Providing End-to-end Bandwidth Guarantees with OpenFlow

Hedi Krishna

Master of Science Thesis

TU Delft Delft University of Technology

Network Architectures and Services

PVM 2016-085

Faculty of Electrical Engineering, Mathematics and Computer Science
Network Architectures and Services Group

Providing End-to-end Bandwidth Guarantees with OpenFlow

Hedi Krishna
4416910

Committee members:
Supervisor: Dr. Ir. Fernando Kuipers
Mentor: Ir. Niels van Adrichem
Member: Dr. Przemyslaw Pawelczak
Member: Ir. Rogier Noldus

July 22, 2016
M.Sc. Thesis No: PVM 2016-085

Copyright © 2016 by Hedi Krishna
All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the permission from the author and Delft University of Technology.

Abstract

QoS (Quality of Service) control is an important concept in computer networking as it is related to end user experience. End-to-end QoS guarantees, in particular, can give firm guarantees to end hosts. Unfortunately, it has never actually been used on the Internet since it was deemed too complicated. With the emergence of Software Defined Networking (SDN) and OpenFlow as its most popular standards, we have an opportunity to re-introduce the QoS control concept. The centralized nature and programmability of OpenFlow allow more flexible and more simple QoS control.

In this thesis, we propose an end-to-end bandwidth guaranteeing model for OpenFlow. The primary design consideration of the model is to allow QoS flow to send more than its guaranteed rate. To further maximize the overall network utilization, best-effort flows are allowed to use any unused bandwidth in the network. Bandwidth borrowing concept is employed to achieved this. To ensure that it will not affect the guaranteed bandwidth for the QoS flows, we analyze the reliability of the bandwidth borrowing concept in Linux HTB, which is used as the underlying mechanism of OpenFlow queue. From the simulations, we found that the borrowed bandwidth is returned instantly when a QoS flow requires the bandwidth. Thus, it is possible to guarantee bandwidth and maximize bandwidth utilization at the same time.

We also explore the possibility of using OpenFlow meter table for traffic aggregation. The aggregation only puts overheads in the first switch, but no other complexities added in the subsequent switches. Therefore, it solves the scalability problem which commonly associated with end-to-end QoS guarantees.

Table of Contents

Abstract i

Acknowledegement xi

1 Introduction 1

1-1 Background . . . 1

1-2 Problem Description . . . 1

1-3 Research Objective . . . 3

1-4 Research Question . . . 3

1-5 Thesis Structure . . . 3

2 SDN, OpenFlow, and QoS 5

2-1 Software Defined Networking . . . 5

2-2 OpenFlow . . . 7

2-2-1 The OpenFlow standard . . . 7

2-2-2 OVSDB . . . 9

2-3 Quality of Service . . . 10

2-3-1 IntServ . . . 11

2-3-2 DiffServ . . . 12

2-4 Related research in OpenFlow QoS . . . 13

3 QoS support in OpenFlow 17

3-1 QoS in the OpenFlow standard . . . 17

3-2 Queue . . . 17

3-2-1 Linux Traffic Control . . . 19

3-2-2 Parallels between OVS's queue and TC classes . . . 20

3-3 Meter table . . . 21

4 Enabling End-to-end Bandwidth Guarantees in OpenFlow 23

4-1 Assumptions . . . 23

4-2 Traffic types . . . 24

4-3 Per-flow, end-to-end bandwidth guarantees . . . 25

4-4 Routing, admission, and reservation . . . 26

4-5 Queue management . . . 28

4-6 Prioritizing best-effort traffic over excess QoS . . . 29

4-7 General program flow . . . 30

5 Proof of Concept 33

5-1 Admission and Reservation . . . 33

5-1-1 Experiment setup . . . 33

5-1-2 Experiment result . . . 34

5-2 Traffic Prioritization . . . 36

5-2-1 Simulation environment . . . 36

5-2-2 Experiment Result . . . 37

6 System test 39

6-1 Bandwidth guaranteeing in single and multiple queues system . . . 39

6-2 TC's traffic shaping reliability . . . 41

6-3 Loading test . . . 45

6-4 Multi-queues system suitability . . . 47

7 Meter Table and Flow Aggregation 49

7-1 Metering and Aggregation Concept . . . 49

7-2 Metering in Pica8 switch . . . 51

8 Conclusion and Future Work **55**
8-1 Conclusion 55
8-2 Future work 56

A Workaround for SURFnet Pica8 Testbed **61**

B Controller Source Code **63**

List of Figures

2-1 SDN vs legacy network architecture [17] 6
2-2 OpenFlow architecture [22] 8
2-3 OVSDB architecture [26] 9
2-4 Open vSwitch database schema [27] 10
2-5 RSVP operation 12

3-1 OVSDB database schema 18
3-2 Example of HTB classes 19

4-1 Comparison of reservation signals required for various flow lifetime 26
4-2 Example of queue arrangement 29
4-3 General flow of the system 31

5-1 Topology for admission test 34
5-2 Admission-reservation test result 35
5-3 Topology for prioritization test 36
5-4 Prioritization test result 38

6-1 Topology used in experiment 6-1 40
6-2 Received throughput for experiment 6-1 40
6-3 Topology used in experiment 6-2 41
6-4 Packet loss in traffic shaping reliability simulation 42
6-5 Flow entries installation time for various number of hops 43

6-6 Received throughput of long lived best-effort flow and on-off QoS flow 44
6-7 Network topology for loading test 46
6-8 Received throughput, loading test experiment 47
6-9 Queues creation time for various number of queue 48

7-1 Metering-aggregation concept 50
7-2 Topology for the metering and aggregation experiment 52
7-3 Prioritization test with metering in Pica8 testbed 53

A-1 Workaround for Pica8 testbed 62

Glossary

ATM Asynchronous Transfer Mode

BE Best Effort

CBQ Class Based Queueing

CIR Committed Information Rate

DiffServ Differentiated Service

DSCP Differentiated Services Code Point

EIR Excess Information Rate

FIFO First In First Out

HFSC Hierarchical Fair-Service Curve

HTB Hierarchical Token Bucket

IntServ Integrated Service

IP Internet Protocol

OVS Open vSwitch

PIR Peak Information Rate

QoS Quality of Service

RSVP Resource Reservation Protocol

RTT Round Trip Time

SLA Service Level Agreement

WSP Widest-Shortest Path

Acknowledegement

I would like to gratefully acknowledge PT. Telkom Indonesia for giving me an opportunity to pursue my master's degree. I would also like to express my gratitude to my supervisor Fernando Kuipers and NAS group of TU Delft. Special thanks to Niels van Adrichem for supporting me, academically and administratively, in this thesis during every stage of its development. Also, Ronald van der Pol and SURFNet for the Pica8 testbed.

Lastly, I want to thank my family and friends, especially Putri and Aliya. Their support has been wonderful throughout my study.

Chapter 1

Introduction

1-1 Background

According to the QoS Forum, Quality of Service (QoS) is the ability of a network element to have some level of assurance that its traffic and service requirements can be satisfied. QoS reflects the performance an application may require and experience in a network. It can be considered as subjective, as users might have different perspectives of quality [1]. QoS is particularly important for applications with strict requirements such as telephony and multimedia delivery. Nevertheless, QoS control has never actually been used in the Internet. End-to-end QoS such as Integrated Service (IntServ) [2] is deemed too complex and not scalable. On the other hand, Differentiated Service (DiffServ) [3] with its aggregation model does not provide QoS guarantees. Service providers prefer to over-provision their network with more resources, simply because it is less complicated than QoS control. Unfortunately, since most of the time the network is not running at full capacity, it leads to low network utilization [4]. This is more apparent in the recent years where the emergence of throughput intensive applications forces service providers to switch to Gigabit networks.

Software Defined Networking (SDN), as a new paradigm in networking offers the opportunity to re-introduce QoS control in the Internet. The centralized nature of SDN significantly reduces the complexity that is commonly associated with end-to-end QoS guarantees. With its adoption and support by leading companies in the tech industry, SDN is well on its way to be adopted as a de-facto standard. By having strong QoS control included in SDN, the future Internet might have native QoS support.

1-2 Problem Description

The Internet Protocol (IP), the underlying technology for the Internet that we are using today, was not designed with QoS in mind. It was initially designed to provide Best Effort (BE) service only. Therefore, unlike Asynchronous Transfer Mode (ATM) [5], the Internet

does not have a native QoS capability. Later, in order to accommodate applications that require certain QoS, the Internet Engineering Task Force (IETF) defined two major QoS control architectures, i.e.: Integrated Service (IntServ) and Differentiated Service (DiffServ).

IntServ provides an end-to-end QoS solution with bandwidth reservation and admission control at each network element. The IntServ reservation system ensures that the portion of bandwidth reserved by a flow, in every link that is used by the flow, can only be used by that particular flow. Nonetheless, IntServ has its problems. First, its reservation signaling system, the Resource Reservation Protocol (RSVP) [6], is not scalable; thus, it is not fit for bigger networks like the Internet. In Intserv, each network element needs to store flow states, and it grows rapidly with increasing number of flows and network elements. To make it worse, RSVP is a soft state protocol which requires periodic reservation-state refresh. Second, the fact that the reserved bandwidth can only be used by the reserving flow might cause low bandwidth utilization.

On the other hand, a soft-QoS architecture such as DiffServ only gives a loose notion of QoS. Rather than providing an end-to-end guarantee for flow, DiffServ employs per-hop behavior (PHB) with aggregation for different classes of traffic. Although DiffServ complexity is significantly lower than IntServ, bandwidth is shared between flows; therefore, there are no QoS guarantees in this architecture.

OpenFlow as one implementation of SDN supports QoS since its early versions with OpenFlow queue. Using this support, researchers try to propose QoS models for OpenFlow. Many of these models, such as [7], [8], [9] and [10], are loosely based on DiffServ; thus, they do not provide end-to-end QoS guarantees. Other models, [11] and [12], give hard QoS guarantees inspired by IntServ. These IntServ based models guarantee bandwidth to QoS flows with strict separation between QoS flows and best-effort flows. Unfortunately, the strictness also put limits on the maximum allowable rate of QoS flows. QoS flows in these models are not allowed to send more than its guaranteed rate. Any excess (non-conformant) traffic of QoS flow will be dropped. The reason for this restriction is because excess QoS traffic might compete with conformant traffic of other QoS flows and best-effort traffic.

Nevertheless, putting a maximum rate limit to a flow might cause a loss of opportunity as the utilization of network bandwidth is not maximized. In a link with low utilization, for example, it will be better to let QoS flow to send more than its guaranteed rate. By doing so, the flow's data transfer process can be finished sooner, freeing the bandwidth to be used by other flows that come next.

In this thesis, we propose a new end-to-end bandwidth guarantees model in which QoS flows are allowed to send more than their guaranteed rates. The QoS guarantees is implemented in both controller and switch level. In the controller, an admission process is performed to make sure that the QoS flows get enough bandwidth. Then, in the switches, OpenFlow queue is used for traffic shaping and policing to ensure bandwidth guarantees. We took our model further by addressing the problem of contention between excess QoS traffic and best-effort traffic. By arranging prioritization of the traffic, best-effort will have more priority than excess QoS traffic.

We also explore the possibility of using OpenFlow meter table for traffic aggregation in our model. The application of meters will reduce the number of queues used in switches by aggregating flows; thus, alleviating the scalability problem. To the best of our knowledge,

the implementation of meter table in end-to-end QoS guarantees and the prioritization of best-effort over excess QoS have never been done before.

1-3 Research Objective

The main objective of this thesis is to enable end-to-end bandwidth guarantees in OpenFlow. The system should have the following three features:

1. Give a per-flow level bandwidth guarantees.
2. Allowing excess QoS traffic.
3. Prioritize best-effort traffic over excess QoS traffic.

1-4 Research Question

From the research objectives, research questions for this thesis can be deduced as the following:

1. How to provide end-to-end bandwidth guarantees in OpenFlow?
2. How to allow excess QoS traffic while still providing firm guarantee for conformant QoS traffic?
3. Does the excess QoS and best-effort traffic have adverse effects on the guaranteed traffic performance?
4. How can the OpenFlow meter table be used to enhance our end-to-end bandwidth guarantees model?

1-5 Thesis Structure

This thesis is structured as follows. Chapter 2 discusses the background on QoS, its implementation in the Internet, and SDN technology in general. OpenFlow, as the most popular implementation of SDN, will be discussed in detail. Chapter 3 covers the current QoS support in OpenFlow, the underlying mechanism of QoS in OpenFlow, and how it is implemented in Open vSwitch (OVS). Previous works on QoS in OpenFlow, particularly end-to-end QoS guarantees, will also be reviewed in this chapter. Chapter 4 focuses on the proposed solution for the research questions. Assumptions made and design considerations taken in this thesis will be explained in this chapter. In Chapter 5, two simulations are conducted as proof of concept for QoS admission control, reservation, and class prioritization. The result of the experiments provides background for the main simulation in the next chapter. A simulation for the proposed algorithm is conducted in the Chapter 6. Here, the "goodness of the system" will be observed, analyzed and discussed. The simulations for Chapter 4 and 5 are conducted in OVS.

While in the previous chapters our models only employs OpenFlow queue to give bandwidth guarantees, in chapter 7 we will explore the possibility of using meter table to enhance our model with traffic aggregation. The experiment is conducted with Pica8 hardware switch. Chapter 8 finalizes this thesis by presenting conclusion and future work.

Chapter 2

SDN, OpenFlow, and QoS

In this chapter, a brief background of Software Defined Networking, OpenFlow, and Quality of Service are given. The first section examines Software Defined Networking and compares it to the "traditional" IP networking. The emergence of OpenFlow, its architecture, working principle and implementation are discussed in the second section. A discussion about OVSDB, which is fundamental for understanding OpenFlow queue, is also provided in this section. Section 2.3 provides a discussion on QoS control in general, why is it important, and the problem with today's Internet QoS. This chapter finishes with a literature review on the topic of QoS in OpenFlow. Several previous studies in this field and proposed OpenFlow QoS models will be presented here.

2-1 Software Defined Networking

The control system of traditional IP networking that we are using today is distributed. Each network element is a separate entity with its own control plane and forwarding plane. The control plane is responsible for device configurations and path computation for data flow. Decisions made by the control plane are then informed to the forwarding plane, in which packets are forwarded to a specific port. The forwarding plane also handles input and output control such as traffic shaping and policing whenever it is necessary. Having control and forwarding plane in every network elements means that all devices in the network make their own decision by using information that is shared with one another.

The distributed control system surely has its benefit. Each network element is autonomous; thus, there is no single point of failure problem. Furthermore, this is a mature system which has been used for decades, and every network administrator has been familiar with. But it also comes with disadvantages. In a large traditional network with many devices, network administrators should manually configure each of these network devices (routers and switches) one by one. The process is tedious and prone to human error [13]. It also significantly increases provisioning time.

To make it worse, different products of various manufacturers typically have different configuration interfaces. For example, Cisco uses Cisco IOS (Internetwork Operating System), while Juniper products have Junos OS. Sometimes, even different products from a single vendor have different configuration interfaces. Network administrators are forced to learn all these different interfaces.

These adversities led to the idea of control plane - forwarding plane separation, which in turn resulted in the invention of Software Defined Networking. In SDN, the control plane is extracted from network devices. A network device becomes a “dumb” device that only performs forwarding. The control function itself is performed by a single entity that is simply called the “Controller”.

In most cases, the controller is a general purpose server. It can easily be upgraded whenever necessary. The controller can even be distributed into several separate machines to improve its scalability and reliability [14] [15] [16]. The network devices itself are less specialized. A single SDN device can be whatever the network administrator wants it to be, whether it is a switch, a router, or even a middlebox such as a firewall, NAT (Network Address Translator) or load balancer.

Since the operating system, which is usually proprietary in traditional networking, is now extracted into the controller, ideally the SDN system will be vendor agnostic. Network operators can use any SDN-enabled network device available in the market, and incorporate it into their network without any issue. In the long term, this will significantly lower the operational expenditures.

