3 and are based around two values: Committed Information Rate (CIR) and Peak Information Rate (PIR). CIR is the rate guaranteed for a client across a network, meaning their allotted bandwidth will never drop below this value. PIR is the maximum amount of bandwidth a client can have at a given time. This value is greater than the CIR and is based around the idea that 100% of the bandwidth is rarely in use 100% of the time providing a means for clients to use up more bandwidth.

Meter tables make use of the underlying queueing infrastructure already in place in switches. For each switch, two flow tables are created. The first table compares the current rate of the incoming traffic. The first table drops any packets that fall above the clients PIR. All packets that fall within the PIR are forwarded to the second table. Here, any packets that fall within the clients CIR are placed in a priority queue. The remaining packets, those that are at a rate between the clients PIR and CIR, are placed in the best-effort queue. Packets that fall within the CIR are marked with a priority that carries and is recognised throughout the network. Any packets that fall between CIR and PIR are marked as priority 0. This approach has the advantage that assuming the correct calculations were done on allocated reservations and their associated PIR and CIR, traffic shaping only occurs at the ingress to the network.

Ryu's QoS REST API provides an endpoint for manipulating meter entries. An HTTP POST request to the /qos/meter/<switch_id> accepts parameters in the following format:

```
{
   "meter_id":"<int>",
   "bands":[
      {
         "action":"<DROP or DSCP_REMARK>",
         "flag":"<KBPS or PKTPS or BURST or STATS",
         "burst_size":"<int>",
         "rate":"<int>",
         "prec_level":"<int>"
      }
   ]
}
```

4.2 Implementation Results

Unfortunately the bandwidth reservation does not work as intended. Due to a number of reasons, the full scope of functionality was not implemented. The following is a walkthrough of what functionality does work with examples at each stage.

4.3 Environment Setup

For this example the topology used to test is outlined in figure 4.1. The links between hosts and switches are 100 Mbps and the links between switches are 1 Mpbs to create a bottleneck.

To begin, create the virtual topology by running:

```
    sudo python topologies/2_switches_4_hosts.py
```

Next, set up the required Ryu APIs by running:

```
ryu-manager ryu.app.rest_qos ryu.app.new_qos.qos_simple_switch_13
    ryu.app.rest_conf_switch --observe-links
```

Once Ryu has setup the APIs, it should recognise the switches which is indicated by the following output from Ryu:

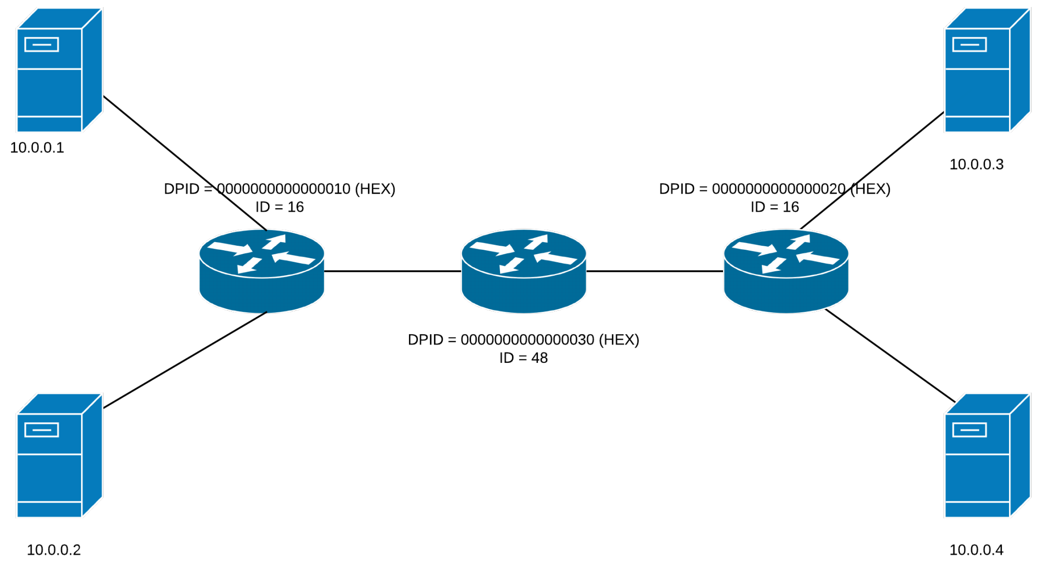
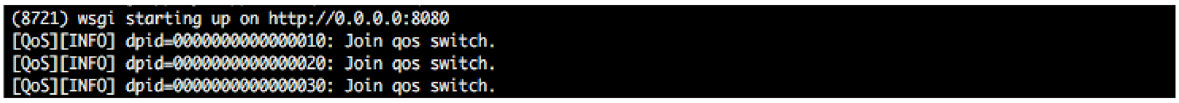

10.0.0.1
10.0.0.3
DPID = 0000000000000010 (HEX)
ID = 16
DPID = 0000000000000020 (HEX)
ID = 16
DPID = 0000000000000030 (HEX)
ID = 48
10.0.0.2
10.0.0.4

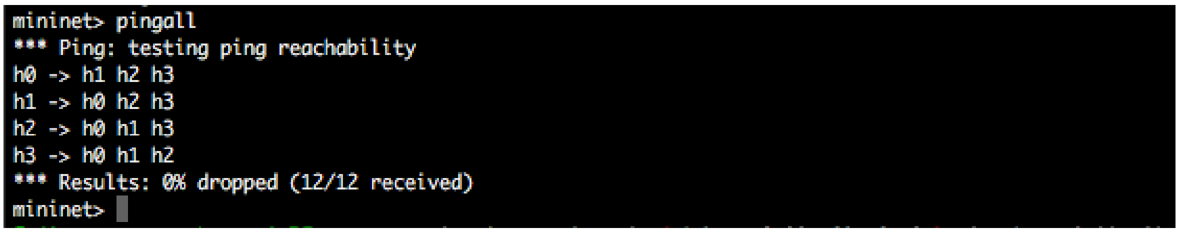
FIGURE 4.1: Network topology diagram.



```
(8721) wsgi starting up on http://0.0.0.0:8080
[QoS][INFO] dpid=0000000000000010: Join qos switch.
[QoS][INFO] dpid=0000000000000020: Join qos switch.
[QoS][INFO] dpid=0000000000000030: Join qos switch.
```

FIGURE 4.2: Ryu recognising the virtual switches.

Once the controller is running and the switches are connected properly, verify that Ryu has correctly populated the flow tables by running the pingall command in mininet:



```
mininet> pingall
*** Ping: testing ping reachability
h0 -> h1 h2 h3
h1 -> h0 h2 h3
h2 -> h0 h1 h3
h3 -> h0 h1 h2
*** Results: 0% dropped (12/12 received)
mininet> 
```

FIGURE 4.3: Mininet output showing the hosts pinging each other.

Now that the switches are able to communicate with each other, the bandwidth reservation system can be started by running the following command:

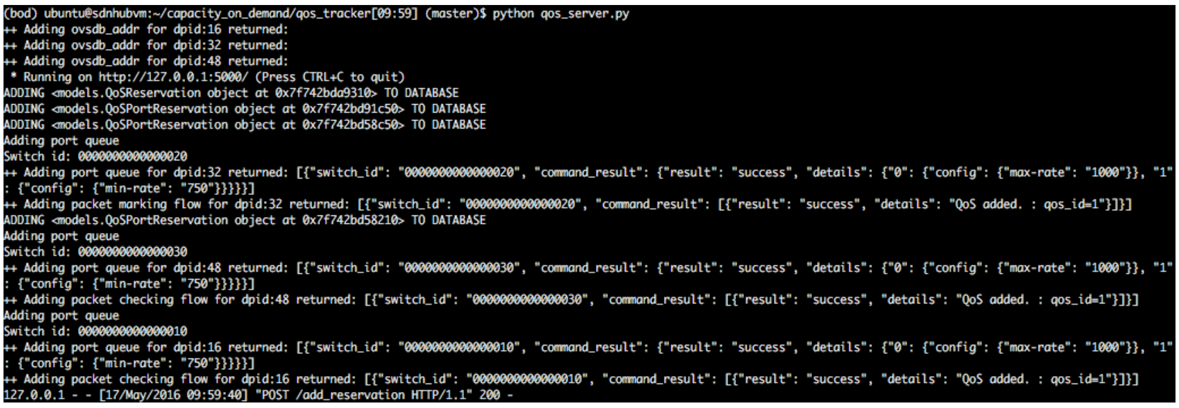
```
sudo python qos_tracker/qos_server.py
```

The bandwidth reservation system begins by forwarding the address of the running OVSDB instanve to configure the switches for QoS rule and queue setting changes.