The central processing allows network administrators to do more efficient network related operations such as routing. New routing policies, or even a new routing algorithm, can be easily deployed in the network. This increases the rate of innovation in the field of computer networks.

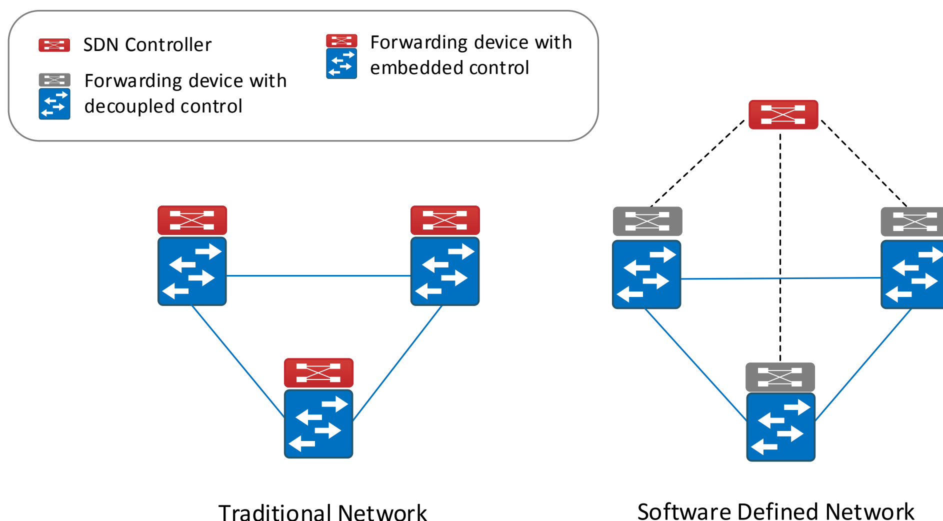


Figure 2-1: SDN vs legacy network architecture [17]

Another important aspect of SDN is its programmability. The programmability of SDN

allows flexible and elastic networks. Applications such as cloud-based networks, virtualized desktop and servers and remote storage can be easily deployed with SDN. Even security can be embedded in the network application itself.

From a QoS perspective, having a centralized control system simplifies many things that otherwise are difficult to perform. Central control allows the controller to have a global view of the network. Network metrics such as hop count between switches, delays, jitter and available bandwidth can be obtained efficiently in a near real time from the network. In QoS routing, fresh values of network metrics are required, as opposed to static metrics in non-QoS routing [18]. Therefore, SDN simplifies the QoS routing process.

Network administrators are now able to use any routing algorithm that fits their needs. They can employ per-flow routing, using different routing metrics for different types of flow. A delay sensitive flow such as telephony might be forwarded via a minimum delay path, while a throughput sensitive application uses a path with highest available bandwidth. It is even possible to compute paths based on multiple constraints [19].

2-2 OpenFlow

OpenFlow is, arguably, the first and most popular implementation of SDN. OpenFlow defines a set of protocols that enables SDN controller to communicate directly with the forwarding plane in network devices. It started as a research project at Stanford University in 2008. The initial goal was to provide a method to do an experiment in a production network. The first protocol specification, version 1.0, was released in 2009. OpenFlow is currently managed by the Open Networking Foundation (ONF).

From there, it widespread, not only in the academic community but also in the computer networking industry. One of the most successful deployments of OpenFlow in large scale is Google B4 [20] in 2013. In this project, Google implements OpenFlow in their Wide Area Network to connect their data centers all around the world.

2-2-1 The OpenFlow standard

OpenFlow protocol standardizes exchanged messages between the controller and switch. In general, these messages contain instruction how switch should handle specific types of packets and collect statistics of these flows.

An OpenFlow switch has one or multiple flow tables to determine how to forward incoming packets. These tables are functioned as lookup tables, to whom the switch learns what to do with the packets. When a new flow arrives, the switch looks to its flow table. If there is no entry matching this flow, the packet is sent to the controller as a `packet_in` message. Based on the controller's application, the controller decides what the network should do with this particular flow (it might be forwarded to a particular port, dropped, or sent back to the sender). The controller then informs the switches about the decision by installing flow entries in switch's flow tables. While waiting for a decision from the controller, packets that do not match any flow entries are stored in a buffer. When the buffer is full, the newly incoming packets will be sent to the controller. If the link to the controller is full, or the controller becomes overloaded, these packets will be dropped.

Flow entry is used to identify and process packets. It contains a set of *match fields*, *priority*, *counters*, *timeout*, and a set of *instructions* [21]. The *match field* is a field against which a packet is matched. As of OpenFlow 1.3, it has 40 matching fields, varies from IP source/destination address to MPLS label. A packet might match several entries in flow tables, in this case, the entry with the highest *priority* will be used to process the packet. *Instruction* describes the processing that needs to be taken when a packet matches a flow entry. It contains a set of actions that will be applied to a packet (such as rewrite IP header, or send to switch output port) or modifies pipeline processing (such as direct the packet for further processing in the next flow table).

There are two ways a flow entry removed from a flow table, either at the controller's request (using `OFPFC_DELETE` message) or via flow expiry mechanism. Each flow entry has `idle_timeout` and `hard_timeout` parameters attached to it. `Idle_timeout` causes flow entry to be removed when it matched no packets after the time it specifies. On the other hand, flow entry will always be removed after `hard_timeout` since the flow is installed, disregarding of how many packets has been matched. When a flow entry is removed, the switch will send a flow removal message to the controller. This message contains the description of the flow entry, the reason for removal, and the flow statistics [21].

Started from OpenFlow 1.3, a switch might have multiple flow tables. The switch will always start the lookup process from table 0. If there is a `go_to` instruction in the flow entry, the flow will be processed further in the next flow table. The multiple tables and pipeline processing allows more flexible flow matching and processing.

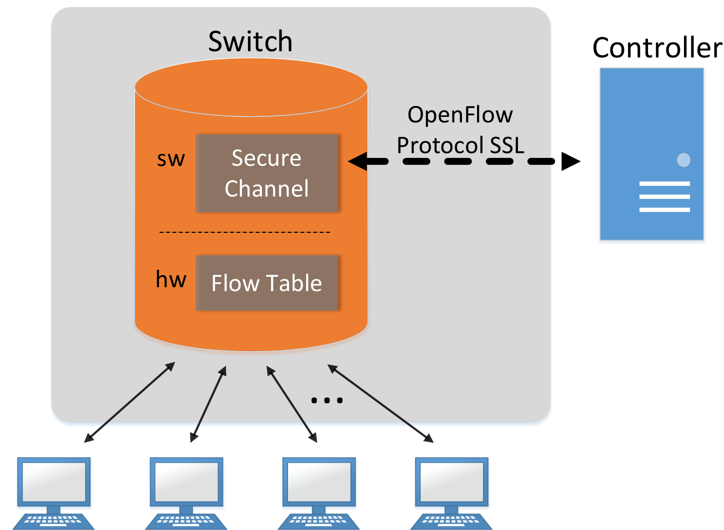


Figure 2-2: OpenFlow architecture [22]

Figure 2-2 shows the basic architecture of OpenFlow. The OpenFlow protocol runs over OpenFlow channel between the controller and switch. In a typical setting, an OpenFlow controller manages multiple connections, each with a different switch. The switch may be connected to a single controller, or multiple controllers using multiple OpenFlow connections.

An OpenFlow switch can be hardware-based or software-based. Software switches such as Open vSwitch [23] and ofsoftswitch [24] can be installed in a general-purpose computer, adding switch functionality into it. One of the most interesting usages of software switch is in cloud computing, in which it provides network virtualization for virtual machines.

On the other hand, hardware switch is a dedicated hardware with networking functionality. As of 2016, big name vendors such as HP and IBM have released dedicated OpenFlow hardware switch products. Other vendors like Cisco and Juniper support OpenFlow in some of their legacy devices. Many hardware switches supporting OpenFlow also run Open vSwitch as a process in the operating system. For example, in Pica8 switch, OVS is a process within its PicOS operating system [25].

2-2-2 OVSDB

OpenFlow is a southbound protocol that enables communication between the controller and switch. The management of the switch itself is not managed by OpenFlow. For that purpose, two different protocols can be used, OVSDB and OF-Config.

OF-Config is management protocol developed by the ONF and is supposed to work with any OpenFlow-enabled device, while OVSDB protocol is specifically developed for Open vSwitch. OVSDB works with both software and hardware implementation of OVS, such as Pica8 [25].

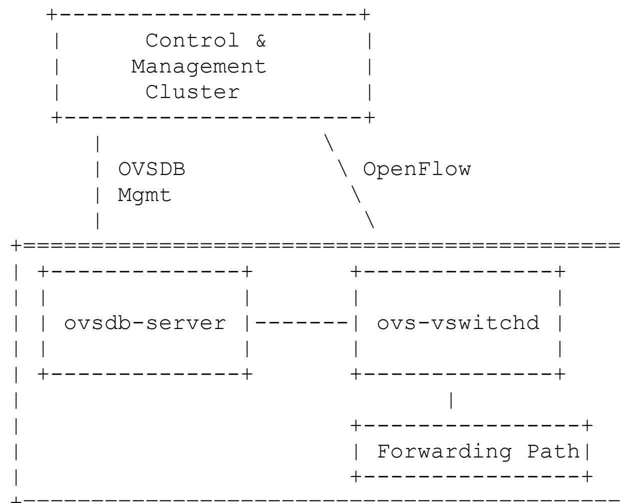


Figure 2-3: OVSDB architecture [26]

OVSDB manages switches operations such as creating interfaces, setting QoS policies, or shutting down a physical port. Figure 2-3 illustrates the architecture of OVSDB. An OVS instance consists of a database server (ovsdb-server) and a vswitch daemon (ovs-vswitchd). The *management and control cluster* are the OVSDB managers and OpenFlow controller, which can be located in the same or different devices. While the controller communicates with switches via OpenFlow channel, OVSDB server talks with its manager via OVSDB Management protocol. The OVSDB switch daemon, located in a switch, monitors the database for additions, deletions, and modifications to this information. Any change in the database is applied to the switch. The OVSDB server stores information about switches in database

form. The configuration in OVSDB is permanent; the switch will not lose its configuration in the event of switch restart.

OVSDB is formalized in RFC7047 [26]. Many OpenFlow controllers, such as Opendaylight and Ryu has integrated API to communicate with OVSDB. This support allows switch management to be incorporated into OpenFlow application.

The switch configuration in OVSDB is stored in database form. Figure 2-4 shows the database schema. Each node represents a table in the database, while the edges represent the relation between tables. Tables that are part of the "root set" are shown with double borders. Root set is tables whose entries will not be automatically deleted when is not reachable from the Open_vSwitch table. Each edge in the graph leads from the table that contains it and points to the table that its value represents. Edges are labeled with their column names. Symbols next to the label shows the number of allowed values: ? for zero or one, * for zero or more, + for one or more.

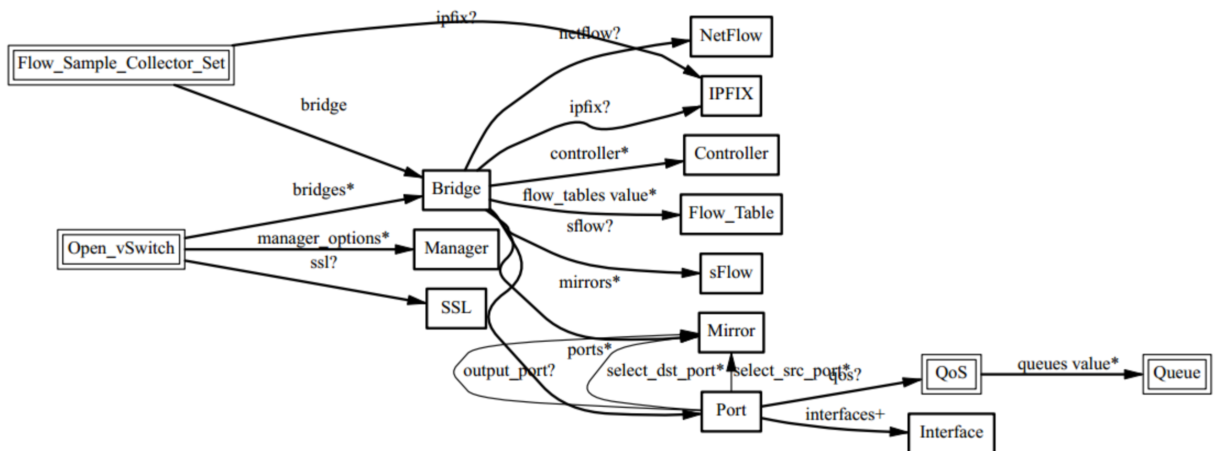


Figure 2-4: Open vSwitch database schema [27]

2-3 Quality of Service

Quality of Service (QoS) control is a mechanism used in a network to ensure high-quality performance. By using QoS control, network administrators can manage their resource more efficiently and provide a high level of service without having to over-provision the network.

QoS is of particular importance in applications that need specific guarantees. An application such as voice conversation or video streaming requires small delay and jitter to be as good as traditional telephone and television, which is demanded by users. On the other hand, data communication are less sensitive to delay and jitter, but more sensitive to packet loss. A failure to meet this standard might lowers Quality of Experience. In the future, we may see more of these "inelastic" applications.

The Internet Engineering Task Force (IETF) defines two major QoS control architecture, i.e.: Integrated Service (IntServ) [2] and Differentiated Service (DiffServ) [3].

We can classify rate guarantees into two categories, i.e: hard and soft guarantees. The hard guarantees is a rigid system where there is a portion of bandwidth capacity that can only be used by a specific flow. Typically, there is a reservation process in which a flow asserts how many bandwidth it wants to use. Hard QoS guarantees also have a strict admission control; when the requested bandwidth is not available, other flow requests are rejected. In that sense, it is similar to connection oriented circuit switched network. Soft guarantees, on the other hand, is more flexible but does not give strong guarantees.

2-3-1 IntServ

IntServ is the first attempt to establish QoS control in IP network. IntServ emulates the resource allocation concept of circuit switching. The network elements are forced to allocate resource for particular traffic flow, analogous to a circuit-switch call session.

There are three defined level of services in IntServ:

1. Guaranteed Service. The Guaranteed Service provides mathematically provable upper bounds on end-to-end delay that allows bandwidth, delay and packet loss guarantees. It is accomplished through a combination of packet classifiers, scheduler, and admission control.
2. Controller Load. The Controlled Load service provides flow with QoS approximating the QoS that it would receive from best-effort service in an unloaded network. This is achieved through admission control.
3. Best Effort. The best-effort traffic does not provide any QoS guarantees whatsoever. This service is similar to the current operation of the IP networking.

Intserv uses the Resource Reservation Protocol (RSVP) for resource reservation. RSVP is a multicast based signaling protocol which is a separate standard from IntServ. It is a soft state protocol. The state of reservation needs to be refreshed periodically; otherwise, the state will be lost.

RSVP has two types of messages to establish a connection: `Path` and `Resv` message. The Path message is sent by the sender. It contains information about previous hop IP address, traffic specification (including sender's address), traffic characteristic, and end-to-end QoS requirement. The `Resv` message is a "reply" to the Path message from the receiver to sender, traveling in the reverse direction of the `Path` message. Along the route, it set resource reservation for the flow.

To tear down a connection, RSVP uses `PathTear` and `ResvTear` message. The `PathTear` is sent by the sender to the receiver following the route used by Path message, while the `ResvTear` travels the reverse direction. The two tear messages remove path state installed and the reservation state.

The main advantage of IntServ is that it provides service classes differentiation. Users can define their traffic type and uses one service that fit their application. A critical and intolerant application can use guaranteed service. While, a critical but more tolerant application can be supported with Controlled Load service. Other elastic applications can use best-effort service.

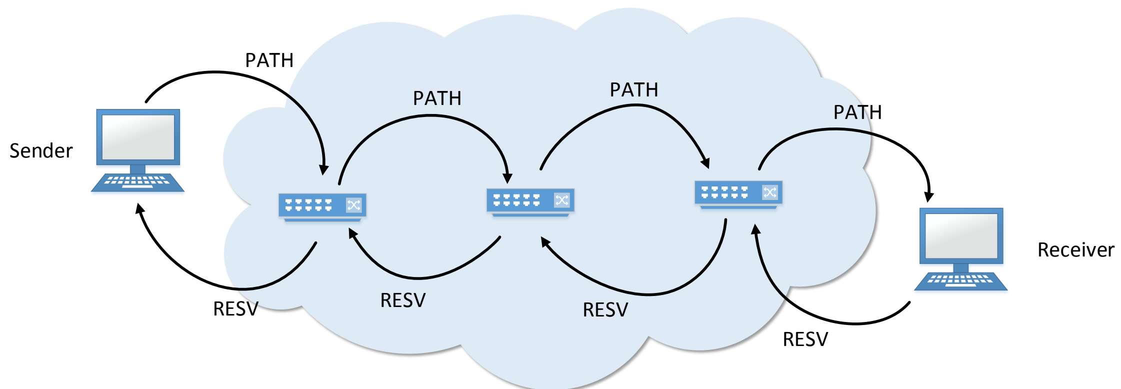


Figure 2-5: RSVP operation

The main disadvantage of IntServ is its scalability problem. RSVP requires an end-to-end signaling and must maintain per-flow soft state at every node along the path. The periodic refresh requires a significant amount of signaling message and only gets bigger when the number of flows and nodes in the network are increasing. The other problem is that IntServ has to store all flows information in all switches/routers in the path, which becomes expensive in a network with many flows. Because of this problem, it has never been used on the Internet and only thrives in local enterprise networks.

2-3-2 DiffServ

DiffServ was introduced to address IntServ's scalability problem. Unlike IntServ, in which flows are treated end-to-end, DiffServ applies its policy when traversing a hop, known as per-hop behaviour (PHB). DiffServ also acts on aggregated flows, rather than treats flows separately. Packets are classified based on its Differentiated Services Code Point (DSCP) bits, which is located in the IP header. Packets with same DSCP bits received equal treatment in router/switch, no matter to which flow they belong.

After the classification, each class is treated according to the applied PHBs. There are several PHBs defined for DiffServ, the most common are:

1. Default PHB. Packets with DSCP bits 000000 and those that do not meet requirements of other classes are forwarded with a best-effort characteristic.

2. Expedited Forwarding. EF PHB has a characteristic of low delay, low loss, and low jitter. EF packets are given priority queuing above other traffic classes. Flows using EF will still compete with each other.

3. Assured Forwarding. AF PHB is analogous to IntServ's Controlled Load. It provides assurance of delivery as long as the traffic does not exceed a particular rate.

4. Class Selector. This PHB is defined to preserve backward compatibility with IP-precedence scheme that predates DiffServ.

DiffServ is more scalable than IntServ since it uses flow aggregates and does not need per-flow states, significantly reduces signaling needed. It is also readily available without setup delay since there is no admission process. Nonetheless, DiffServ does not provide end-to-end QoS guarantees. It is possible to give more priority to one class over other classes, but it is impossible to provide certain QoS guarantees in a flow level.

2-4 Related research in OpenFlow QoS

OpenFlow QoS is said to be limited since it only realized with two features, i.e. queue and meter table. There are no hard defined standards like IntServ and DiffServ in OpenFlow. In one way, it can be seen as a flexibility for network administrators to implement their own QoS algorithm, but on the other hand, it adds difficulty to implementing QoS in OpenFlow.

This lack of a QoS model in OpenFlow inspired some researchers to propose such models. Some of these proposals are high-level frameworks covering complete QoS aspects, while others cover specific areas, such as rate guaranteeing, automation of QoS control, and QoS implementation in home networks.

[28][29] propose a high-level QoS framework. In [28], a QoS extension for the Ofelia [30] testbed is proposed. The framework utilizes the Queue table in OpenFlow 1.3 to enable QoS control in multiple types of OpenFlow-enabled hardware switches. The model focuses on how a northbound interface management system is created to work with different switches with nonuniform queue implementation. Jeong et al. [29] propose Service Level Agreement (SLA)-conscious QoS control based on Multiprotocol Label Switching - Transport Profile (MPLS-TP). However, these two papers only cover the high-level concept and do not discuss in detail how QoS and bandwidth guaranteeing are performed in the framework.

Other papers [7][8][9][31][10] attempt to achieve QoS in OpenFlow by utilizing QoS routing and rate guaranteeing.

HiQoS [7] investigates the usage of DiffServ and multipath QoS routing in OpenFlow. HiQoS differentiates traffic into three categories, i.e. high throughput video streaming, low delay interactive multimedia, and best effort data stream. In the switches, these three types of traffic are forwarded via three preconfigured queues with fixed rates. This creates bandwidth slicing. Routes for flows are calculated using current bandwidth utilization as weight, and flows from the same class may use different paths. Although low delay traffic is categorized in its own class, the author does not monitor real-time delay and only uses current bandwidth utilization in the routing computation. This is based on an assumption that higher utilization (bandwidth and queue) equals to higher delay. Rather than flow level guarantees, HiQoS guarantees bandwidth per class level.

Another QoS framework called OpenQoS [8], focuses on end-to-end multimedia delivery. OpenQoS does not use priority queueing and resource reservation. Egilmez et al. argue that such technique has an adverse effect on non-QoS flows, as QoS flows are always prioritized over non-QoS flows. Instead, QoS control in OpenQoS is established using QoS routing. QoS flows (multimedia traffic) use dynamic routing (Constrained Shortest Path), while non-QoS flows (data traffic) always use shortest path routing algorithm. The authors extend their research in [32], in which they proposed a distributed QoS architecture for multimedia streaming.

Wallner and Cannistra [9] implement soft QoS using the Floodlight controller. The paper presents a proof of concept model for class-based QoS control. The model is similar to DiffServ. First, packets are classified based on their ToS/DSCP bits in the IP header. Then, egress traffic shaping is implemented in switches along the path using OpenFlow's Queue.

Celenlioglu [10] suggests using precomputed paths for QoS routing to improve scalability in a highly loaded network. Rather than being triggered by new flows, the routing algorithm is computed offline every few seconds. The precomputed paths are then used to install flow entries when there is a new flow request. Path resizing for load balancing also proposed in this research.

End-to-end bandwidth guarantees in OpenFlow

Tomovic et al. [11] propose an IntServ-like hard QoS guarantees for OpenFlow. The hard QoS is established using bandwidth reservation and admission control. A QoS flow begins by announcing its bandwidth requirement. The controller then performs a Constrained Shortest Path computation using the asked bandwidth as a constraint. The algorithm itself uses free bandwidth that is not reserved by other flows as link weight. This ensures that only the reserving flow can use the reserved bandwidth. If there is not enough bandwidth available to establish a path between source and destination, the flow is rejected. The controller also periodically checks all links used by the QoS flow. If the utilization is more than 80%, it reroutes best-effort flow to another path with lower utilization. This is done to prevent degradation of existing best-effort flow when new QoS flows arrives. In the switches, the bandwidth guarantees are enforced using OpenFlow's queue. For each flow, a queue is created in the ingress switch and intermediate switches with `min-rate` and `max-rate` equal to the guaranteed bandwidth.

A similar end-to-end guarantees model with per-flow bandwidth reservation and admission control is proposed in [31]. Here, QoS and non-QoS traffic are identified by their DSCP bits. On discovering a route for a flow, two flow entries (for QoS and non-QoS flow) are installed in switches at the same time, but directed via two different queues. Bandwidth reservation for a flow is performed in its ingress switch. How the bandwidth is guaranteed in other switches in the path is not mentioned in the paper. The goal of this research is to investigate resiliency of QoS in the case of link failure in both single and multiple domain networks.

In [12], Dwarakanathan et al. aim to solve the scalability problem of IntServ by combining it with the aggregation of Diffserv, particularly in a cloud environment. In the ingress switch's port (which is located inside a VM, and managed exclusively by a controller), a queue is created for all flows to reserve bandwidth. In the intermediate switches along the path, one queue per port is used to forward all QoS flows forwarded via this port. The rate of these queues is dynamic, with its initial value equal to zero. When a new flow arrives, the rate is increased by the rate of the new reserving bandwidth. By using only two queues per port (one for flows originated from the switch and one for transit flows), it is more simple compared to [11] which uses one queue per flow.

In both Tomovic's [11] and Dwarakanathan's [12] model, QoS flows are not allowed to exceed its guaranteed rate. In Tomovic's the max-rate is set for all queues, making sure that QoS flows will stay within the guaranteed rate. In Dwarakanathan's paper, it is not mentioned whether it uses a max-rate limiter for the queues or not. However, since only a single queue

is used for multiple QoS flows, QoS flows may contend with each other if there are QoS flows that send more than the guaranteed rate. A QoS flow with a lower rate than its guaranteed that happens to be in the same queue with QoS flows that exceed it might get penalized (the phenomenon is shown in section 6-1 of this thesis). In this case, the concept of rate guaranteeing itself becomes overridden. That is a problem that we want to address in this thesis. In this thesis, we allow QoS flows to send more than the guaranteed rate while still providing strict bandwidth guarantees for each individual QoS flow. Furthermore, the excess QoS traffic should not give adverse effect to the best-effort and conformant QoS traffic.

Chapter 3

QoS support in OpenFlow

This chapter provides background on QoS support in OpenFlow. Queue, an OpenFlow feature that is employed in this thesis, will be discussed in detail. The discussion is extended to Linux's Traffic Control (TC) application, which is used to implement OpenFlow queue in Open vSwitch. Within the same section, we also examine the Hierarchical Token Bucket algorithm. The last section of this chapter is discussing OpenFlow meter table.

3-1 QoS in the OpenFlow standard

OpenFlow has accommodated a notion of Quality of Service (QoS) since its earliest versions. However, the support was limited to a simple queueing mechanism with minimum-rate guarantee. It is further improved in OpenFlow 1.2 with the implementation of a queue with a maximum-rate limit mechanism. Later, in OpenFlow 1.3, similar rate-limiting functionality through meter tables was introduced.

OpenFlow switches also have the ability to read and write Type of Service (ToS) bits in an IP header. ToS is one of the *fields* that can be used to match a packet in a flow entry. The combination of these features allows network administration to apply QoS in their network.

3-2 Queue

OpenFlow's queue is an egress packet queuing mechanism in the OpenFlow switch port. Queue is first supported in OpenFlow 1.0, with a guaranteed *minimum rate* property. Later, it was extended in OpenFlow 1.2 with a *maximum rate* which limits the maximum throughput of a queue. Although it is specified in the OpenFlow switch specification, the OpenFlow protocol does not handle queue management. Queue management (creation, deletion, alteration) is handled by the switch configuration protocol, such as OF-Config or Open vSwitch Database (OVSDB). OpenFlow itself is only able to query queue statistics from the switch.

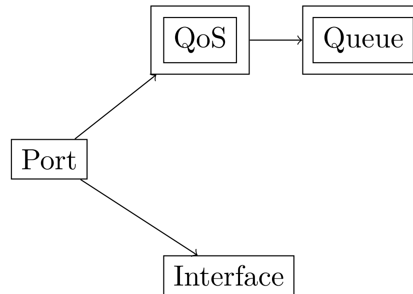


Figure 3-1: OVSDB database schema

Figure 3-1 shows the schema of Port table in OVSDB. Port table is related to interface table and QoS table. The relation to the interface table is mandatory, which means that all ports should have an interface. In contrast, the relation to QoS table is optional. A switch port might or might not have QoS settings attached to it. It can only have at most one configured QoS, while a QoS can have multiple queues.

A queue is used in the action field of a flow entry using the *set_queue* instruction. This instruction forwards packets that match the flow entry via the aforementioned queue. One single queue can be used to process several flows at the same time. In this case, the aggregate of actual throughput of those flows will be used in the queue's `max_rate` limitation and `min_rate` guarantee.

```
cookie=0x0, duration=50.164s, table=0, n_packets=51, n_bytes=4998,
idle_timeout=2, hard_timeout=1600, idle_age=0, priority=1,
ip,in_port=2,nw_src=10.0.0.1,nw_dst=10.0.0.3,nw_tos=64
actions=set_queue:4,output:3
```

The example above shows a flow entry with action forwarding to a queue 4. Any packets from 10.0.0.1 to 10.0.0.3 with ToS bits equal to 64 will be processed via queue 4, before forwarding to egress port 3.

The OpenFlow switch specification states the following properties for a queue:

1. **min_rate**. This property defines the guaranteed minimum data rate for a queue. If the `min_rate` property is set, the switch will prioritize this queue (and any flow forwarded via this queue) to achieved the mentioned minimum rate, at the cost of other flows' rates. If there are more than one queue in a port, with total `min_rate` higher than the capacity of the link, the rates of all those queues are penalized. The capacity is shared proportionally based on each queue's `min_rate`.

2. **max_rate**. Maximum data rate allowed for this queue. If the actual rate of flows using this queue is more than the specified `max_rate`, the switch will delay packets or drop them in order to satisfy the `max_rate`.

The OpenFlow specification only mentions these requirements for queue properties. How it is implemented in the switch is decided by the switch manufacturers themselves. Open vSwitch,

as a software switch based on Linux, implements OpenFlow queue with Linux's Traffic Control (TC) program.

3-2-1 Linux Traffic Control

Traffic Control (TC) [33] is a user-space utility program used to configure packet scheduling in the Linux kernel. This program is commonly used by network administrators to manage traffic entering and leaving servers or other network elements. In general, TC consists of several control mechanisms, i.e., shaping, scheduling, policing and dropping. The traffic processing itself is controlled with queuing discipline (qdisc), classes and filters.

TC supports both classless and classful queuing disciplines. There are two classful queuing disciplines that are used in OVS, i.e., Hierarchical Token Bucket (HTB) and Hierarchical Fair-Service Curve (HFSC). Both HTB and HFSC are hierarchical qdisc that allow bandwidth "borrowing". The main difference between the two is that HSFC balances delay-sensitive traffic against throughput sensitive traffic [34]. Since our model only guarantees bandwidth and not delay, HTB, as the first and more popular implementation, will be used in this thesis.

HTB

In the hierarchical token bucket algorithm, tokens are generated at a fixed rate, then stored in a fixed capacity "bucket". Packets can only be dequeued or sent to an output port if there is available token in the bucket. HTB is a queuing discipline that uses the concepts of multi-level token buckets to allow granular control over the outbound bandwidth on a given link. It is intended to be a replacement for Class Based Queueing (CBQ) which was a standard in older TC implementations.

Within an HTB instance, multiple classes may exists.

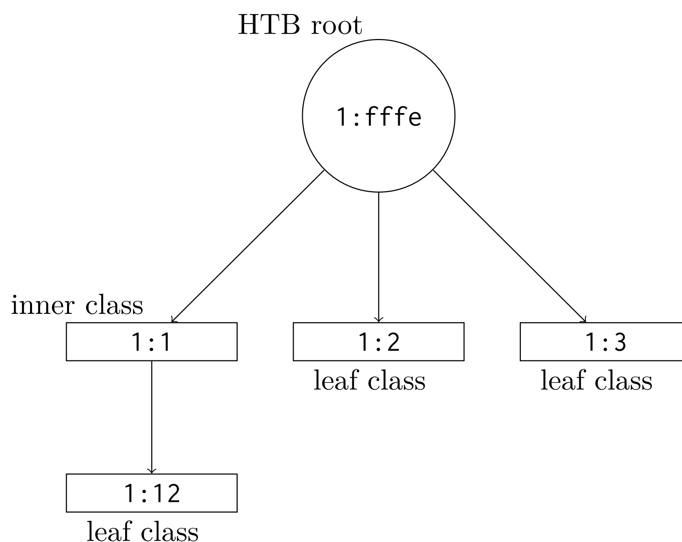


Figure 3-2: Example of HTB classes

Figure 3-2 shows an example of HTB classes arrangement. Class `1:fffe` is the HTB root class, it has three child classes: class `1:1`, `1:2` and `1:3`. Class `1:1` is another HTB class which has its own child, class `1:12`. A class that is not a parent to another class is known as a leaf class. Here, for each of the leaf classes, a First In First Out (FIFO) qdisc is attached.