Now, bandwidth reservations can be requested. The following is an example call to the bandwidth reservation system:

```
curl -i -H "Accept: application/json" -H "Content-Type:
    application/json" -X POST -d '{"src": "10.0.0.4",
    "dst": "10.0.0.1", "bw": 750}'
    http://localhost:5000/add_reservation
```

Here, a client is requesting 750 bits of reserved bandwidth from the host with the IP address 10.0.0.4 to the host with IP address 10.0.0.1. On the bandwidth reservation terminal, the output is displayed upon receiving the request in figure 4.4.



```
(bod) ubuntu@sdnhubvm:~/capacity_on_demand/qos_tracker[09:59] (master)$ python qos_server.py
++ Adding ovsdb_addr for dpid:16 returned:
++ Adding ovsdb_addr for dpid:32 returned:
++ Adding ovsdb_addr for dpid:48 returned:
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
ADDING <models.QoSReservation object at 0x7f742bda9310> TO DATABASE
ADDING <models.QoSPortReservation object at 0x7f742bd91c50> TO DATABASE
ADDING <models.QoSPortReservation object at 0x7f742bd58c50> TO DATABASE
Adding port queue
Switch id: 0000000000000020
++ Adding port queue for dpid:32 returned: [{"switch_id": "0000000000000020", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1"
: {"config": {"min-rate": "750"}}}}}]
++ Adding packet marking flow for dpid:32 returned: [{"switch_id": "0000000000000020", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}]
ADDING <models.QoSPortReservation object at 0x7f742bd58210> TO DATABASE
Adding port queue
Switch id: 0000000000000030
++ Adding port queue for dpid:48 returned: [{"switch_id": "0000000000000030", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1"
: {"config": {"min-rate": "750"}}}}}]
++ Adding packet checking flow for dpid:48 returned: [{"switch_id": "0000000000000030", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}]
Adding port queue
Switch id: 0000000000000010
++ Adding port queue for dpid:16 returned: [{"switch_id": "0000000000000010", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1"
: {"config": {"min-rate": "750"}}}}}]
++ Adding packet checking flow for dpid:16 returned: [{"switch_id": "0000000000000010", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}]
127.0.0.1 - - [17/May/2016 09:59:40] "POST /add_reservation HTTP/1.1" 200 -
```

FIGURE 4.4: Output from the bandwidth reservation system upon receiving a bandwidth reservation request.

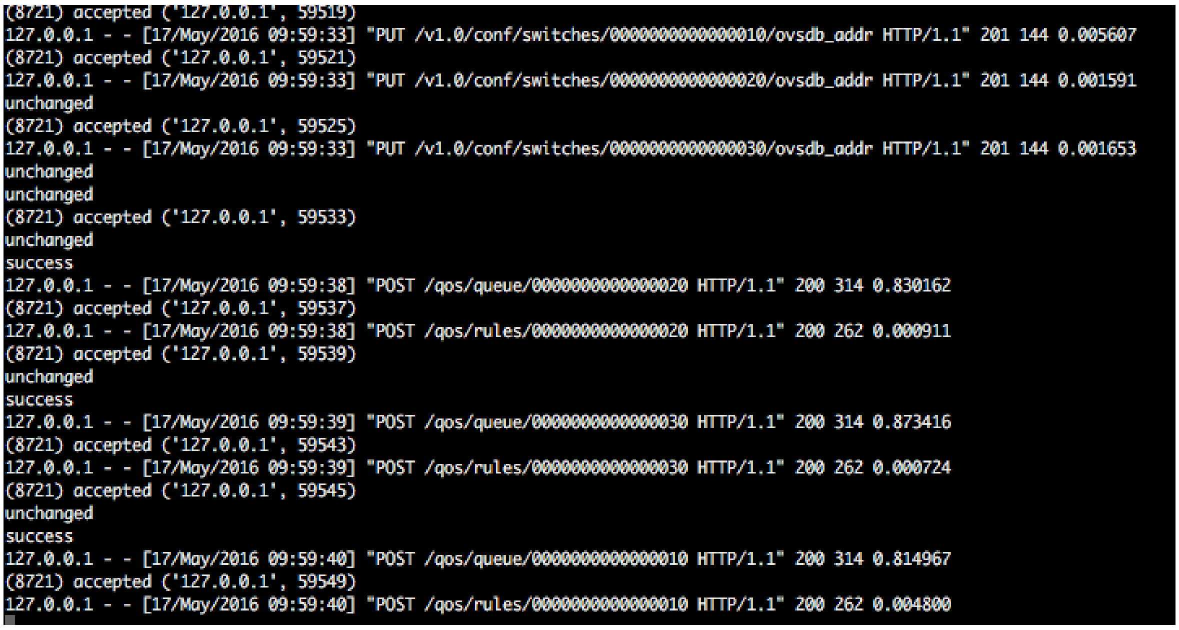
The messages beginning with '++' indicate that what is being printed is the response from Ryu's APIs. The messages beginning with 'ADDING' indicate an entry to the database being added. Here, we can see that it adds a reservation entry and three port_reservation entries, one per switch along the path.

In figure 4.5, the requests to the /qos/queue and /qos/rule endpoints sent by the bandwidth reservation system can be seen. For each switch, a rule and a queue setting are posted as outlined in sections 3.2.2 and 3.1.1.

The full terminal output from the three components will be included in Appendix A.

4.3.1 Component Validation

This section will be a demonstration of the components that do work.

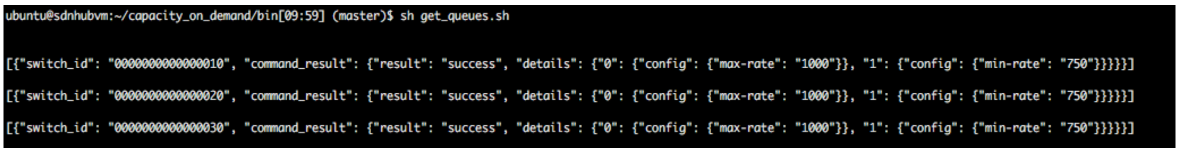


```
(8721) accepted ('127.0.0.1', 59519)
127.0.0.1 - - [17/May/2016 09:59:33] "PUT /v1.0/conf/switches/0000000000000010/ovsdb_addr HTTP/1.1" 201 144 0.005607
(8721) accepted ('127.0.0.1', 59521)
127.0.0.1 - - [17/May/2016 09:59:33] "PUT /v1.0/conf/switches/0000000000000020/ovsdb_addr HTTP/1.1" 201 144 0.001591
unchanged
(8721) accepted ('127.0.0.1', 59525)
127.0.0.1 - - [17/May/2016 09:59:33] "PUT /v1.0/conf/switches/0000000000000030/ovsdb_addr HTTP/1.1" 201 144 0.001653
unchanged
unchanged
(8721) accepted ('127.0.0.1', 59533)
unchanged
success
127.0.0.1 - - [17/May/2016 09:59:38] "POST /qos/queue/0000000000000020 HTTP/1.1" 200 314 0.830162
(8721) accepted ('127.0.0.1', 59537)
127.0.0.1 - - [17/May/2016 09:59:38] "POST /qos/rules/0000000000000020 HTTP/1.1" 200 262 0.000911
(8721) accepted ('127.0.0.1', 59539)
unchanged
success
127.0.0.1 - - [17/May/2016 09:59:39] "POST /qos/queue/0000000000000030 HTTP/1.1" 200 314 0.873416
(8721) accepted ('127.0.0.1', 59543)
127.0.0.1 - - [17/May/2016 09:59:39] "POST /qos/rules/0000000000000030 HTTP/1.1" 200 262 0.000724
(8721) accepted ('127.0.0.1', 59545)
unchanged
success
127.0.0.1 - - [17/May/2016 09:59:40] "POST /qos/queue/0000000000000010 HTTP/1.1" 200 314 0.814967
(8721) accepted ('127.0.0.1', 59549)
127.0.0.1 - - [17/May/2016 09:59:40] "POST /qos/rules/0000000000000010 HTTP/1.1" 200 262 0.004800
```

FIGURE 4.5: Output from the terminal running Ryu's APIs.

Firstly, to validate that the switch queue settings were correctly added, the following shell script can be ran that returns the queue settings for each switch:

```
echo "\n"
curl -X GET http://localhost:8080/qos/queue/0000000000000010
echo "\n"
curl -X GET http://localhost:8080/qos/queue/0000000000000020
echo "\n"
curl -X GET http://localhost:8080/qos/queue/0000000000000030
echo "\n"
```



```
ubuntu@sdnhubvm:~/capacity_on_demand/bin[09:59] (master)$ sh get_queues.sh

[{"switch_id": "0000000000000010", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1": {"config": {"min-rate": "750"}}}}}]
[{"switch_id": "0000000000000020", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1": {"config": {"min-rate": "750"}}}}}]
[{"switch_id": "0000000000000030", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1": {"config": {"min-rate": "750"}}}}}]
```

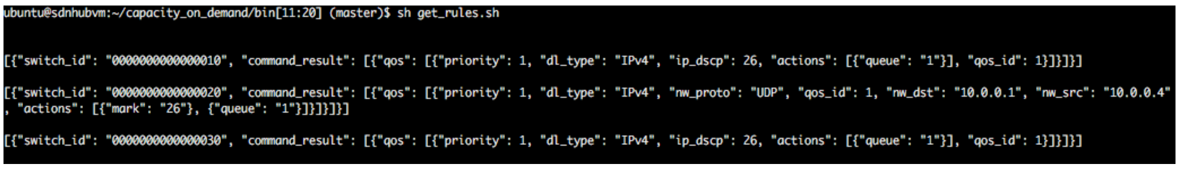
FIGURE 4.6: Output from executing get_queues.sh.

This returns the output seen in figure 4.6. In this screenshot we can see the max rate applied to best-effort traffic. Note how there is no minimum value for best-effort as they have no level of guaranteed bandwidth. For priority traffic, a queue of min rate 750 bps is defined which means that no matter how much traffic is passing through that switch at any given time, that 750 bps is guaranteed.

Secondly, to validate that Ryu is correctly adding rules, the following shell script can be executed to return the rules associated with each switch:

```
echo "\n"
curl -X GET http://localhost:8080/qos/rules/0000000000000010
echo "\n"
curl -X GET http://localhost:8080/qos/rules/0000000000000020
echo "\n"
curl -X GET http://localhost:8080/qos/rules/0000000000000030
echo "\n"
```

The output returned from this shell script is shown in figure 4.7. Note how the switch with the datapath ID of 32 (0000000000000020) has different rules than the other two switches. This is so due to the switch with datapapth ID of 32 is the ingress node of the flow i.e. 10.0.0.4 (the source host) is connected directly to this switch. As a result, this switch is tasked with marking the packets with the appropriate DSCP value. Note how the other two switches only check for the DSCP value and assign it to a queue.



```
ubuntu@sdnhubvm:~/capacity_on_demand/bin[11:20] (master)$ sh get_rules.sh


[{"switch_id": "0000000000000010", "command_result": [{"qos": [{"priority": 1, "dl_type": "IPv4", "ip_dscp": 26, "actions": [{"queue": "1"}], "qos_id": 1}]}]}]

[{"switch_id": "0000000000000020", "command_result": [{"qos": [{"priority": 1, "dl_type": "IPv4", "nw_proto": "UDP", "qos_id": 1, "nw_dst": "10.0.0.1", "nw_src": "10.0.0.4"
, "actions": [{"mark": "26"}, {"queue": "1"}]}]}]}]

[{"switch_id": "0000000000000030", "command_result": [{"qos": [{"priority": 1, "dl_type": "IPv4", "ip_dscp": 26, "actions": [{"queue": "1"}], "qos_id": 1}]}]}]
```

FIGURE 4.7: Output from executing get_rules.sh.

To test that the packet marking rule is correctly identifying and marking the packets, start a wireshark[Orebaugh et al., 2007] (a tool for monitoring and debugging network traffic) session by running the following command:

```
sudo wireshark &
```

Next, in the mininet terminal, start up individual terminals for the hosts at 10.0.0.1 (h0) and 10.0.0.4 (h3) by running:

```
xterm h0 h3
```

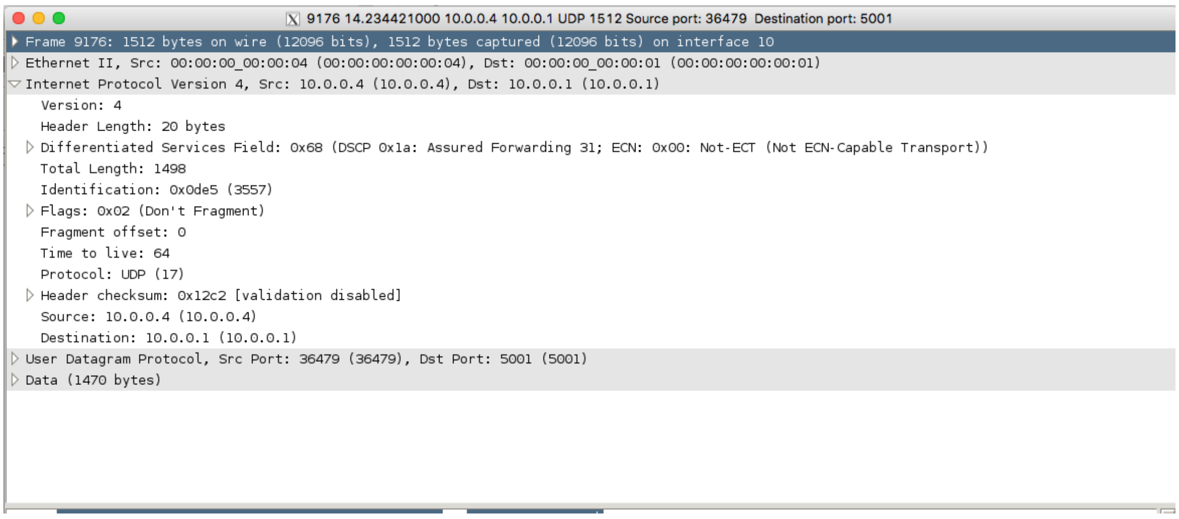
On 10.0.0.1 (the destination IP of the reservation) run the following command that sets up an iperf (a tool for simulating network traffic) session:

```
iperf -s -u -i 1
```

This creates a running server (-s flag) that is ready to accept UDP traffic (determined by the -u flag) and reports any incoming traffic in intervals of 1 second (-c 1).

On the 10.0.0.4 (recall that this is the source IP of the reservation), start up a client iperf session by running the following command:

```
iperf -c 10.0.0.1 -i 1 -u -b 1M
```



```
9176 14.234421000 10.0.0.4 10.0.0.1 UDP 1512 Source port: 36479  Destination port: 5001
▸ Frame 9176: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface 10
▷ Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
▽ Internet Protocol Version 4, Src: 10.0.0.4 (10.0.0.4), Dst: 10.0.0.1 (10.0.0.1)
    Version: 4
    Header Length: 20 bytes
  ▷ Differentiated Services Field: 0x68 (DSCP 0x1a: Assured Forwarding 31; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 1498
    Identification: 0x0de5 (3557)
  ▷ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: UDP (17)
  ▷ Header checksum: 0x12c2 [validation disabled]
    Source: 10.0.0.4 (10.0.0.4)
    Destination: 10.0.0.1 (10.0.0.1)
▷ User Datagram Protocol, Src Port: 36479 (36479), Dst Port: 5001 (5001)
▷ Data (1470 bytes)
```

FIGURE 4.8: Output from wireshark capturing the DSCP value of the marked traffic.

This command tells 10.0.0.4 to stream UDP (-u flag) at a rate of 1 Mbps (-b 1M) to 10.0.0.1. Capturing this traffic in wireshark shows that the correct DSCP value is being added to the packet. The default for best-effort traffic is 0 but as can be seen in figure figure:Wireshark, this contains an actual DSCP value and so has been identified and marked correctly.

This was as much of the project that works. The actual queueing of prioritised traffic does not work. In the even that two traffic streams are traversing the bottleneck at the same time, the switches assign equal bandwidth to each stream instead of 750 bps to the prioritised traffic and 250 to the best-effort traffic.

4.4 Additional Results

Due to the shortcomings of the implementation, the following is an account of how to implement capacity reservation in a single switch using Ryu's APIs to prove that it is indeed possible to use Ryu to carry out capacity reservation. It follows an example detailed in chapter 11 of the Ryubook[*Ryu SDN Framework*.

First, the topology is built. It consists of a single switch with two hosts attached. The following mininet comand will generate it:

```
sudo mn --mac --switch ovsk --controller remote -x
```

Next, modify Ryu's simple_switch_13 app so that flow entries are added to table with ID 1 instead of 0 with the following command:

```
sed '/OFPFlowMod(/,/)/s/)/, table_id=1)/'
    ryu/ryu/app/simple_switch_13.py >
    ryu/ryu/app/qos_simple_switch_13.py
```

Then, install the new module by running the following command within the ryu directory:

```
python ./setup.py install
```

Then get the appropriate Ryu apps running:

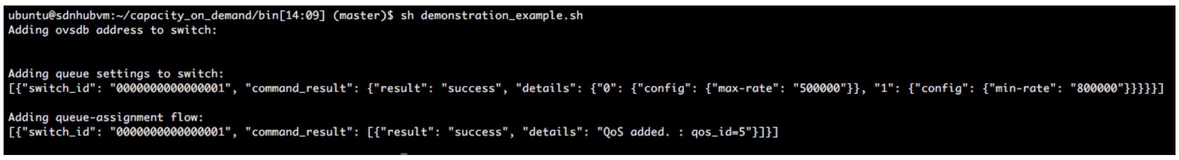
```
ryu-manager ryu.app.rest_qos ryu.app.qos_simple_switch_13
    ryu.app.rest_conf_switch
```