In the OVS implementation of the OpenFlow Queue, a child class of a root can not have any children. So, there exist only two levels of hierarchy.

HTB classes have several important properties.

1. **`rate`**. Maximum guaranteed rate for this class and its children. It is equivalent to Committed Information Rate (CIR) [35].
2. **`ceil_rate`**. Maximum rate at which this class is allowed to send.
3. **`priority`**. Defines the priority of the class. Classes with higher priority (prio 0 has the highest priority) are offered idle bandwidth first. This prioritization should not affect other classes' guaranteed rate.

HTB extends the traffic shaping system of token bucket with a token borrowing model. Using this borrowing model, when a class uses less bandwidth than the amount assigned (`rate` property), the idle bandwidth is available for any other classes to use. It is important to note that the term "borrow" is not entirely accurate since the borrowing class does not have any obligation to return the resource that was borrowed.

When a child class' `rate` is exceeded, it is allowed to borrow tokens from its parent class until it reaches `ceil` (maximum allowable rate). When it reaches the `ceil`, the system will begin to queue packets. In the queue, packets will be dequeued (sent) if there are enough tokens, or dropped when the queue is full. In Figure 3-2, class `1:1`, `1:2` and `1:3` are allowed to "borrow" tokens from the root class, while class `1:12` can borrow from `1:1`.

3-2-2 Parallels between OVS's queue and TC classes

Queues are created using the ovs-vsctl command. This command creates and entry in OVSDB and then implements it in the switch using Linux TC.

```
hedi@node01:~$ ovs-vsctl set port eth1 qos=@newqos -- --id=@newqos create qos
type=linux-htb other-config:max-rate=10000000 queues:1=@newqueue1
queues:2=@newqueue2 -- --id=@newqueue1 create queue
other-config:min-rate=4000000 -- --id=@newqueue2 create queue
other-config:min-rate=4000000
```

The example above creates a "QoS" and "queues" in port eth1. In OVSDB, it is manifested as new entries in QoS table and Queue table. OVSDB then puts a relation between entry "eth1" in Port table and the newly created QoS entry. This relation indicates that eth1 should behave according to rules stated in this QoS.

During this process, the switch invokes the TC application to create qdisc and classes in the background. The creation of qdisc and its classes can be verified by running the following command in the switch's terminal.

```
hedi@node01:~$ tc -s class show dev eth1
class htb 1:fffe root rate 10000Kbit ceil 10000Kbit burst 1500b cburst 1500b
class htb 1:1 parent 1:fffe prio 0 rate 12000bit ceil 10000Kbit burst 1563b
cburst 1563b
class htb 1:2 parent 1:fffe prio 0 rate 4000Kbit ceil 10000Kbit burst 1564b
cburst 1563b
class htb 1:3 parent 1:fffe prio 0 rate 4000Kbit ceil 10000Kbit burst 1564b
cburst 1563b
```

The queues created with the ovs-vsctl command are actualized in the switch's port as TC classes. Queue 1 and queue 2 in OVSDB correspond to class 1:2 and 1:3 in TC. Class 1:1 is a default class which corresponds to queue 0 in OVSDB. When creating QoS with ovs-vsctl, queue 0 is always created although it is not explicitly mentioned. Queue 0 is also used to processed all flow entries without `set_queue` instruction in the flow entries.

In a similar fashion, we can also see that the max-rate attribute of QoS in the ovs-vsctl command becomes `ceil` for the HTB root class.

`burst` and `cburst` are the amount in bytes that can be burst at `ceil` speed and theoretical "infinite" speed respectively. Since in our model all `ceil` are set equal to the link capacity, and it is unlikely to have any classes to transmit more than the link capacity, we do not change these parameters from their default value.

All classes created with ovs-vsctl have a default `prio` equal to 0. It is possible to prioritize one class over the other class by changing this value. In the next chapter we will see how we prioritize best-effort traffic over excess QoS traffic in the forwarding plane by changing queue/class priority.

3-3 Meter table

Meter table is a new feature introduced in OpenFlow 1.3. Metering allows ingress rate monitoring of a flow and then perform operations based on the rate of the flow. Unlike queue, which is a property of a switch port, meter is attached to the flow entries.

A meter table consists of meter identifier, meter bands, and counter. The meter band specifies its `rate` and band `type`. If the flow's rate is higher than the specified band's `rate`, the operation specified in the band `type` will be performed to the flow.

There are two band types that define how packets are processed. The first type is "drop". This band type drops packets that exceed the specified in the band's rate. This operation is similar to queue min_rate. The second band type is "dscp remark". This band type increases the drop precedence of the DSCP field in the IP header of the packet. For example, if there is a 40 Mbps flows with ToS 64 processed via meter with 30 Mbps `rate` and `prec_level` equal

to 1, the ToS bits of excess 10 Mbps packets will be remarked to ToS 32. The other 30 Mbps packets will keep its ToS equal to 64.

Meter table is a complement to queue, which was developed earlier. There are several differences between the two.

1. OpenFlow queue can guarantee a minimum rate to a flow using traffic shaping and policing in switches. It is not possible to achieve this with meter table, as meter table only has maximum rate limiter.

2. Rather than just dropping packets like the queue's min_rate, meter can perform DSCP bits alteration. Thus, packets from a single flow might have different DSCP values, allowing different processes for packets from a single flow.

3. Queue can not be configured with OpenFlow channel. On the other hand, meter is can be installed, modified and removed at runtime using OpenFlow protocol.

While most OpenFlow hardware switches already support meter table, as of May 2016, CPqD's ofsoftswitch13 [24] is the only OpenFlow software switch that implements meter table.

Chapter 4

Enabling End-to-end Bandwidth Guarantees in OpenFlow

The goal of this thesis is to propose an OpenFlow controller design with an end-to-end bandwidth guaranteeing system. The bandwidth guarantees is accomplished in two levels. In the controller, admission and bandwidth reservation are performed to limit how many QoS flows can be admitted. In switches, a traffic shaping mechanism provides QoS traffic their guaranteed rate. In this chapter, all design considerations taken and methods we are using to achieve it are explained in detail. The end-to-end model used in this thesis is grounded on IntServ's Guaranteed Service, as our aim is to provide hard-QoS guarantees. Nevertheless, some legacy problems that commonly associated with IntServ are avoided in our design. In this chapter, we also propose a new approach in traffic prioritizing by giving more priority to best-effort when contending for bandwidth with excess QoS traffic.

4-1 Assumptions

IntServ's Guaranteed Service, proposed in RFC 2212 [36], guarantees both delay and bandwidth. In QoS frameworks, delay guarantees (as well as other QoS metrics like jitter, loss and minimum bandwidth) are provided using QoS routing, which selects a path that can meet such requirements [19]. Currently, OpenFlow does not have native support to measure throughput and delay. While it is easy to measure current flow throughput by comparing flow statistics from time to time, measuring delay is more complicated. One study by van Adrichem et al. [37] shows how this can be achieved in an efficient manner. To measure delay, the controller needs to insert probing packets into the network and measure its Round Trip Time (RTT). Recreating this monitoring function in our controller design requires much work and is out of the scope of this thesis. Therefore, we decided to focus only on bandwidth guarantees. Bandwidth is the key component for offering QoS, without which many services cannot be delivered [1].

The controller is designed using the Ryu OpenFlow controller [38]. Ryu was started in NTT (Nippon Telegraph and Telephone), and is now freely available under Apache 2.0 license. Ryu, as well as its controller applications, is fully written in Python. The application written for this thesis uses the OpenFlow 1.3 standard. Topology discovery is performed using built-in functionality in Ryu by activating the "-observe-links" flag. This functionality uses LLDP (Link Layer Discovery Protocol) to learn links between switches and report it to the controller.

OpenFlow is a southbound interface standard, managing SDN communication between controller and switch/router. Queue implementation and supporting protocols (such as OVSDB) may be different from one switch to another. Our model is created for and tested with the OVS software switch.

Additionally, our model and all simulations performed in this thesis are based on an assumption that the system works without any failure in the links.

4-2 Traffic types

In our model, there are two kinds of flows: best-effort flows and QoS flows. A best-effort flow is a flow without reservation and bandwidth guarantees, while a QoS flow is a flow with guaranteed bandwidth, analogous to the Guaranteed Service. The system will try to accommodate QoS flow to achieve its guaranteed throughput. This is accomplished by using admission control and bandwidth reservation (in the controller), and rate guaranteeing (in the switches).

Allowing excess QoS traffic

QoS flows are allowed to send packets with data rate more than the guaranteed bandwidth. In this case, non-conformant traffic violating the contract are called "excess traffic". In the network, excess traffic is considered as a different type of traffic, and will get different treatment from QoS traffic that stays within its guaranteed rate.

Using an analogy from Frame Relay, guaranteed (conformant) traffic is CIR, excess traffic is Excess Information Rate (EIR), while the actual data rate sent is Peak Information Rate (PIR). The relation is given by $PIR = CIR + EIR$.

From an economic point of view, allowing QoS flow to send more than what is allowed in the contract is beneficial for both network provider and its customer. The customer gets a higher throughput than the contract, while the network provider might charge a small amount of fee to permit excess traffic without any guarantee. From a network perspective, higher throughput enables data transfer process to finish sooner. When the flow is over, the bandwidth is free to be used for other flows. This is particularly important in a data center, especially in big data applications, where data of large size are moved from server to server.

In IntServ's Guaranteed Service, excess (non-conformant) QoS packets are treated as best-effort packets. In this case, both types of packets have the same priority. When the network is congested, they will compete directly for resources. As excess QoS traffic is an additional service given to customers with little importance, we believe that it should not hurt best-effort traffic's performance. Therefore, in this thesis, we propose a differentiation between

best-effort traffic and excess QoS traffic. Best-effort traffic gets higher priority than excess QoS traffic.

4-3 Per-flow, end-to-end bandwidth guarantees

Real bandwidth guarantees can only be achieved on a per-flow level. A class aggregation model such as DiffServ does not provide a strong bandwidth guarantee since it forces flows to share resources. In a case of congestion, these flows will have to compete for the available resources. In our model, bandwidth is guaranteed per flow by using reservation, similar to IntServ. Nevertheless, some modifications are made; described in the following.

Idle reserved bandwidth can be used by other flows

A QoS flow reserves a certain amount of bandwidth in all links along the path between source and destination switch. In IntServ, reserved bandwidth can only be used by the reserving flow. The reservation potentially causes low bandwidth utilization if the actual data rate is less than the reserved bandwidth. This problem is described in [39] and [40].

According to a study by Rao [41], one of the most common strategies employed in video streaming is *ON-OFF cycles*. During the ON period, the client downloads as fast as the network allows. It is then followed by an OFF period, in which data transfer rates are much smaller. This cycle ensures that the client buffer is not overloaded by the amount of data transferred by the server. Similarly, Big Data applications tend to be bursty and varied. Exclusive use of bandwidth for such traffic potentially leads to low link utilization.

The usage of OpenFlow's Queue for rate guarantee can solve this problem. In OVS, the queue is based on HTB Linux. HTB uses hierarchical token buckets that allow one class' idle bandwidth to be used by other classes. Thus, if no limited, best-effort traffic and excess QoS traffic can "fill" the idle-reserved-bandwidth of a QoS flow.

Hard reservation state

One of the drawbacks of RSVP is that its QoS state is soft-state. PATH and RESV messages require periodic refresh by sender and receiver with a typical interval of 30 seconds. If these messages are not received, the end hosts will assume that the connection has ended. Reserved resources are then will be released. The periodic refreshment can lead to an enormous number of signaling packets, especially when there is a large number of flows.

In our model, we use hard-state reservation. QoS reservation begins when there is a QoS flow request to the controller. The request is signaled with the `OFPT_PACKET_IN` message with appropriate ToS bits. The resources are freed from reservation when the flow entry is removed from the switches, signaled by the `OFPT_FLOW_REMOVED` message. Flow removal messages are sent by switches to the controller after a flow reaches its lifetime limit, defined by `idle_timeout` or `hard_timeout`. Both `OFPT_PACKET_IN` and `OFPT_FLOW_REMOVED` are standard OpenFlow messages. There is no extra signaling exchanged between the switch and the controller.

Hard state reservation significantly reduces signaling, especially in a network with many flows. The number of reservation signals s required in the hard-state with f number of flows and n average switches on the path is $s = 2 \times f \times n$. While for the soft-state, the number of signals is $s = 1 + (t/r) \times f \times n$, with t the average flow lifetime and r the refresh interval.

Figure 4-1 shows the comparison of signaling messages per switch required by soft state and hard state reservation, assuming RSVP's default state refresh rate of 30s [6]. The four curves in the graph show the number of signaling messages required for hard state and soft state with a lifetime of 30s, 60s, and 450s. For the short-lived flows (equal or less than refresh interval), the soft state system sends state signals twice, one in the beginning (Path) and one at the end of the flow (PathTear). This is equal to the number of signals used in the hard state system. But for long-lived flows, soft signals uses much more signals than hard state system. The hard-state uses the same number of signaling messages no matter how long the flow live.

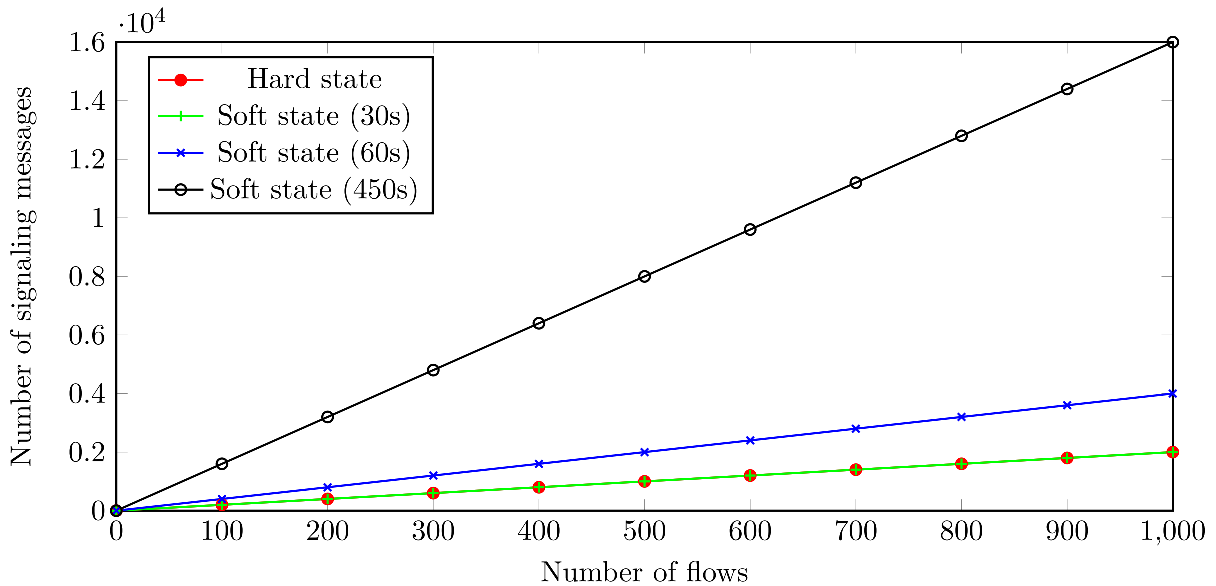


Figure 4-1: Comparison of reservation signals required for various flow lifetime

4-4 Routing, admission, and reservation

Route computation for QoS flows in our model is performed using the Widest-Shortest Path (WSP) algorithm. The algorithm is based on a modified Dijkstra algorithm, proposed by Ma and Steenkiste [42]. This algorithm selects the shortest path between two nodes. If there is more than one path candidate, it will select the one with the biggest bandwidth.

Two network graphs are used in WSP algorithm. The first graph is adjacency matrix between switches in the network, representing shortest path between nodes. The second graph stores information about reservable bandwidth in links. Reservable bandwidth is a portion of the bandwidth that is not currently reserved by other QoS flows; thus, available to be used by

incoming QoS flow. In both graphs, if the available bandwidth of a link is less than the requested bandwidth, the link is pruned.

WSP uses hop count as primary weight and Minimum Reservable Bandwidth (MRB) as secondary weight. For a given path p, MRB is the minimum of the reservable bandwidth of all links in the path, formally stated as,

$$MRB_p = \min \left\{ R_{i_j} | i_j \in p \right\}$$

When selecting a node candidate to mark, the node with minimum hop count is used. If there are several candidates with the same minimum hop count, the one with largest MRB is selected. The algorithm is illustrated in Algorithm 4-4.1.

Algorithm 4-4.1 Widest Shortest Path Algorithm [42]

1: function Dijkstra(Adjacency graph G, Bandwidth graph B, source s, destination d):
2: Prune links if available bw less than requested bw
3: **for** each vertex v in G
4: $\quad dist[v] \leftarrow \infty$
5: $\quad cap[v] \leftarrow 0$
6: $\quad prev[v] \leftarrow \emptyset$
7: $\quad$ add v to Q
8: $dist[s] = 0$
9: $cap[s] = \infty$
10: **while** Q is not empty
11: $\quad x \leftarrow$ vertex in Q with min $dist[x]$
12: $\quad$ **if** $|X| > 1$ \\if there are more than one candidate
13: $\qquad x \leftarrow$ vertex in Q with max $cap[x]$ \\use link with bigger capacity
14: $\quad$ remove x from Q
15: $\quad$ **for** each neighbour y of x
16: $\qquad altdist \leftarrow dist[x] + G[x, y]$
17: $\qquad altcap \leftarrow min(cap[x], B[x, y]$
18: $\qquad$ **if** $alt > dist[y]$
19: $\qquad\quad dist[y] \leftarrow altdist$
20: $\qquad\quad cap[y] \leftarrow altcap$
21: $\qquad\quad prev[y] \leftarrow x$
22: **if** $prev[d] = \emptyset$
23: $\quad$ Not enough bandwidth between s and d. Return.
24: **else**
25: $\quad$ return $prev[]$

To ensure bandwidth availability, admission control is performed when a QoS flow request arrives. The admission process is directly related to the routing computation. When the routing computation does not find a route from source to destination, it signals the admission system that there is not enough bandwidth to satisfy this QoS flow. The admission system then installs a flow entry in the originating switch to drop subsequent packets that match this particular flow.

In a successful QoS flow request, the reservation system reserves a certain amount of bandwidth requested by the flow. The reservation is made by updating the reservable bandwidth graph in all links on the path between sender and receiver. By doing this, when the next QoS is coming, the system will know how much bandwidth is allowed to be used. The controller also maintains a flow database, in which it keeps track of how many flows currently exist in the system, and how much bandwidth each of them is reserving. Compared to IntServ, where reservation states are stored in switches, storing reservation state in the controller is more effective and efficient. Database design and database machines in which the states are stored can be easily optimized to accommodate a high number of flows.

On the contrary, best-effort flow is not commenced with an admission process and uses Dijkstra's shortest path algorithm. The shortest path algorithm ensures best-effort flows to use minimum resources in the network.

4-5 Queue management

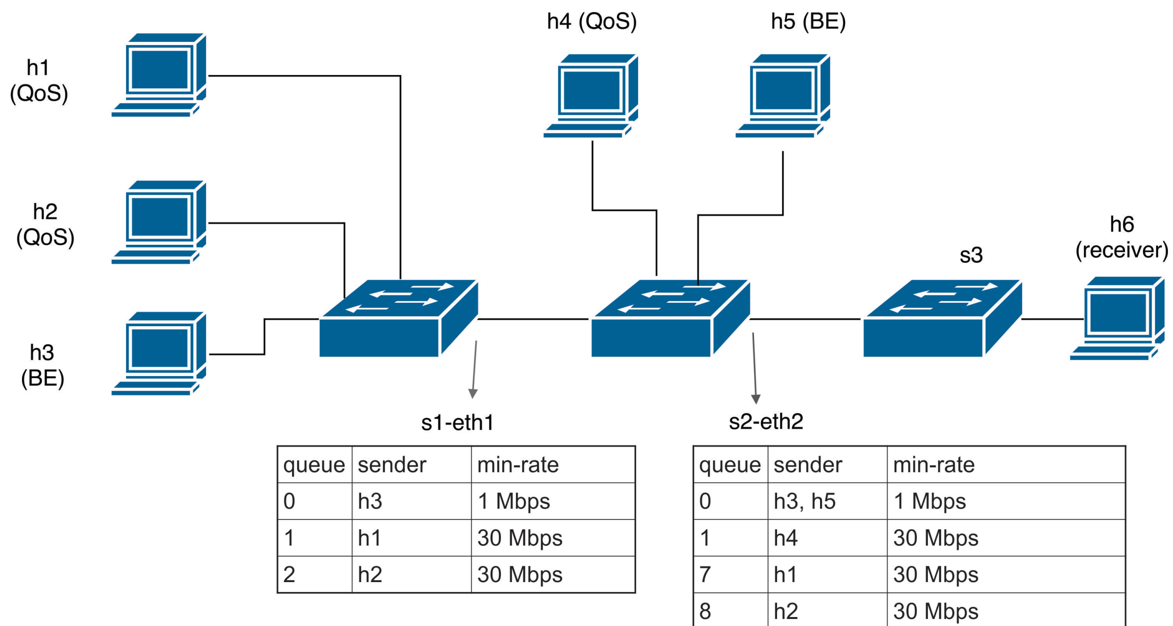
As mentioned in chapter 3, OpenFlow's queue provides traffic shaping mechanism that we use to guarantee QoS flow rate. All packets in the network are forwarded via queues before being sent to the egress port of a switch. In our model, we categorize the queue into three types. Each of these queues is used to forward a particular type of traffic.

1. Source QoS queue
   For each QoS flow, a queue is created in the egress port of the source switch (switch connected to sender host). The flow is forwarded via this queue to provide bandwidth guarantees. Min-rate of this queue is set to be equal to the guaranteed bandwidth. While the max-rate is equal to link capacity. This setup ensures the flow to achieve the guaranteed rate, even when the links are congested. If the link is free, unused by other flows, it is allowed to send packets as fast as the link capacity.

2. Intermediate QoS queue
   Intermediate switches are switches between source switch and destination switch. A flow with five switches between end hosts, has three intermediate switches. Similar to the source QoS queue, each QoS flow transiting in a switch's port is forwarded via individual intermediate queue.

3. Best-effort queue
   Queue 0 of all switch ports is designated to be used for best-effort flows. All best-effort flows are forwarded through queue 0 in both originating switch and intermediate switches. Min-rate for this flow is set very low, so the best-effort flows will not competing for bandwidth with QoS flows.

The source queue arrangement is different from single queue model presented in [12]. In single queue model, a single QoS queue is used for all QoS flows. With such arrangement, if there is a QoS flow A sending more than its guaranteed rate, the excess traffic will contend with QoS flows B originating from the same switch. Flow B's rate cannot be guaranteed as the queue will see both flow A and flow B as a single entity. This is of course, unfair for flow B.



queue	sender	min-rate
0	h3	1 Mbps
1	h1	30 Mbps
2	h2	30 Mbps

queue	sender	min-rate
0	h3, h5	1 Mbps
1	h4	30 Mbps
7	h1	30 Mbps
8	h2	30 Mbps

Figure 4-2: Example of queue arrangement

Figure 4-2 shows an illustration of queue arrangement. In this network, hosts h1-h5 send packets to h6. Hosts h1-h3 are connected to switch s1, h4-h5 connected to s2, while h6 is connected to s3. The network is a linear topology, the two links between switches are shared by the flows. In port s1-eth1, two *source QoS queues* are used to forward QoS flows. Queue 1 is used by flow h1-h6, while queue 2 is used by flow h2-h6. In the second switch, these flows are forwarded via queue 7 and queue 8, respectively. The min-rate of these queues are equal to the guaranteed bandwidth; in this case, 30 Mbps.

In both s1-eth1 and s2-eth2, queue 0 is used to forward best-effort flows. The min-rate of queue 0 is 1 Mbps, so it virtually has no guaranteed rate. Port s3-eth3, which is connected to receiver h6, does not have any queue created on it. All the flows are forwarded without `set_queue` action. The reason is that the traffic is already shaped in the previous switch ports.

Max_rate of all queues in all switch ports is set to be equal to the capacity of the link. This will allow any flows to send as much as possible when links are idle.

In our model, the queues are created when a switch joins the network. The queues are created when a switch joins the network. After exchanging hello message, the controller sends an ovs-vsctl command to create QoS and Queue table in the switch's OVSDB. OVSDB supports maximum 4,294,967,295 queues entry in the database [27]; however, due to a limitation of Linux TC, only 65,535 queues can be created in a single port.

4-6 Prioritizing best-effort traffic over excess QoS

By default, all queues/classes have the same priority. It means that non-conformant QoS traffic and best-effort traffic are competing for bandwidth that is not reserved by QoS flows.

Since we want to offer idle bandwidth to best-effort traffic first, before offering it to excess QoS traffic, we need to change its priority in the Linux TC application. This is done by changing the `other-setting:priority` in the queue.

QoS traffic is given priority equal to 1, while best-effort priority is 0 (queue with smaller priority number has higher priority). With this setting, the bandwidth will be offered to QoS flow first to satisfy its minimum guaranteed `rate`. After the QoS flow achieves its `rate` (conformant traffic), the idle bandwidth is then offered to best-effort flows. If there is still bandwidth left unused, it is offered to excess traffic from QoS flow.

4-7 General program flow

Figure 4-3 shows the complete program flow of our model. When a new flow arrives at a switch, the first packet is sent to the controller. This `packet_in` event triggers the admission process in the controller. After the arrival of this packet, the controller checks ToS bits of the packet to determine its traffic type. ToS equal to 0 means that it is a best-effort flow. There is no admission process for best-effort flows; the controller simply computes a path with shortest path algorithm and installs flow entries in switches along the path. In all of these switches, the flow is processed through queue 0.

If ToS bits are not equal to 0, the flow is a QoS flow. The controller checks the database to determine how much bandwidth B it requires according to contract. It then computes a route for this packet with the widest-shortest path algorithm. Based on the current network graph, the algorithm decides whether there is enough available bandwidth in the network.

The are two possible results of the algorithm. If the network does not have enough bandwidth, the flow is rejected. The controller then installs a flow entry in the ingress switch to drop subsequent packets of this flow. If there is enough bandwidth, the algorithm returns a path for this flow. The controller checks for an unused queue in the egress port of the first switch in the path to look for an idle queue. Idle queue q, which is not currently used to forward any other QoS flows, is selected to forward this flow. The controller then installs flow entries in switches. The same process also conducted for intermediate queues.

The last step is refreshing the network bandwidth graph and storing reservation in the flow database. In the graph, the available bandwidth in all links in the path is decreased by the requested bandwidth. The graph will be used for the next QoS flow path computation.

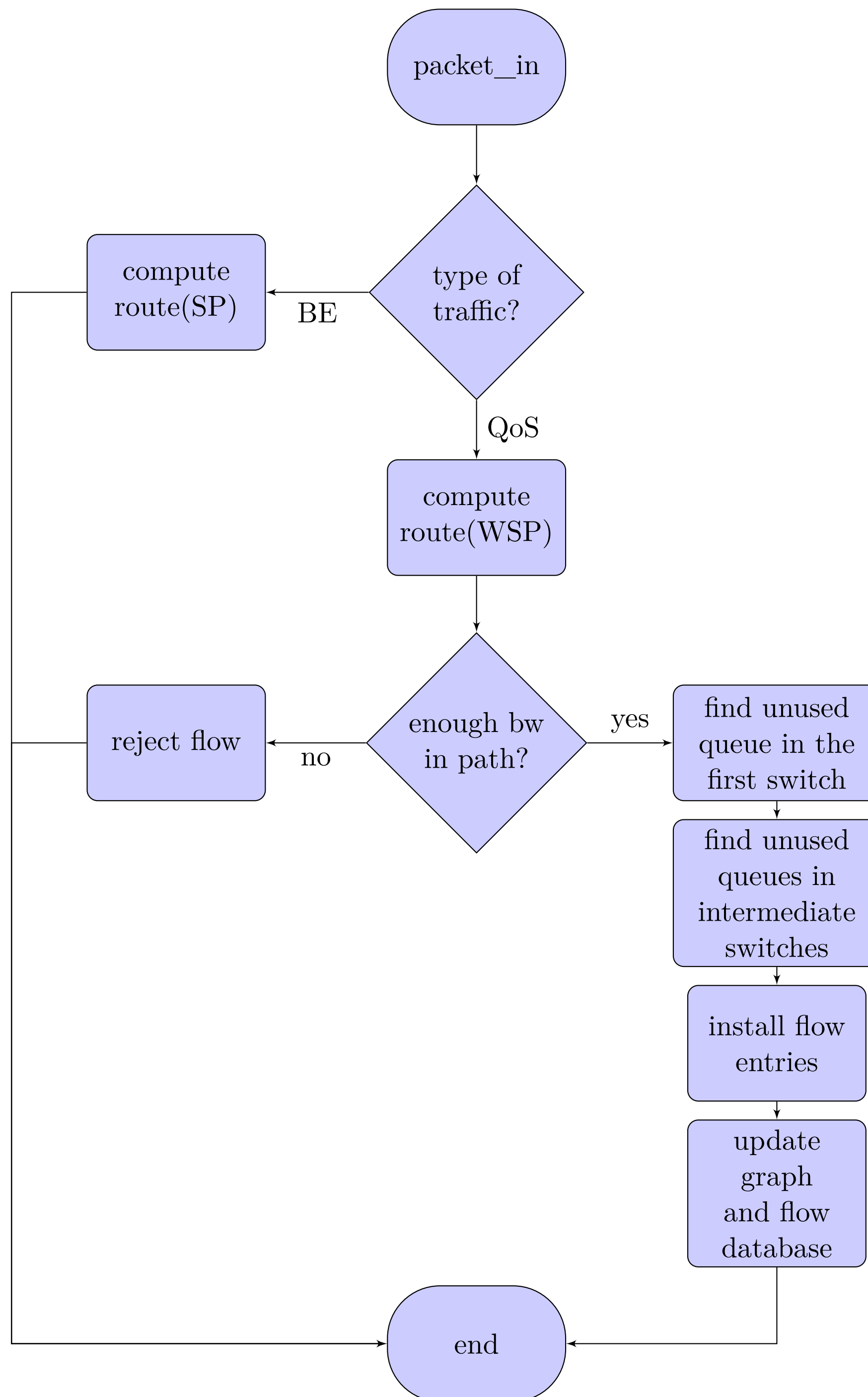


Figure 4-3: General flow of the system

Chapter 5

Proof of Concept

The two fundamental aspects of our model in this thesis, the admission-reservation process and class prioritization, are demonstrated in this chapter. First, an experiment is conducted to see how admission control and bandwidth reservation are working hand-in-hand to ensure bandwidth guarantees. In the next section, we will see how class prioritization works and how it increases link utilization. The experiments also demonstrate that our controller design is working as expected.