Once Ryu is running and a successful connection has been made with the virtual switch, the following bash script will add a bandwidth reservation:

```
sudo ovs-vsctl set Bridge s1 protocols=OpenFlow13
sudo ovs-vsctl set-manager ptcp:6632

echo "Adding ovsdb address to switch:"
# Add ovsdb address to switch
curl -X PUT -d '"tcp:127.0.0.1:6632"'
    http://localhost:8080/v1.0/conf/switches/0000000000000001/
        ...ovsdb_addr
echo "\n"

echo "Adding queue settings to switch:"
# Add queue settings to switch
curl -X POST -d '{"port_name":"s1-eth1","type":"linux-htb",
    "max_rate":"1000000","queues":[{"max_rate":"500000"},
    {"min_rate":"800000"}]}'
    http://localhost:8080/qos/queue/0000000000000001
echo "\n"
echo "Adding queue-assignment flow:"
# Install flow entry to assign queue id
curl -X POST -d '{"match":{"nw_dst":"10.0.0.1","nw_proto":"UDP",
    "tp_dst":"5002"},"actions":{"queue":1}}'
    http://localhost:8080/qos/rules/0000000000000001
echo "\n"
```



```
ubuntu@sdnhubvm:~/capacity_on_demand/bin[14:09] (master)$ sh demonstration_example.sh
Adding ovsdb address to switch:


Adding queue settings to switch:
[{"switch_id": "0000000000000001", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "500000"}}, "1": {"config": {"min-rate": "800000"}}}}}]

Adding queue-assignment flow:
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "QoS added. : qos_id=5"}]}]
```

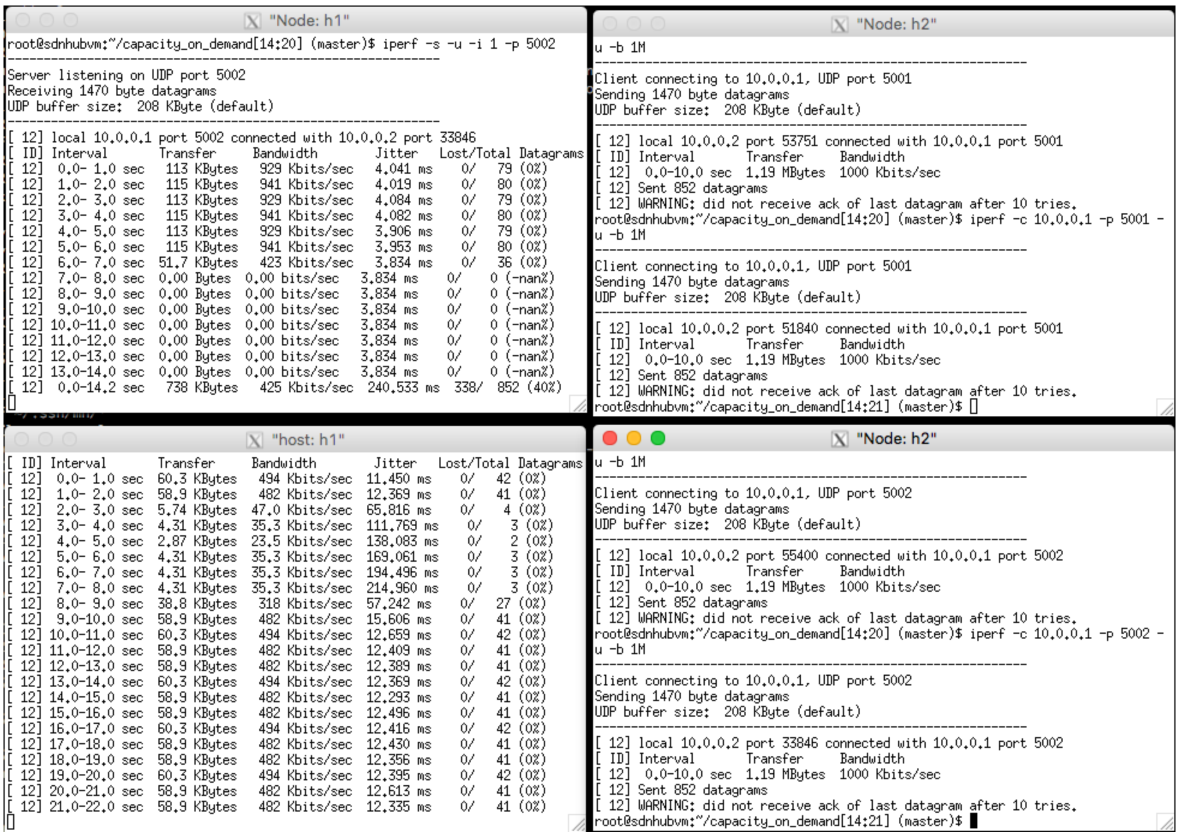
FIGURE 4.9: Output from the shell script that adds QoS rules to the switch.

This script begins by setting the version of OpenFlow to be used by the switch. Then it adds the address of the running OVSDB process to the switches configuration. Then queue settings are defined and a packet checking rule is pushed. The output from the script can be seen in figure 4.9.

Note how rather than match on a source and destination IP address, this approach matches on a destination IP address and a destination port, 5002 in this example.

To verify that the queue and rule configuration changes were implemented correctly, open up two terminals per host. On the first h1 terminal, run a server instance of iperf on port 5001 and on the second repeat the command but on port 5002. On the first h2 terminal send UDP traffic to the iperf server running on port 5001 on h1 while at the same time sending UDP traffic from the other h2 terminal to the other running iperf server on h1 on port 5002.