5-1 Admission and Reservation

As already discussed in the previous chapter, the end-to-end bandwidth guarantees in our model is established using bandwidth reservation in every link in the path between sender and receiver. The bandwidth admission and bandwidth reservation are performed in the controller for newly arrived QoS flows. On the contrary, best-effort flows do not require an admission process. To demonstrate this concept, an experiment is performed. Figure 5-3 depicts the topology for the experiment.

5-1-1 Experiment setup

The network consists of three switches in a ring topology and eight hosts. Both switch s1 and s2 are connected to four hosts. Links via s3 provide an alternative path between s1 and s2. Each host in s1 is paired with a host in s2 for traffic generation. The traffic is bidirectional between these host pairs.

The switches are Open vSwitch software switches installed in TU Delft NAS group's network testbed. Each switch is run on a server with Quad Intel(R) Xeon(TM) CPU 3.00GHz processor and 4 GB memory. The Open vSwitch used is OVS version 2.3.2, supporting OpenFlow 1.3. All links have a bandwidth capacity of 100 Mbps.

There are two kinds of flows generated for this experiment, best-effort (BE) and QoS. QoS flows have bandwidth requirements of 70 Mbps. If there is no path between two host pairs that

Table 5-1: Traffic generated for admission test

Host pairs	Type	Guaranteed bandwidth	Actual traffic from sender
h1 - h5	QoS	70 Mbps	70 Mbps
h2 - h6	QoS	70 Mbps	70 Mbps
h3 - h7	QoS	70 Mbps	20 Mbps
h4 - h8	BE	N/A	20 Mbps

meets this requirement, the flow is rejected. Best-effort and QoS packets are differentiated by their Type of Service (ToS) bits. The flow is identified by a matching field of three tuples, i.e. IP source address, IP destination address and IP ToS bits.

Three pairs of hosts (h1-h5, h2-h6, h3-h7) generate QoS traffic, while h4-h8 generate best-effort traffic. Although both ends of the flow are sending and receiving packets, for the sake of clarity, the initiating host is referred to as "sender", while its paired host is referred to as "receiver". All sender hosts are connected to switch s1, while receiver hosts are connected to switch s2.

The traffic used in the experiment is UDP, generated with iperf [43]. Table 5-1 shows the actual rate of traffic generated by sender hosts.

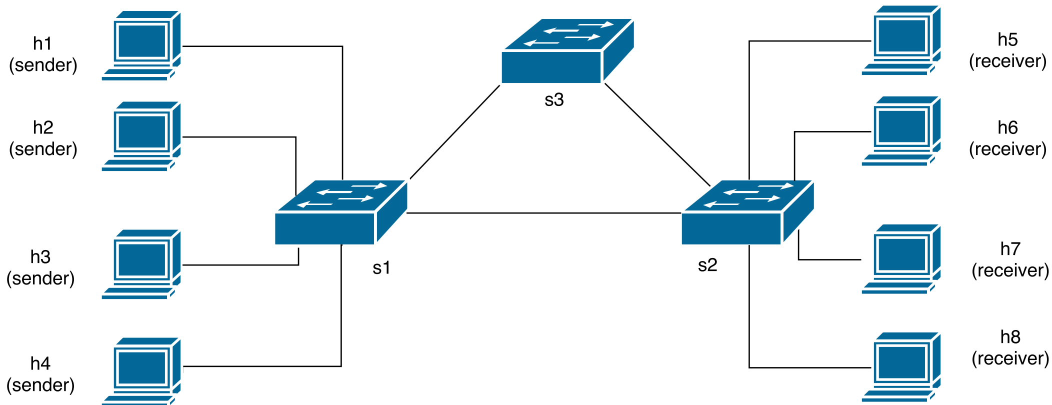


Figure 5-1: Topology for admission test

5-1-2 Experiment result

Figure 5-2 shows the result of the experiment. The left graph is the rate of traffic generated in the sender hosts. The graph on the right is throughput captured in the receiver hosts.

The controller keeps track of how much bandwidth is available and how much is reserved by QoS flows. In the system, this is represented by network graph metric, with a value equal to available bandwidth (in Mbps). Metric in this graph is used in route computation and is updated in the event of successful QoS flow request and QoS flow removal.

In the beginning of the experiment, the links are free from reservation. All links' weights in the network graph are equal to bandwidth capacity. The initial graph is as follows:

```
graph = {1: {1: inf, 2: 100, 3: 100}, 2: {1: 100, 2: inf, 3: 100}, 3: {1:
100, 2: 100, 3: inf}}
```

In this particular experiment, since the traffic is bidirectional with the same rate, link weight from host A to host B is always equal to weight from host B to host A. Weight from a host to itself is infinite; on the contrary, link from a host to another host without direct link is equal to 0.

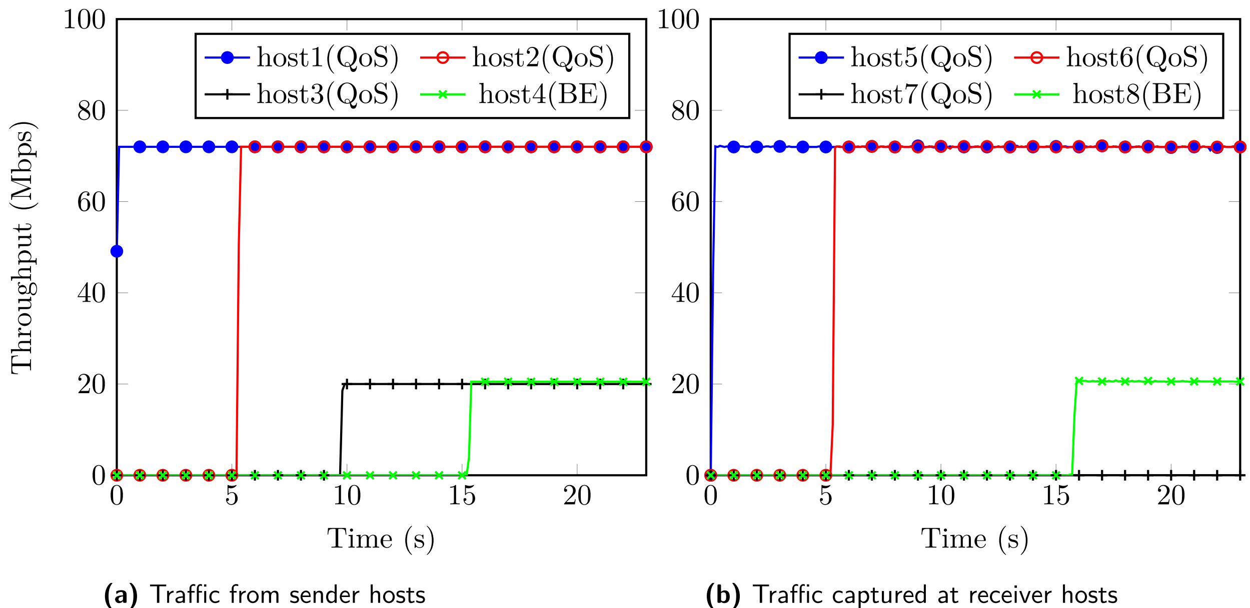
The experiment begins at time $t = 0$ when host h1 sends 70 Mbps QoS traffic to h5. Since both paths have the same amount of available bandwidth, this flow is routed using the shortest path (link s1-s2). The system reserves 70 Mbps in this link for flow h1-h5, and then updates the graph weight. The new weight is 30 Mbps, equal to the currently available bandwidth.

```
graph = {1: {1: inf, 2: 30, 3: 100}, 2: {1: 30, 2: inf, 3: 100}, 3: {1:
100, 2: 100, 3: inf}}
```

At time $t = 5$, host h2 starts QoS flow to h6. The system routes this flow using the updated graph shown above. Link s1-s2 no longer has enough available bandwidth to accommodate the new QoS flow because it is less than 70 Mbps. Thus, flow h3-h5 is routed through the alternative path (s1-s3-s2). After that, the graph is updated into the following.

```
graph = {1: {1: inf, 2: 30, 3: 30}, 2: {1: 30, 2: inf, 3: 30}, 3: {1: 30,
2: 30, 3: inf}}
```

Host h3 starts QoS flow to h5 at time $t = 10$. The throughput of this flow is 20 Mbps, which is less than the available bandwidth in either path. However, as QoS flow, it requires 70 Mbps available bandwidth in the path. Since no paths between sender and receiver can satisfy this requirement, the flow is blocked by the admission control system. Switch s1 installs a flow to drop this packet and subsequent packets that match this source-destination-ToS tuple.



(a) Traffic from sender hosts

(b) Traffic captured at receiver hosts

Figure 5-2: Admission-reservation test result

On the contrary, a 20 Mbps best-effort flow from h4 to h6 (starts at $t = 15$) is delivered via link s1-s2 (shortest path). This flow is allowed because as a best-effort flow it does not have

specific bandwidth requirement. Available bandwidth in this path is 30 Mbps, less than this flow's throughput; therefore, the traffic is received in h6 with a full rate of 20 Mbps. This flow is routed via the shortest path s1-s2.

From the experiment, we see that the admission process ensures QoS flows not to compete with each other. Therefore, guaranteed bandwidth is provided at the controller level. On the other hand, best-effort flows bypass the admission process and excess QoS traffic is also allowed. To ensure that these traffic do not disturb the conformant QoS traffic, traffic prioritization is used.

5-2 Traffic Prioritization

As discussed in the previous chapter, OpenFlow Queue in OVS in established using Linux's TC application, which allows us to differentiate class priority. Class/queue prioritization is employed in our model to allow unused reserved bandwidth to be used by other flows, while still give a firm guarantee to the conformant QoS traffic. In this section, this concept is demonstrated with an experiment.

5-2-1 Simulation environment

The simulation uses a network topology of three switches in a line topology and eight hosts, illustrated in Figure 5-3. Hosts h5-h8 act as receivers, to which hosts h1-h4 send their traffic. Hosts h1 and h5 form a best-effort traffic pair, while the other hosts use QoS traffic. All the links have a bandwidth of 100 Mbps. Table 5-2 summarizes the traffic.

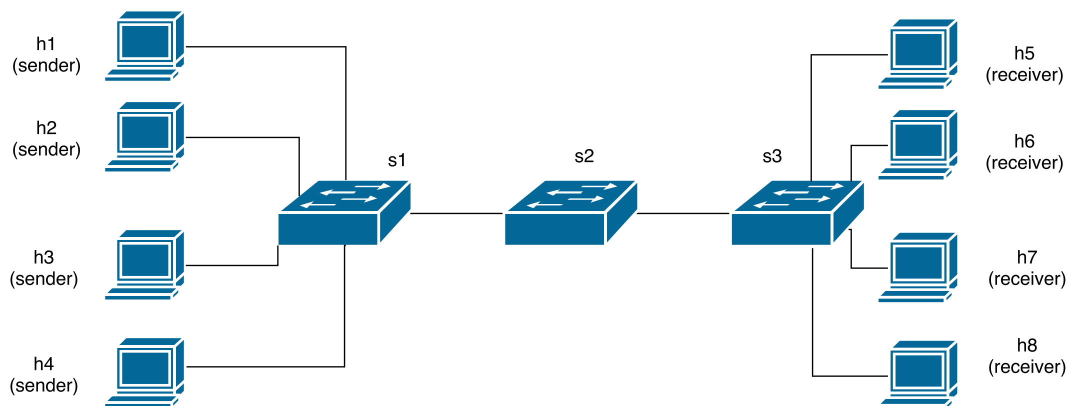


Figure 5-3: Topology for prioritization test

Each QoS flow has guaranteed bandwidth of 30 Mbps. QoS flows are allowed to send more than their guaranteed bandwidth. In the simulation, h2 and h3 send 80 and 50 Mbps respectively during the whole period of flow. Host h4 sends 20 Mbps at time $t = 15$ to $t = 25$, and then increases to 30 Mbps for the next 10 seconds. Traffic from host h1 to h5 is best-effort, with a rate of 20 Mbps. The `min_rate` for all QoS queues are set to 30 Mbps, equal to the

Table 5-2: Traffic generated for prioritization test

Sender	Receiver	Traffic type	Guaranteed bandwidth	Actual traffic	Queue id	
					s1-eth1	s2-eth1
h1	h5	BE	N/A	20 Mbps	0	0
h2	h6	QoS	30 Mbps	80 Mbps	1	1
h3	h7	QoS	30 Mbps	50 Mbps	2	2
h4	h8	QoS	30 Mbps	20-30 Mbps	3	3

guaranteed rate. For the best-effort queue it is equal to 1 Mbps; so the system will only guarantee a relatively low rate compared to QoS flow

Prioritization is established by changing the priority setting of queues. Queue 0 is given priority equal to 0 (highest priority). For QoS flows, all queues priorities (*source* and *intermediate*) are equal to 1. With this arrangement, a switch will first try to satisfy the guaranteed rate of QoS flows. The remaining bandwidth is then offered to best-effort flows since it has higher priority value. If there is still bandwidth left unused in the link, it will be used by excess traffic of QoS flows.

5-2-2 Experiment Result

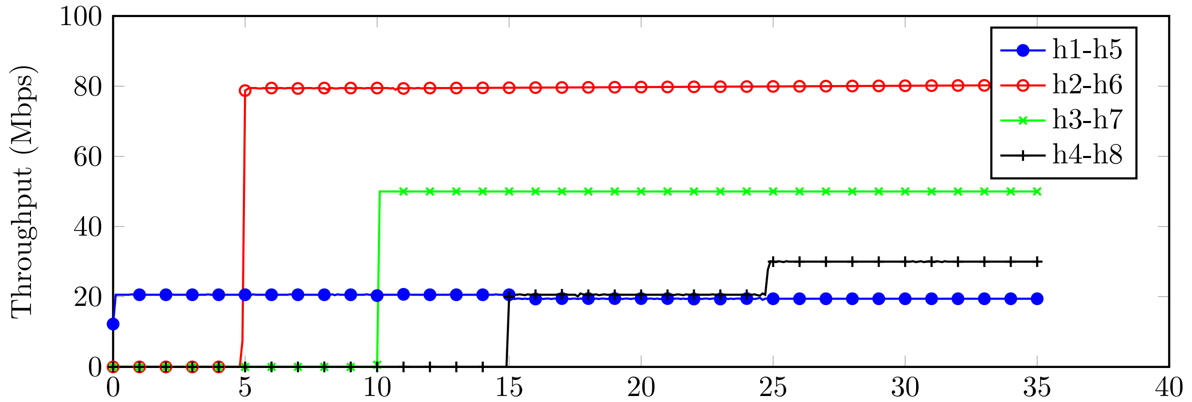
Figure 5-4 shows the result of the experiment. The graph is analyzed second-by-second as follows:

1. **0s to 5s**. At time $t = 0$, h1 starts a best-effort flow to h5 with a rate of 20 Mbps. At the receiving host h5, the flow throughput is measured at 20 Mbps.

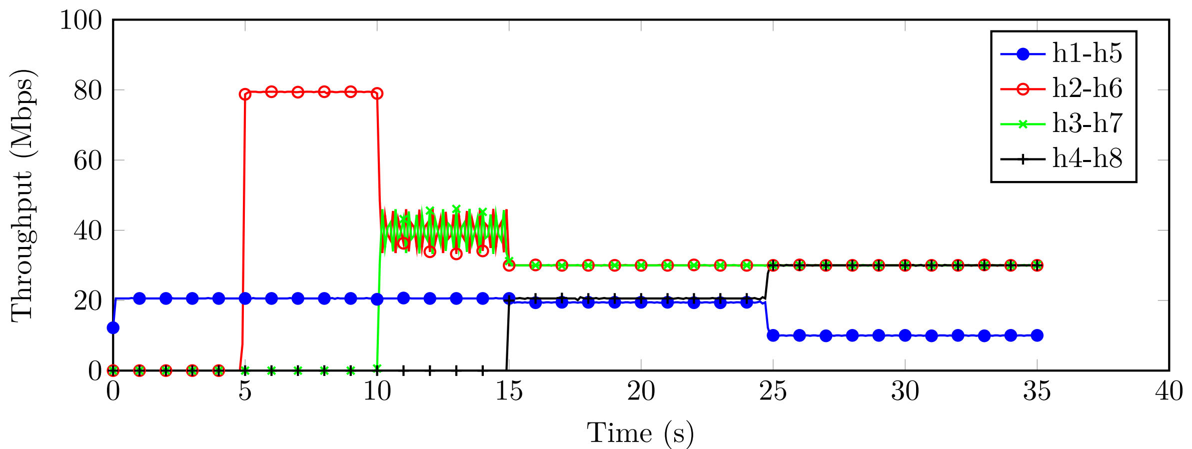
2. **5s to 10s**. Host h2 starts its QoS flow to h6 with a throughput of 80 Mbps. Although only 30 Mbps are guaranteed, it is allowed to send more. The excess traffic is served as long as link bandwidth is unused. At $t = 5$, there is exactly 80 Mbps free bandwidth (100 Mbps capacity minus 20 Mbps used by h1-h5 flow); therefore, no packets are dropped, and h6 receives this flow with a throughput of 80 Mbps.