Figure 4.10 shows the output from these commands. The requested reserved bandwidth was 800 Kbps. Although there are slight timing issues with the terminal output, the top-left terminal shows bandwidth at 900 Kbps with the lower, non-reserved stream of traffic only getting approximately 60 Kpbs. The reason behind the traffic not being split 800/200 Kbps is that the switch divides up the remaining bandwidth evenly between the two streams of traffic so that the reserved flow gets an additional 70-100 Kbps as does the unreserved flow.



```
root@sdnhubvm:~/capacity_on_demand[14:20] (master)$ iperf -s -u -i 1 -p 5002
------------------------------------------------------------
Server listening on UDP port 5002
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 12] local 10.0.0.1 port 5002 connected with 10.0.0.2 port 33846
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[ 12]  0.0- 1.0 sec   113 KBytes   929 Kbits/sec   4.041 ms    0/   79 (0%)
[ 12]  1.0- 2.0 sec   115 KBytes   941 Kbits/sec   4.019 ms    0/   80 (0%)
[ 12]  2.0- 3.0 sec   113 KBytes   929 Kbits/sec   4.084 ms    0/   79 (0%)
[ 12]  3.0- 4.0 sec   115 KBytes   941 Kbits/sec   4.082 ms    0/   80 (0%)
[ 12]  4.0- 5.0 sec   113 KBytes   929 Kbits/sec   3.906 ms    0/   79 (0%)
[ 12]  5.0- 6.0 sec   115 KBytes   941 Kbits/sec   3.953 ms    0/   80 (0%)
[ 12]  6.0- 7.0 sec  51.7 KBytes   423 Kbits/sec   3.834 ms    0/   36 (0%)
[ 12]  7.0- 8.0 sec  0.00 Bytes  0.00 bits/sec   3.834 ms    0/    0 (-nan%)
[ 12]  8.0- 9.0 sec  0.00 Bytes  0.00 bits/sec   3.834 ms    0/    0 (-nan%)
[ 12]  9.0-10.0 sec  0.00 Bytes  0.00 bits/sec   3.834 ms    0/    0 (-nan%)
[ 12] 10.0-11.0 sec  0.00 Bytes  0.00 bits/sec   3.834 ms    0/    0 (-nan%)
[ 12] 11.0-12.0 sec  0.00 Bytes  0.00 bits/sec   3.834 ms    0/    0 (-nan%)
[ 12] 12.0-13.0 sec  0.00 Bytes  0.00 bits/sec   3.834 ms    0/    0 (-nan%)
[ 12] 13.0-14.0 sec  0.00 Bytes  0.00 bits/sec   3.834 ms    0/    0 (-nan%)
[ 12]  0.0-14.2 sec   738 KBytes   425 Kbits/sec  240.533 ms  338/  852 (40%)
```

```
u -b 1M
------------------------------------------------------------
Client connecting to 10.0.0.1, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 12] local 10.0.0.2 port 53751 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 12]  0.0-10.0 sec  1.19 MBytes  1000 Kbits/sec
[ 12] Sent 852 datagrams
[ 12] WARNING: did not receive ack of last datagram after 10 tries.
root@sdnhubvm:~/capacity_on_demand[14:20] (master)$ iperf -c 10.0.0.1 -p 5001 -
u -b 1M
------------------------------------------------------------
Client connecting to 10.0.0.1, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 12] local 10.0.0.2 port 51840 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 12]  0.0-10.0 sec  1.19 MBytes  1000 Kbits/sec
[ 12] Sent 852 datagrams
[ 12] WARNING: did not receive ack of last datagram after 10 tries.
root@sdnhubvm:~/capacity_on_demand[14:21] (master)$
```

```
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[ 12]  0.0- 1.0 sec  60.3 KBytes   494 Kbits/sec  11.450 ms    0/   42 (0%)
[ 12]  1.0- 2.0 sec  58.9 KBytes   482 Kbits/sec  12.369 ms    0/   41 (0%)
[ 12]  2.0- 3.0 sec  5.74 KBytes  47.0 Kbits/sec  65.816 ms    0/    4 (0%)
[ 12]  3.0- 4.0 sec  4.31 KBytes  35.3 Kbits/sec  111.769 ms    0/    3 (0%)
[ 12]  4.0- 5.0 sec  2.87 KBytes  23.5 Kbits/sec  138.083 ms    0/    2 (0%)
[ 12]  5.0- 6.0 sec  4.31 KBytes  35.3 Kbits/sec  169.061 ms    0/    3 (0%)
[ 12]  6.0- 7.0 sec  4.31 KBytes  35.3 Kbits/sec  194.496 ms    0/    3 (0%)
[ 12]  7.0- 8.0 sec  4.31 KBytes  35.3 Kbits/sec  214.960 ms    0/    3 (0%)
[ 12]  8.0- 9.0 sec  38.8 KBytes   318 Kbits/sec  57.242 ms    0/   27 (0%)
[ 12]  9.0-10.0 sec  58.9 KBytes   482 Kbits/sec  15.606 ms    0/   41 (0%)
[ 12] 10.0-11.0 sec  60.3 KBytes   494 Kbits/sec  12.659 ms    0/   42 (0%)
[ 12] 11.0-12.0 sec  58.9 KBytes   482 Kbits/sec  12.409 ms    0/   41 (0%)
[ 12] 12.0-13.0 sec  58.9 KBytes   482 Kbits/sec  12.389 ms    0/   41 (0%)
[ 12] 13.0-14.0 sec  60.3 KBytes   494 Kbits/sec  12.369 ms    0/   42 (0%)
[ 12] 14.0-15.0 sec  58.9 KBytes   482 Kbits/sec  12.293 ms    0/   41 (0%)
[ 12] 15.0-16.0 sec  58.9 KBytes   482 Kbits/sec  12.496 ms    0/   41 (0%)
[ 12] 16.0-17.0 sec  60.3 KBytes   494 Kbits/sec  12.416 ms    0/   42 (0%)
[ 12] 17.0-18.0 sec  58.9 KBytes   482 Kbits/sec  12.430 ms    0/   41 (0%)
[ 12] 18.0-19.0 sec  58.9 KBytes   482 Kbits/sec  12.356 ms    0/   41 (0%)
[ 12] 19.0-20.0 sec  60.3 KBytes   494 Kbits/sec  12.395 ms    0/   42 (0%)
[ 12] 20.0-21.0 sec  58.9 KBytes   482 Kbits/sec  12.613 ms    0/   41 (0%)
[ 12] 21.0-22.0 sec  58.9 KBytes   482 Kbits/sec  12.335 ms    0/   41 (0%)
```

```
u -b 1M
------------------------------------------------------------
Client connecting to 10.0.0.1, UDP port 5002
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 12] local 10.0.0.2 port 55400 connected with 10.0.0.1 port 5002
[ ID] Interval       Transfer     Bandwidth
[ 12]  0.0-10.0 sec  1.19 MBytes  1000 Kbits/sec
[ 12] Sent 852 datagrams
[ 12] WARNING: did not receive ack of last datagram after 10 tries.
root@sdnhubvm:~/capacity_on_demand[14:20] (master)$ iperf -c 10.0.0.1 -p 5002 -
u -b 1M
------------------------------------------------------------
Client connecting to 10.0.0.1, UDP port 5002
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 12] local 10.0.0.2 port 33846 connected with 10.0.0.1 port 5002
[ ID] Interval       Transfer     Bandwidth
[ 12]  0.0-10.0 sec  1.19 MBytes  1000 Kbits/sec
[ 12] Sent 852 datagrams
[ 12] WARNING: did not receive ack of last datagram after 10 tries.
root@sdnhubvm:~/capacity_on_demand[14:21] (master)$
```

FIGURE 4.10: Output from the four terminals that show traffic being shaped.

Chapter 5

Conclusion

5.1 Evaluation of Implementation

Although certain functionality was provided by the bandwidth reservation system, the key component of actually reserving the bandwidth was never fully implemented. This was for a number of reasons but primarily due to a poor implementation strategy.

From the beginning, the system was being built and designed with complex and large networks in mind. Rather than get the system to work with a single switch, then a linear topology of switches followed by a mesh of switches one-by-one, the system tried to support multiple topologies with varying link bandwidths at the beginning. A more iterative approach would have yielded better progress. The key component to the whole system was correctly configuring the queue and rules settings of switches and it ended up being one of the few parts that didn't work in the end. If one was to re-implement a bandwidth reservation system similar to the one detailed in section 3, it would be advised to get the initial reservation mechanisms working on simpler and more manageable topologies before building for scale. The results detailed in section 4.4 indicate that Ryu can indeed provide mechanisms to reserve bandwidth. Attempts were made to get the system to work with a single switch but it still would not work meaning that it was an error in the logic of the bandwidth reservation system that was preventing it from working rather than a failure of the functionality provided by Ryu.

One challenging aspect of developing the bandwidth reservation system with Ryu was the lack of supporting documentation. The only official documentation is found at [*Ryu SDN Framework*]. Originally written by the Ryu project team in Japanese, the book has since been released in English. Although the book provides many step-by-step examples of how to leverage the APIs provided by Ryu, at certain parts, it seems as if some of the original intended meaning has been lost in translation which caused some confusion early on in the implementation stage. The relative newness of both the field and Ryu itself meant that when solutions to certain problems were discovered, no other occurrence of the problem could be found on the forums online and manual debugging was necessary.

Another aspect of the project that hampered progress was understanding how to structure an application in an SDN context. When initially developing, certain tutorials were followed to achieve topology discovery such as [*SDN Lab - Topology Discovery with Ryu* 2014] which suggested including business logic inside the Ryu applications themselves and calling the library functions rather than using an API. This approach doesn't align with how SDN applications should be structured as detailed in section 2.2.3. The separation of business logic from the NOS provides an additional layer of abstraction. Although the approach to topology discovery covered in that tutorial was never actually implemented, the idea of keeping the business logic coupled with the controller logic continued until very late into the implementation. Having the two components so tightly coupled created many implementation challenges that would have been avoided had the two elements been separated from the beginning. Only when they were separated and the project had a more SDN-appropriate architecture was significant progress achievable.

5.2 Evaluation of Results

The intention of this dissertation was to investigate what Ryu can lend to network operators in implementing bandwidth reservation. The two components that were identified as important to such a system were the ability of the controller to effectively determine the topological elements of the network and the bandwidth reservation functionality it offers.

Based on the results outlined in sections 4.2 and **??**, and the experience of developing a bandwidth reservation system using Ryu as the SDN controller, it has been determined that Ryu provides sufficient functionality to network operators to implement capacity reservation in their networks.

In terms of recommending an approach to bandwidth reservation, the use of meter tables should be considered above a customised implementation compromised of QoS rule and queue settings. Meter tables provide a layer of abstraction above the queueing infrastructure. They require less configuration and scale better due to the reduced number of flow entries, rules and queue settings required.

Although Ryu is lacking functionality such as host discovery and bandwidth discovery, it is believed that as the field of SDN and the number of people using Ryu as their selected controller increases, more attention will be given to improving the feature set provided. Similarly, as more people develop with Ryu, the number of online documentation, resources and forum posts will increase providing an improved knowledge base for using Ryu.

The results outlined in section 4 indicate that open source SDN controllers can compete with their proprietary counterparts.

5.3 Future Work

There are a number of improvements that would enhance the functionality of the bandwidth reservation system. The two most pertinent ones are improving the topology layer and adding functionality to dynamically re-allocate reservation flows.

5.3.1 Improved Topological Logic

The topology layer of the bandwidth system is the most lacking component. In particular, the function that finds a path through the network from one host to another is too static and naive to be implemented in a production environment. Mentioned briefly in section 3.1.1, the function does not account for cycles in topologies, does not take the bandwidth of the path into account and always returns the first available path from source to destination host.

An improved routing function would yield a more efficient and intuitive reservation system. It could return multiple paths for the QoSTracker component to choose from and return the available bandwidth of each option to enable a more informed routing decision. An alternative solution would be to replace the function completely with an implementation of Dijkstra's shortest path algorithm which could take into account the weights of paths (link bandwidths) in determining the best path.

5.3.2 More Intuitive Reservations

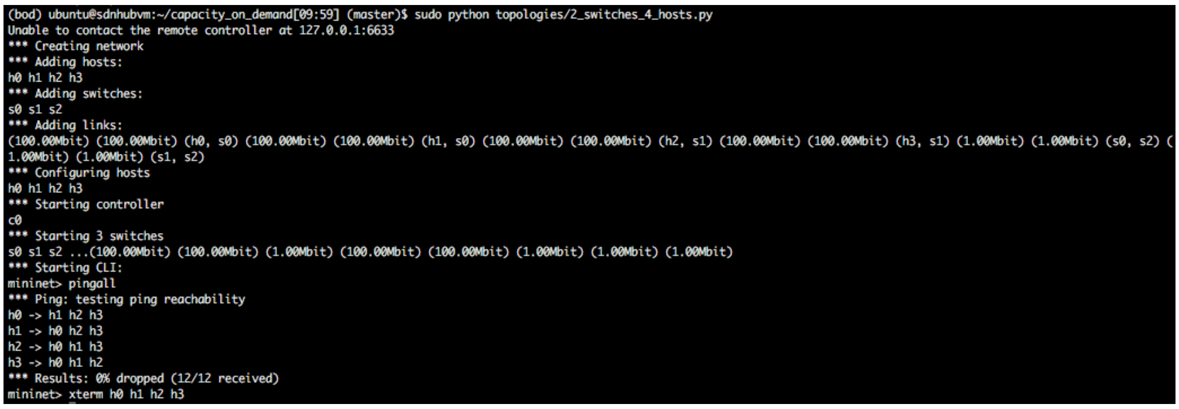
Currently, once a bandwidth reservation has been created, it's path through the network stays the same. As a particular path or entire network begins to reach maximum reservable capacity, the ability to rearrange reservation paths would improve the overall efficiency of the system. Relocating reservations to make room for incoming requests that otherwise would not be grantable would mean that more reservations are possible compared to the current non-intuitive approach. The ability to move reservations would enable the system to tolerate certain node failures - any bandwidth reservations affected by such a failure could be diverted around the dead node without impacting system functionality.

In addition to making reservation flows movable, adding a start and end time to reservations could enable improved levels of reservation organisation. A scheduler component could be added to schedule any upcoming reservations and interact with the flow moving capabilities mentioned previously to move existing capacity reservations to accommodate upcoming ones.

Appendix A

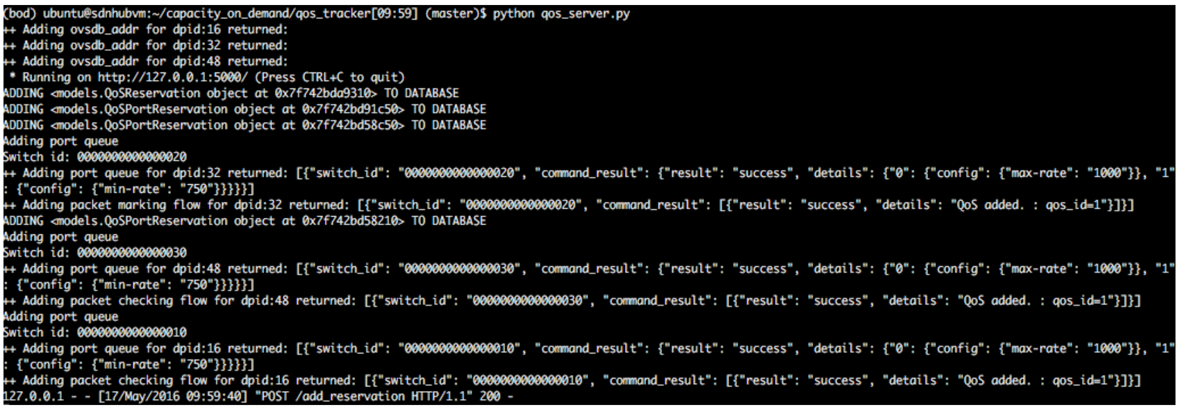
Complete Terminal Output from Section 4.2

For the example given in the results section, here is the complete terminal output for the mininet, Ryu and bandwidth reservation manager components.



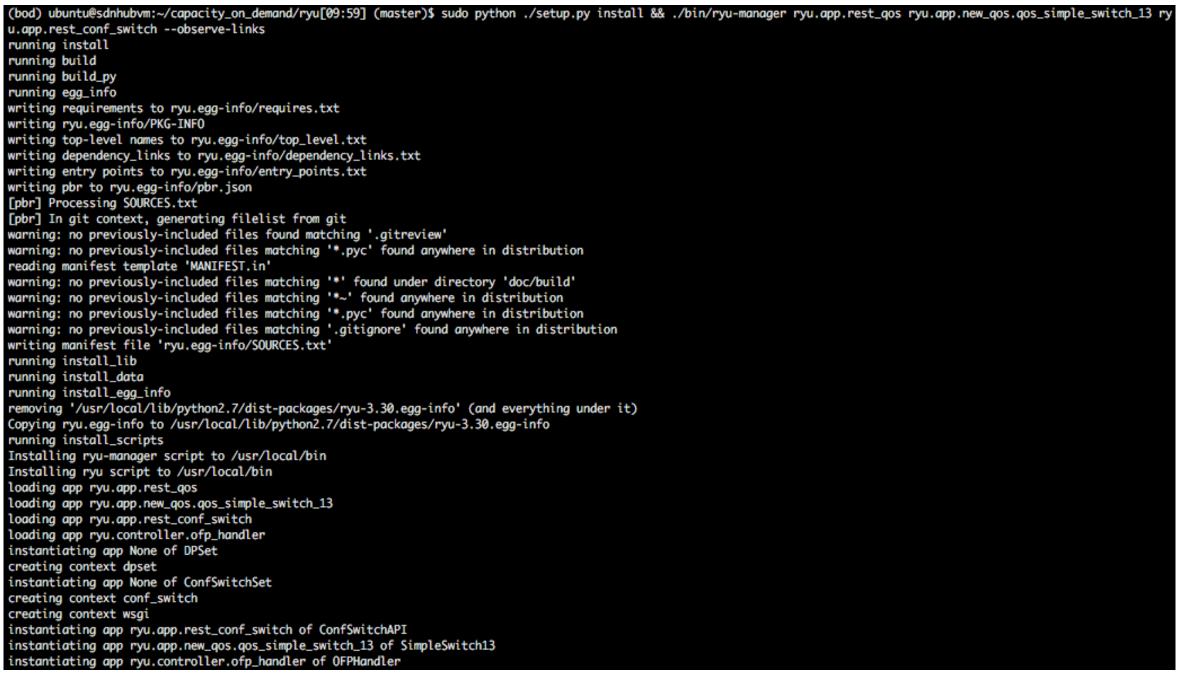
```
(bod) ubuntu@sdnhubvm:~/capacity_on_demand[09:59] (master)$ sudo python topologies/2_switches_4_hosts.py
Unable to contact the remote controller at 127.0.0.1:6633
*** Creating network
*** Adding hosts:
h0 h1 h2 h3
*** Adding switches:
s0 s1 s2
*** Adding links:
(100.00Mbit) (100.00Mbit) (h0, s0) (100.00Mbit) (100.00Mbit) (h1, s0) (100.00Mbit) (100.00Mbit) (h2, s1) (100.00Mbit) (100.00Mbit) (h3, s1) (1.00Mbit) (1.00Mbit) (s0, s2) (
1.00Mbit) (1.00Mbit) (s1, s2)
*** Configuring hosts
h0 h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s0 s1 s2 ...(100.00Mbit) (100.00Mbit) (1.00Mbit) (100.00Mbit) (100.00Mbit) (1.00Mbit) (1.00Mbit) (1.00Mbit)
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h0 -> h1 h2 h3
h1 -> h0 h2 h3
h2 -> h0 h1 h3
h3 -> h0 h1 h2
*** Results: 0% dropped (12/12 received)
mininet> xterm h0 h1 h2 h3
```

FIGURE A.1: Mininet output.