3. **10s to 15s**. Host h3 sends QoS flow of 80 Mbps. The bandwidths of the links are now insufficient to accommodate all flows at their full rate. The system tries to accommodate QoS flows first by using 60 Mbps for h2 and h3. The bandwidth is then offered to the best-effort flow from h1, which rate is 20 Mbps. The leftover bandwidth of 20 Mbps is distributed equally to h4 and h5. In the end, the QoS flows get an average of 40 Mbps (alternating between 30 and 50 Mbps, because of round robin scheduling when offering the leftover bandwidth to source QoS queues). The best-effort flow is still received at 20 Mbps because it has higher priority than QoS flows.

4. **15s to 25s**. At $t = 15$, there is 40 Mbps available bandwidth (not reserved by QoS flows) in the links. However, the links are fully used by two QoS flows (2 x 40 Mbps) and a best-effort flow (20 Mbps). When QoS flow from h4 to h8 starts (20 Mbps) at $t = 15$, traffic shaping in port s1-eth1 is working to accommodate the newly arrived



(a) Data rate generated at sender hosts



(b) Actual throughput at receiving hosts

Figure 5-4: Prioritization test result

QoS flow. The switch drops excess traffic from QoS flows h2-h6 and h3-h7. Each of them only gets 30 Mbps.

5. **25s to 35s**. At $t = 25$, flow h4-h8 increases its throughput from 20 Mbps to 30 Mbps. Switch s1 shapes traffic to ensure all QoS flows got the guaranteed 30 Mbps. By doing this, it shapes the best-effort traffic (h1-h5) traffic to 10 Mbps.

From the example above, we can see how traffic shaping works in Open vSwitch. The property of OpenFlow queue's `min-rate` and `priority` are exploited to differentiate QoS flows into conformant and non-conformant traffic, and give them different priorities.

Chapter 6

System test

In the previous chapter, a proof of concept for our controller design is presented. In this chapter, more simulations are performed to see the how the system works in comparison with other hard-QoS models and to investigate the goodness of the system in general.

First, we compare our bandwidth guaranteed with the model presented in [12]. We choose this paper as it has a similar underlying Hard-QoS concept with our model. In the next section, a simulation is conducted to measure how bandwidth borrowing mechanism affects QoS flow's packet loss. In the third experiment a large scale experiment with thousands of flows is conducted to see how the class prioritization limits excess (non-conformant) QoS and increases the best-effort traffic rate.

6-1 Bandwidth guaranteeing in single and multiple queues system

As mentioned in chapter 2, the single queue model presented in [12] does not allow QoS flows to send more than the guaranteed rate. In such model, all QoS flows are forwarded via single queue model. On the contrary, in our model, each QoS flow is forwarded via individual queue. In this section, an experiment is conducted to see how excess QoS traffic might override bandwidth guaranteeing concept in a single queue model, and how our model can solve this problem.

For this experiment, a network shown in Figure 6-1 is used. Hosts h1 and h2 send QoS traffic to h4 and h5 with guaranteed rate of 30 Mbps. The actual rate sent by h1 and h2 are 50 Mbps and 20 Mbps respectively. Host h3 generates 90 Mbps best-effort traffic to h6. All flows are generated with iperf. The received throughput is observed from iperf's statistics.

Figure 6-2 shows the result of the experiment: the comparison of received throughput at end hosts, measured by the receiving hosts. The left graph is throughput obtained using single queue model, while the right graph is the result obtained using multiple queues system in our model.

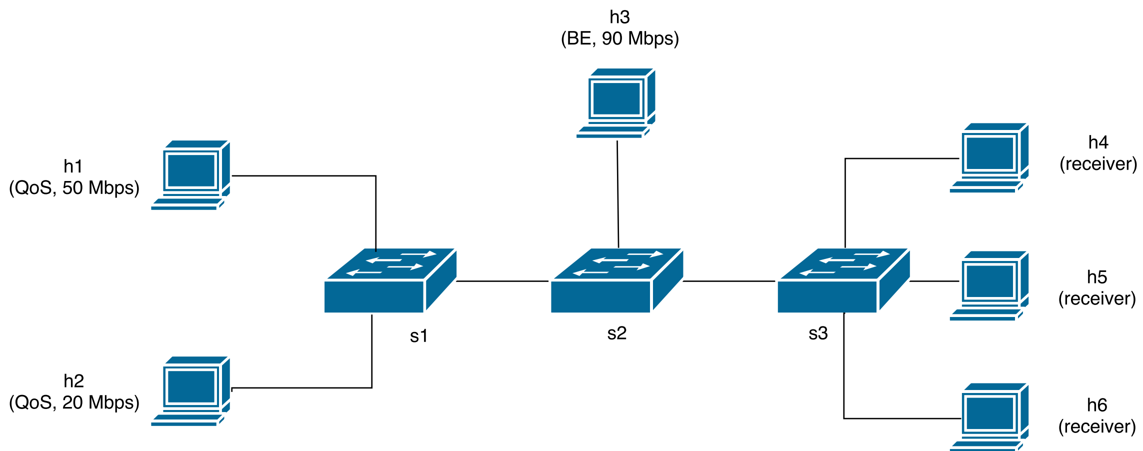
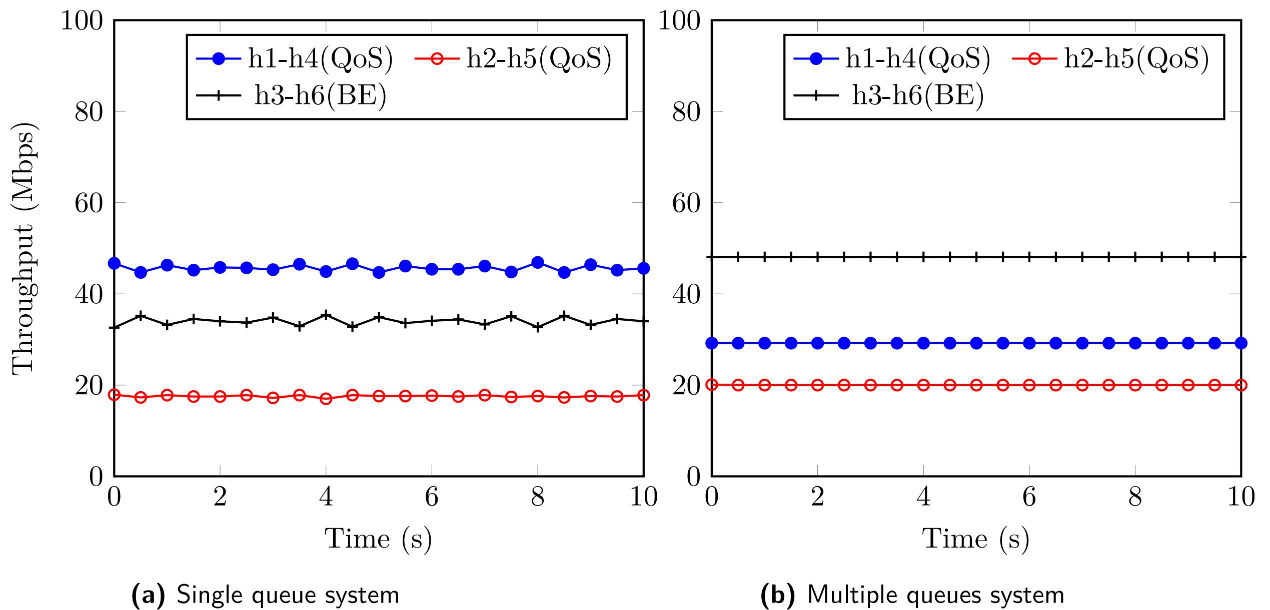


Figure 6-1: Topology used in experiment 6-1



(a) Single queue system

(b) Multiple queues system

Figure 6-2: Received throughput for experiment 6-1

In the left graph, the received throughput of flow h2-h5 is 18 Mbps, less than the transmitted rate of 20 Mbps. Although it is only slightly less than the transmitted rate, it is unacceptable because this is a QoS flow that is given a guaranteed rate. It happens because in the second switch (s2), flow h1-h4 and h2-h5 are forwarded via single intermediate queue. The guaranteed rate in this intermediate queue is 60 Mbps; however, the traffic shaping mechanism cannot distinguish between the two QoS flows. Since the total rate forwarded via this queue is 80 Mbps (60 Mbps + 20 Mbps), the two QoS flows are competing for bandwidth and the smaller rate queue is penalized.

The sum of the QoS flows' rate itself is more than 60 Mbps (around 63-64 Mbps) because the idle bandwidth is shared proportionally according to the queue/class rate. In this case, the QoS queue/class rate is 60 Mbps, while the class/queue for best-effort flow is 10 Mbps. Thus,

after the 60 Mbps guaranteed rate has been satisfied, the remaining bandwidth is given to QoS and best-effort flow with 6:1 proportion.

This problem does not appear in our model. Exclusive usage of queue ensures bandwidth separation between QoS flows in all switches. Received rate for flow h2-h5 is 20 Mbps, equal to the transmitted rate. In addition, flow h1-h4 is received at 30 Mbps (equal to its guaranteed rate) since the best-effort is prioritized over excess QoS traffic.

6-2 TC's traffic shaping reliability

Linux HTB is used as traffic shaping mechanism for bandwidth guarantees in our model. HTB employs bandwidth borrowing concept that allows child class to borrow tokens from its parent class. Inherently, the QoS flow bandwidth reservation is "virtual", as the "reserved" bandwidth is not exclusive to the reserving flow. Therefore, it is necessary for this model that the "borrowed" bandwidth should be returned instantly if a QoS flow requires the resource. In this section, a simulation is conducted to investigate the adverse effect that might occur as a result of bandwidth borrowing mechanism.

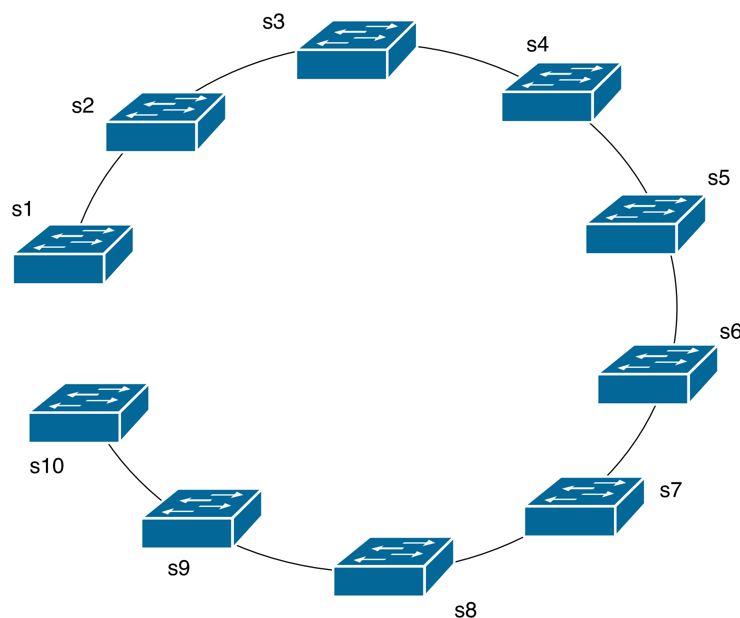


Figure 6-3: Topology used in experiment 6-2

For this simulation, a network shown in Figure 6-3 is used. The topology of the network is linear with 10 hosts, shown in Figure 6-3. For each of these switches, one host is connected. Switch s1 connected to host h1, switch s2 to host h2 and so on.

QoS flows is generated from host h2 to host h4 and h9. Flow h2-h4 has 2 hops in its path, while flow h2-h9 flow has 7 hops. Both of these flow have guaranteed bandwidth of 30 Mbps. The actual traffic sent by the sender is 30 Mbps, equal to the guaranteed bandwidth. For each QoS flow, 5 MBytes of data is sent from sender to receiver. Flows h2-h4 and h2-h9 are generated alternately, one flow at a time, and repeated 1000 times. There is a 5-seconds

Table 6-1: Traffic generated in experiment 6-2

Host pairs	Traffic Type	Guaranteed rate	Actual rate	Lifetime	Remark
h1-h10	Best-effort	N/A	100 Mbps	Throughout the simulation	Used to "disturb" the QoS flows
h2-h4	QoS	30 Mbps	30 Mbps	5 MBytes data	Sent alternately with flow h2-h9
h2-h9	QoS	30 Mbps	30 Mbps	5 Mbytes data	Sent alternately with flow h2-h4

interval between flows to make sure flow entries from the previous flow is already removed. Table 6-1 summarizes the traffic.

There are several scenarios performed in this simulation. In the first scenario, a best-effort flow is generated between two end nodes, s1 and s10, before the QoS flows start. The rate of this best-effort flow is 100 Mbps, equal to the capacity of the links. This flow is a long-lived flow that alive during the whole period of simulation. Since the QoS flows uses a subset of the best-effort flow's path, we expect to see some disturbance in the guaranteed QoS flow. We want to see how fast the "borrowed" bandwidth by the best-effort flow is "returned" to be used by the QoS flow, and much it affects the performance of the QoS flow. For comparison, in the other scenarios, the best-effort flow uses 90%, 50% and 0% capacity of the links.

Both BE flows and QoS flows are generated using iperf. Packet loss of the QoS flows is measured as parameters of system goodness. The measurement is performed in the receiver using iperf's own statistics report. Figure 6-4 shows the results of the simulation.

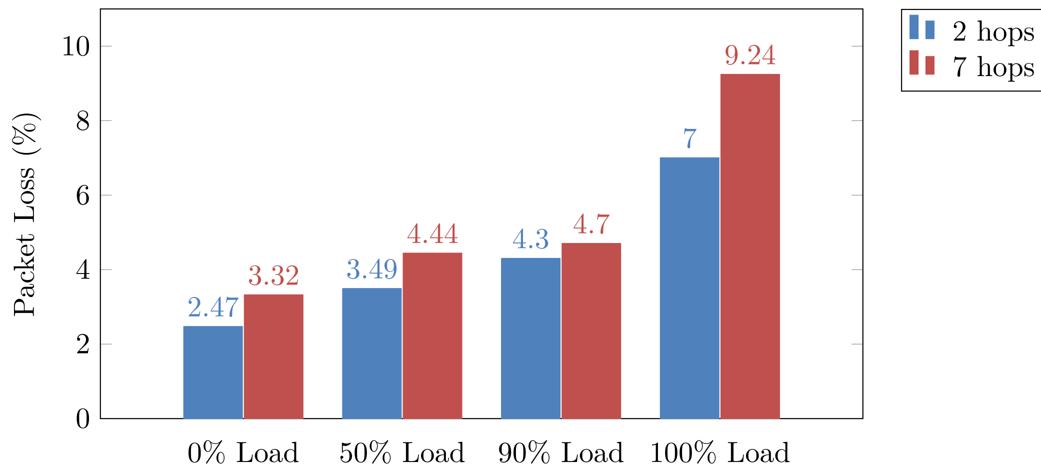


Figure 6-4: Packet loss in traffic shaping reliability simulation

For the scenario with no initial network load (0% load), the QoS flow is flowing normally with relatively low packet loss. The loss occurs because of the time needed to install flow entries in the switches, as happens in regular OpenFlow's flow installation. Naturally, flow with more hops in the path has higher packet loss, as it needs more time to install flow entries in the

switches. As a reference, Figure 6-5 shows the time needed to install flow entries for a various number of hops.