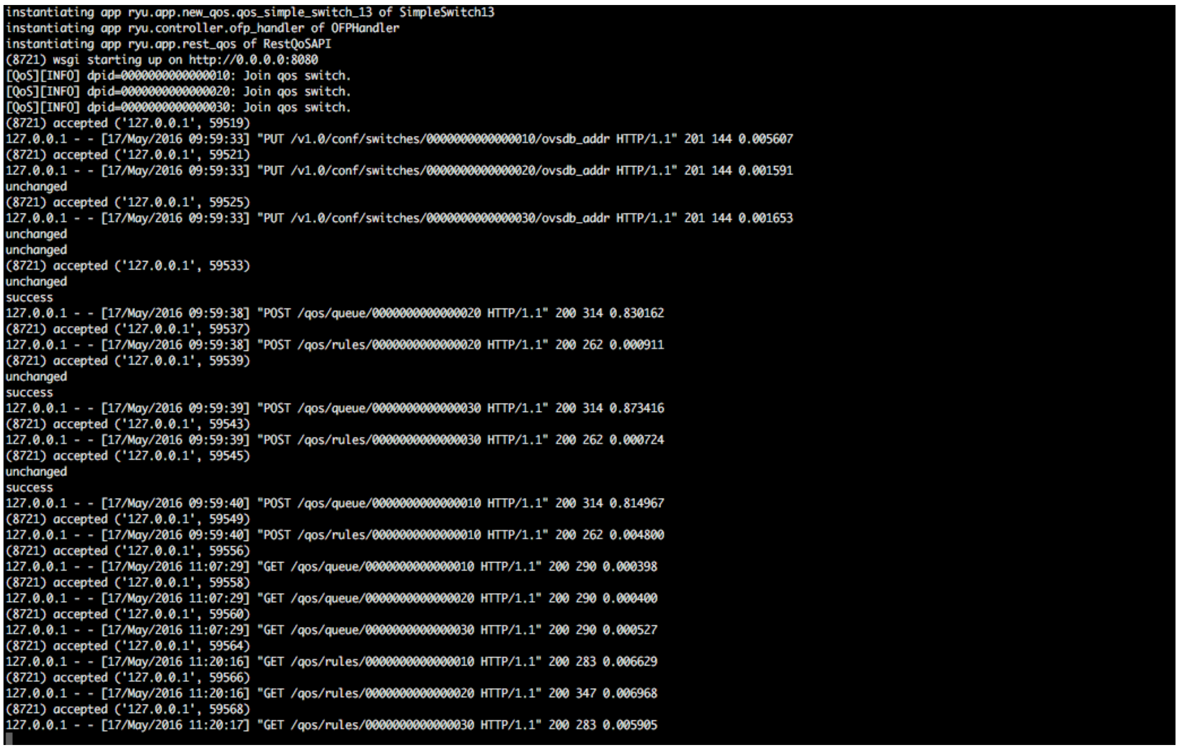
```
(bod) ubuntu@sdnhubvm:~/capacity_on_demand/qos_tracker[09:59] (master)$ python qos_server.py
++ Adding ovsdb_addr for dpid:16 returned:
++ Adding ovsdb_addr for dpid:32 returned:
++ Adding ovsdb_addr for dpid:48 returned:
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
ADDING <models.QoSReservation object at 0x7f742bda9310> TO DATABASE
ADDING <models.QoSPortReservation object at 0x7f742bd91c50> TO DATABASE
ADDING <models.QoSPortReservation object at 0x7f742bd58c50> TO DATABASE
Adding port queue
Switch id: 0000000000000020
++ Adding port queue for dpid:32 returned: [{"switch_id": "0000000000000020", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1"
: {"config": {"min-rate": "750"}}}}}]
++ Adding packet marking flow for dpid:32 returned: [{"switch_id": "0000000000000020", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}]
ADDING <models.QoSPortReservation object at 0x7f742bd58210> TO DATABASE
Adding port queue
Switch id: 0000000000000030
++ Adding port queue for dpid:48 returned: [{"switch_id": "0000000000000030", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1"
: {"config": {"min-rate": "750"}}}}}]
++ Adding packet checking flow for dpid:48 returned: [{"switch_id": "0000000000000030", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}]
Adding port queue
Switch id: 0000000000000010
++ Adding port queue for dpid:16 returned: [{"switch_id": "0000000000000010", "command_result": {"result": "success", "details": {"0": {"config": {"max-rate": "1000"}}, "1"
: {"config": {"min-rate": "750"}}}}}]
++ Adding packet checking flow for dpid:16 returned: [{"switch_id": "0000000000000010", "command_result": [{"result": "success", "details": "QoS added. : qos_id=1"}]}]
127.0.0.1 - - [17/May/2016 09:59:40] "POST /add_reservation HTTP/1.1" 200 -
```

FIGURE A.2: Bandwidth reservation system terminal output.



```
(bod) ubuntu@sdnhubvm:~/capacity_on_demand/ryu[09:59] (master)$ sudo python ./setup.py install && ./bin/ryu-manager ryu.app.rest_qos ryu.app.new_qos.qos_simple_switch_13 ry
u.app.rest_conf_switch --observe-links
running install
running build
running build_py
running egg_info
writing requirements to ryu.egg-info/requires.txt
writing ryu.egg-info/PKG-INFO
writing top-level names to ryu.egg-info/top_level.txt
writing dependency_links to ryu.egg-info/dependency_links.txt
writing entry points to ryu.egg-info/entry_points.txt
writing pbr to ryu.egg-info/pbr.json
[pbr] Processing SOURCES.txt
[pbr] In git context, generating filelist from git
warning: no previously-included files found matching '.gitreview'
warning: no previously-included files matching '*.pyc' found anywhere in distribution
reading manifest template 'MANIFEST.in'
warning: no previously-included files matching '*' found under directory 'doc/build'
warning: no previously-included files matching '*~' found anywhere in distribution
warning: no previously-included files matching '*.pyc' found anywhere in distribution
warning: no previously-included files matching '.gitignore' found anywhere in distribution
writing manifest file 'ryu.egg-info/SOURCES.txt'
running install_lib
running install_data
running install_egg_info
removing '/usr/local/lib/python2.7/dist-packages/ryu-3.30.egg-info' (and everything under it)
Copying ryu.egg-info to /usr/local/lib/python2.7/dist-packages/ryu-3.30.egg-info
running install_scripts
Installing ryu-manager script to /usr/local/bin
Installing ryu script to /usr/local/bin
loading app ryu.app.rest_qos
loading app ryu.app.new_qos.qos_simple_switch_13
loading app ryu.app.rest_conf_switch
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
instantiating app None of ConfSwitchSet
creating context conf_switch
creating context wsgi
instantiating app ryu.app.rest_conf_switch of ConfSwitchAPI
instantiating app ryu.app.new_qos.qos_simple_switch_13 of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
```

FIGURE A.3: Ryu output part 1.



```
instantiating app ryu.app.new_qos.qos_simple_switch_13 of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.app.rest_qos of RestQoSAPI
(8721) wsgi starting up on http://0.0.0.0:8080
[QoS][INFO] dpid=0000000000000010: Join qos switch.
[QoS][INFO] dpid=0000000000000020: Join qos switch.
[QoS][INFO] dpid=0000000000000030: Join qos switch.
(8721) accepted ('127.0.0.1', 59519)
127.0.0.1 - - [17/May/2016 09:59:33] "PUT /v1.0/conf/switches/0000000000000010/ovsdb_addr HTTP/1.1" 201 144 0.005607
(8721) accepted ('127.0.0.1', 59521)
127.0.0.1 - - [17/May/2016 09:59:33] "PUT /v1.0/conf/switches/0000000000000020/ovsdb_addr HTTP/1.1" 201 144 0.001591
unchanged
(8721) accepted ('127.0.0.1', 59525)
127.0.0.1 - - [17/May/2016 09:59:33] "PUT /v1.0/conf/switches/0000000000000030/ovsdb_addr HTTP/1.1" 201 144 0.001653
unchanged
unchanged
(8721) accepted ('127.0.0.1', 59533)
unchanged
success
127.0.0.1 - - [17/May/2016 09:59:38] "POST /qos/queue/0000000000000020 HTTP/1.1" 200 314 0.830162
(8721) accepted ('127.0.0.1', 59537)
127.0.0.1 - - [17/May/2016 09:59:38] "POST /qos/rules/0000000000000020 HTTP/1.1" 200 262 0.000911
(8721) accepted ('127.0.0.1', 59539)
unchanged
success
127.0.0.1 - - [17/May/2016 09:59:39] "POST /qos/queue/0000000000000030 HTTP/1.1" 200 314 0.873416
(8721) accepted ('127.0.0.1', 59543)
127.0.0.1 - - [17/May/2016 09:59:39] "POST /qos/rules/0000000000000030 HTTP/1.1" 200 262 0.000724
(8721) accepted ('127.0.0.1', 59545)
unchanged
success
127.0.0.1 - - [17/May/2016 09:59:40] "POST /qos/queue/0000000000000010 HTTP/1.1" 200 314 0.814967
(8721) accepted ('127.0.0.1', 59549)
127.0.0.1 - - [17/May/2016 09:59:40] "POST /qos/rules/0000000000000010 HTTP/1.1" 200 262 0.004800
(8721) accepted ('127.0.0.1', 59556)
127.0.0.1 - - [17/May/2016 11:07:29] "GET /qos/queue/0000000000000010 HTTP/1.1" 200 290 0.000398
(8721) accepted ('127.0.0.1', 59558)
127.0.0.1 - - [17/May/2016 11:07:29] "GET /qos/queue/0000000000000020 HTTP/1.1" 200 290 0.000400
(8721) accepted ('127.0.0.1', 59560)
127.0.0.1 - - [17/May/2016 11:07:29] "GET /qos/queue/0000000000000030 HTTP/1.1" 200 290 0.000527
(8721) accepted ('127.0.0.1', 59564)
127.0.0.1 - - [17/May/2016 11:20:16] "GET /qos/rules/0000000000000010 HTTP/1.1" 200 283 0.006629
(8721) accepted ('127.0.0.1', 59566)
127.0.0.1 - - [17/May/2016 11:20:16] "GET /qos/rules/0000000000000020 HTTP/1.1" 200 347 0.006968
(8721) accepted ('127.0.0.1', 59568)
127.0.0.1 - - [17/May/2016 11:20:17] "GET /qos/rules/0000000000000030 HTTP/1.1" 200 283 0.005905
```

FIGURE A.4: Ryu output part 2.

Bibliography

Benson, Theophilus, Aditya Akella, and David Maltz (2009). “Unraveling the Complexity of Network Management”. In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI’09. Boston, Massachusetts: USENIX Association, pp. 335–348. URL: http://dl.acm.org/citation.cfm?id=1558977.1559000.

Berde, Pankaj et al. (2014). “ONOS: Towards an Open, Distributed SDN OS”. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: ACM, pp. 1–6. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620744.

Bernstein, G.M (2006). *IP Bandwidth on Demand and Traffic Engineering via Multi-Layer Transport Networks*. San Francisco, CA, USA.

Bill Oliver (2014). *Pica8: First to Adopt OpenFlow 1.4*. URL: http://www.tomsitpro.com/articles/pica8-openflow-1.4-sdn-switches,1-1927.html.

Chowdhury, NM Mosharaf Kabir and Raouf Boutaba (2010). “A survey of network virtualization”. In: *Computer Networks* 54.5, pp. 862–876.

Cisco (2015). *Cisco Plug-in for OpenFlow Configuration*. URL: http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst4500/XE3-7-0E/15-23E/configuration/guide/b-openflow-config/b-openflow-config_chapter_00.html.

Cisco Global IP Traffic Forecast (2015). URL: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html.

Erickson, David (2013). “The Beacon OpenFlow Controller”. In: *Proceedings of the second workshop on Hot topics in software defined networks*. HotSDN ’13. Hong Kong, China: ACM.

Ferro, Greg (2012). *Northbound API, Southbound API, East/North â LAN Navigation in an OpenFlow World and an SDN Compass*.

Grinberg, Miguel (2014). *Flask Web Development: Developing Web Applications with Python*. 1st. O’Reilly Media, Inc. ISBN: 1449372627, 9781449372620.

Grossman, Daniel B. (2013). *New Terminology and Clarifications for Diffserv*. RFC 3260. DOI: `10.17487/rfc3260`. URL: `https://rfc-editor.org/rfc/rfc3260.txt`.

Gude, Natasha et al. (2008a). "NOX: towards an operating system for networks". In: *Comp. Comm. Rev.* ISSN: 0146-4833. DOI: `10.1145/1384609.1384625`.

– (2008b). "NOX: towards an operating system for networks". In: *Comp. Comm. Rev.* ISSN: 0146-4833. DOI: `10.1145/1384609.1384625`.

Jain, Sushant et al. (2013). "B4: Experience with a Globally-deployed Software Defined Wan". In: *SIGCOMM Comput. Commun. Rev.* 43.4, pp. 3–14. ISSN: 0146-4833. DOI: `10.1145/2534169.2486019`. URL: `http://doi.acm.org/10.1145/2534169.2486019`.

Jamjoom, Hani, Dan Williams, and Upendra Sharma (2014). "Don'T Call Them Middleboxes, Call Them Middlepipes". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA: ACM, pp. 19–24. ISBN: 978-1-4503-2989-7. DOI: `10.1145/2620728.2620760`. URL: `http://doi.acm.org/10.1145/2620728.2620760`.

Korotky, Steven K. (2013). "Semi-Empirical Description and Projections of Internet Traffic Trends Using a Hyperbolic Compound Annual Growth Rate." In: *Bell Labs Technical Journal* 18.3, pp. 5–21. URL: `http://dblp.uni-trier.de/db/journals/bell/bell18.html#Korotky13`.

Kreutz, Diego et al. (2014). "Software-Defined Networking: A Comprehensive Survey". In: *CoRR* abs/1406.0440. URL: `http://arxiv.org/abs/1406.0440`.

Lantz, Bob, Brandon Heller, and Nick McKeown (2010). "A Network in a Laptop: Rapid Prototyping for Software-defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: ACM, 19:1–19:6. ISBN: 978-1-4503-0409-2. DOI: `10.1145/1868447.1868466`. URL: `http://doi.acm.org/10.1145/1868447.1868466`.

Lazar, AA, Koon-Seng Lim, and F. Marconcini (1996). "Realizing a foundation for programmability of ATM networks with the binding architecture". In: *Selected Areas in Communications, IEEE Journal on* 14.7, pp. 1214–1227. ISSN: 0733-8716. DOI: `10.1109/49.536363`.

McKeown, Nick et al. (2008). "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2, pp. 69–74. ISSN: 0146-4833. DOI: `10.1145/1355734.1355746`. URL: `http://doi.acm.org/10.1145/1355734.1355746`.

Messaoud, Aouadj et al. (2014). "Towards a virtualization-based control language for SDN platforms". In: *10th International Conference on Network and Service Management, CNSM 2014 and Workshop, Rio de Janeiro, Brazil, November 17-21, 2014*, pp. 324–327. DOI: `10.1109/CNSM.2014.7014185`. URL: `http://dx.doi.org/10.1109/CNSM.2014.7014185`.

Moy, J. (1998). *OSPF Version 2*. United States.

Nippon Telegraph and Telephone Corporation (2012). *Ryu Network Operating System*. URL: `http://osrg.github.com/ryu/`.

ONF (2014). *Open Networking Foundation - https://www.opennetworking.org/*. URL: `https://www.opennetworking.org/`.

Open vSwitch - http://vswitch.org (2013). URL: `http://vswitch.org/`.

OpenFlow Switch Specification (2012). URL: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf`.

Orebaugh, Angela et al. (2007). *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Syngress Publishing. ISBN: 1597490733, 9781597490733.

Pfaff, B. and B. Davie (2013a). *The Open vSwitch Database Management Protocol*. RFC 7047 (Informational). Internet Engineering Task Force. URL: `http://www.ietf.org/rfc/rfc7047.txt`.

Pfaff, Ben and Bruce Davie (2013b). *The Open vSwitch Database Management Protocol*. Internet-Draft draft-pfaff-ovsdb-proto-00. Work in Progress. Internet Engineering Task Force. 33 pp. URL: `https://tools.ietf.org/html/draft-pfaff-ovsdb-proto-00`.

Pfaff, Ben et al. (2015). "The Design and Implementation of Open vSwitch". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, pp. 117–130. ISBN: 978-1-931971-218. URL: `https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff`.

Prince, Matthew (2013). *Happy IPv6 Day: Usage On the Rise, Attacks Too*. URL: `https://blog.cloudflare.com/ipv6-day-usage-attacks-rise/` (visited on 06/06/2013).

Rosen, E., A. Viswanathan, and R. Callon (2001). *Multiprotocol Label Switching Architecture*. United States.

Ryu SDN Framework. URL: `https://osrg.github.io/ryu-book/en/Ryubook.pdf`.

SDN Lab - Topology Discovery with Ryu (2014). URL: `https://sdn-lab.com/2014/12/31/topology-discovery-with-ryu/`.

Sheinbein, D. and R. P. Weber (1982). “800 Service Using SPC Network Capability”. In: *The Bell System Technical Journal* 61.7.

Suresh, Lalith et al. (2012). “Towards Programmable Enterprise WLANS with Odin”. In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN ’12. Helsinki, Finland: ACM, pp. 115–120. ISBN: 978-1-4503-1477-0. DOI: 10.1145/2342441.2342465. URL: http://doi.acm.org/10.1145/2342441.2342465.

Tennenhouse, D.L. et al. (1997). “A survey of active network research”. In: *Communications Magazine, IEEE* 35.1, pp. 80–86. ISSN: 0163-6804. DOI: 10.1109/35.568214.

Weissberger, Alan (2013). *VMware’s Network Virtualization Poses Huge Threat to Data Center Switch Fabric Vendors*. URL: http://viodi.com/2013/05/06/vmwares-network-virtualization-poses-huge-threat-to-data-center-switch-fabric-vendors/.

Zimmermann, H. (1988). “Innovations in Internetworking”. In: ed. by C. Partridge. Norwood, MA, USA: Artech House, Inc. Chap. OSI Reference Model&Mdash;The ISO Model of Architecture for Open Systems Interconnection, pp. 2–9. ISBN: 0-89006-337-0. URL: http://dl.acm.org/citation.cfm?id=59309.59310.