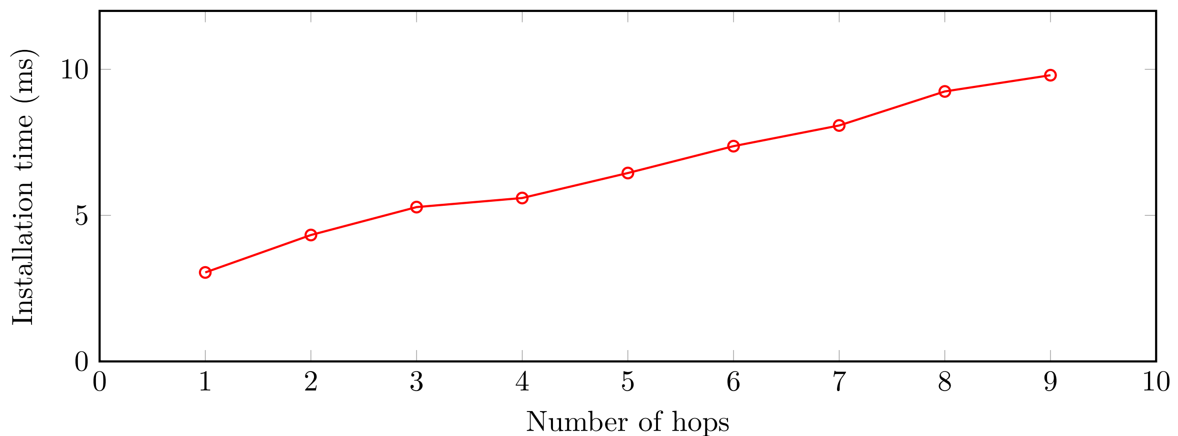


Figure 6-5: Flow entries installation time for various number of hops

During the flow entries installation process, the arriving packets are buffered in the OpenFlow switch's buffer. These packets are waiting for this process to finish because the switch does not know how to process this packet. The switch's buffer has a limited memory. Because the QoS flow rate is 30 Mbps, which is quite high, the switch's buffer cannot accommodate all of this newly arriving packets. When the buffer is full, some packets are dropped. Flow with more hops needs to install flow entries in more switches; consequently, the switches drop more packets and the packet loss is higher.

In the token bucket filter algorithm, which is used by Linux HTB, tokens are generated at a rate that corresponds to the configured rate. In our case, the configured rate is 100 Mbps, while the actual rate (best-effort flows) at the moment is also 100 Mbps. Packets are forwarded using token. If no tokens are available, packets are queued, up to the queue size. In the scenario with no best-effort traffic (0% load), all the generated tokens are conserved, as no other packets use the HTB instance. As soon as the flow entries are installed, the packets are dequeued from the buffer and forwarded through the queue specified in the flow entry. The HTB class has accumulated enough token in HTB to send the queued packets with minimal loss.

For the scenario with 50% and 90% occupancy, the tokens are still generated and reserved. But because some of the tokens are used by the existing best effort packets, the conserved tokens are not as many as in the previous scenario. After the flow entry is installed and the packets are moved from the switch's buffer to the queue. Since the queue size is limited, and there are not enough tokens to dequeued all the packets at once, some packets are dropped resulting higher packet loss.

For the scenario with 100% link occupancy, no tokens are conserved, as the token generation rate is equal to the usage rate. All the packets are moved to the queue after the flow entry is installed. None of the packets are forwarded to the output port instantly; they have to wait until new tokens are generated. The queue becomes full faster than the previous scenarios, resulting more packet loss.

Table 6-2: Packet loss comparison of QoS flows (100 Mbps BE flow)

Setting	2 hops	7 hops
With flow installation	7%	9.24%
Without flow installation	0%	0%

It is important to notice that the packet loss only occurs in the first few milliseconds of the flow, during the flow entries installation. In this simulation, a single QoS flow only sends 5 MBytes of data. The lifetime is very short, around 1.5 seconds. Iperf statistic (with a time resolution of 500 ms) shows that the high packet loss only occurs in the first 0.5 second. For the next 1 second, the packet loss is zero.

For comparison, we conducted another experiment. The setup is similar to the previous simulation; the network is initially 100% loaded with best-effort flow from host h1 to host h10. 5 MBytes of data is sent from host h2 to h9 (7 hops). This flow is a QoS flow with 30 Mbps rate and 30 Mbps guaranteed rate. Different from the previous, prior to this simulation, the flow entries are already installed. Thus, the controller does not have to install flows entries in the switches.

The result and comparison with the previous simulation are shown in Table 6-2. Although the bandwidth is fully used by other flow, the QoS flow has zero packet loss. There are no signs of quality degradation caused by bandwidth borrowing process by the best-effort flow. It means that the "borrowed" bandwidth is instantly returned when a QoS flow needs it. With this result, it is safe to say that the traffic shaping process by TC is precise and reliable enough to support our bandwidth guaranteeing model.

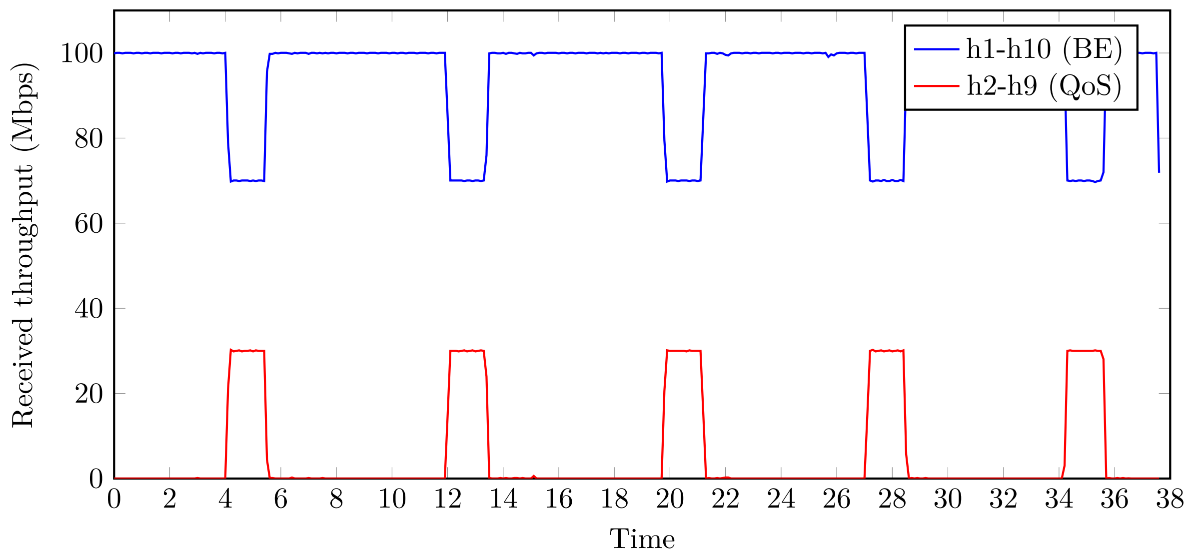


Figure 6-6: Received throughput of long lived best-effort flow and on-off QoS flow

Figure 6-6 shows the throughput of the received packet in host h9 and host h10. This data is obtained with tcpdump with a time resolution of 0.1 ms. It only takes 1.5 seconds to transfer 5 MBytes of data; yet, we can see in the graph that the QoS flow almost instantaneously

increases its rate from 0 Mbps to 30 Mbps. On the other hand, the best-effort flow's rate (as the borrower of the bandwidth) decreases during this duration.

6-3 Loading test

One of the primary goals of this thesis is to prioritize best-effort over excess QoS traffic by utilizing queue's priority parameter. Without prioritization, excess QoS traffic in the network will have the same priority as best-effort; thus, the two type of traffic will compete for bandwidth.

In our model, best-effort traffic has higher priority than excess QoS. To see how much the prioritization affects and benefits the best-effort flows, we conducted a loading test simulation. In this test, the network is highly loaded with best-effort traffic. QoS flows are then generated randomly from various hosts. The rate of QoS flows is random with average equal to the guaranteed rate. Some flow's rate are higher than the guaranteed rate, while the others are lower. With this arrangement, we expect to see the effects of the prioritization. For QoS flow with a lower actual rate, the difference between the actual and the guaranteed rate is "idle bandwidth" which can be used by any other flow. While QoS flow with an actual rate higher than the guaranteed will have excess packets. This excess traffic will be treated differently in a system with and without best-effort traffic prioritization.

The network topology shown in Figure 6-7 is used for this experiment. There are 10 hosts, each of them connected to two hosts. Hosts h1-h9 are generating QoS traffic, while hosts h11-h19 generating best-effort traffic. Host h10 and h20, connected to switch s10, are designated to be traffic sinks, to which all sender hosts send their packets. All links have a bandwidth of 100 Mbps. To make sure that no packets are dropped by the last switch (s10), the links between s10 and its hosts are enlarged to 200 Mbps.

Best-effort traffic from hosts h1-h9 is sent with a rate of 30 Mbps. This rate is constant during the simulation run-time.

Each of the QoS flows is 10s long and repeated 200 times. For each iteration of the simulation, there are 1800 QoS flows in total, generated by 9 hosts. All of the QoS flows has a guaranteed rate of 30 Mbps; thus, any links in the network will only able to accommodate 3 QoS flow at the maximum. Since there is only one sink in the network, this setup creates a bottleneck at link s9-10 and s8-s10.

After a QoS flow is finished, there is a random backoff time (between 5-10s) before the sender starts another QoS flow. The backoff time makes sure that every host has a random chance to use the network despite the bottleneck.

There are two QoS flow rates, 20 Mbps and 40 Mbps, which is chosen randomly. These two rates are 10 Mbps less or more than the guaranteed rate. By randomizing rate, we expect to see both QoS flows with lower and higher rate than its guaranteed rate. More importantly, we want to see how it affects the rate of BE flows.

Figure 6-8 shows the result of the simulation. The graph shows the average received throughput in the sink hosts (h10 for QoS and h20 for best-effort). The total of average throughput for both types of flows is 194 and 195 Mbps, which is only slightly less than the total capacity of the two bottleneck links s8-s10 and s9-s10 (200 Mbps).

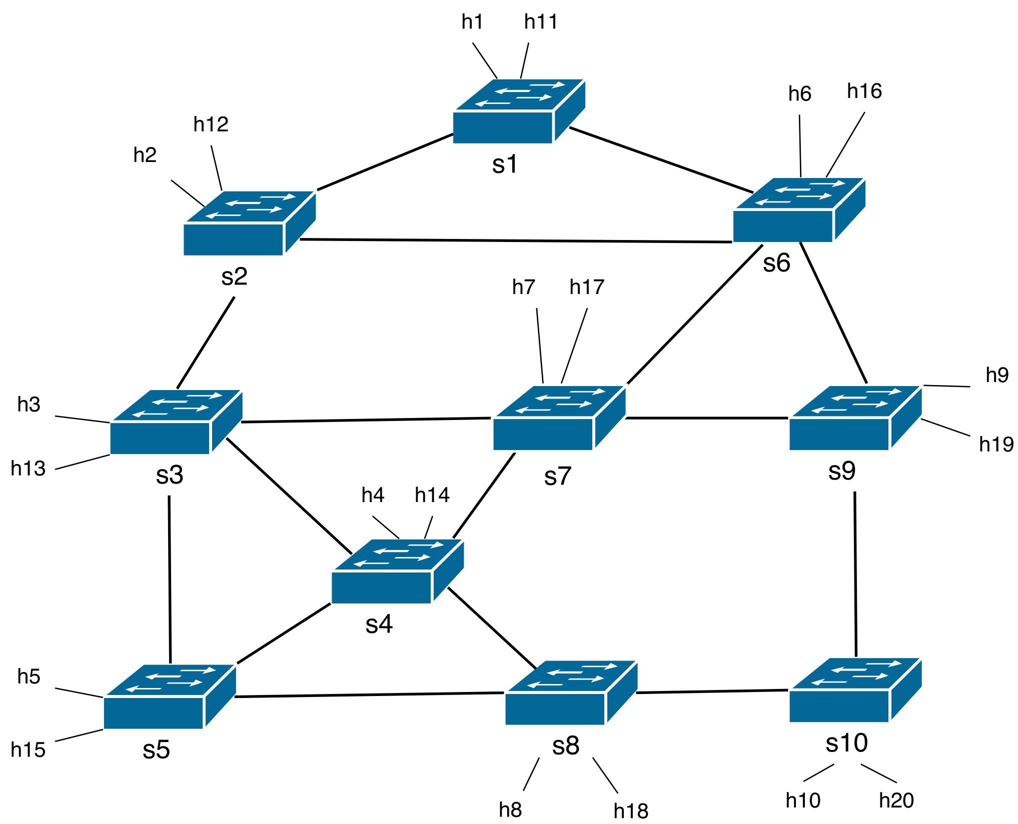


Figure 6-7: Network topology for loading test

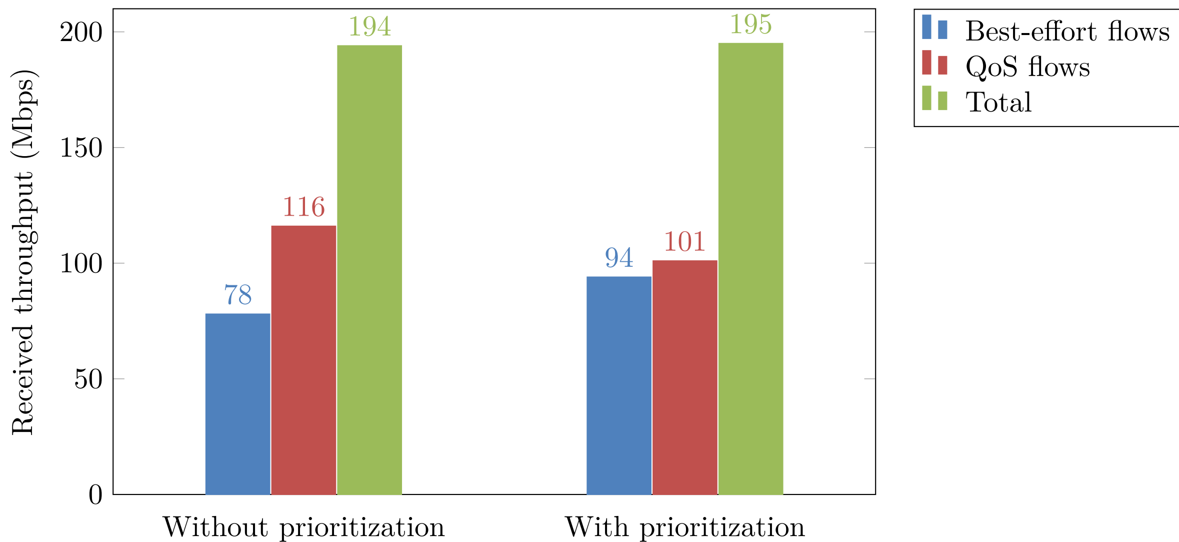


Figure 6-8: Received throughput, loading test experiment

Consider the 100% loaded of link s8-s10 and s9-s10. In the system without prioritization, best-effort and excess QoS traffic both have a chance to use the bandwidth even in a fully loaded link. In the bandwidth borrowing mechanism, after the guaranteed rate is satisfied, the parent class' token are distributed proportionally to the child class' `rate`. QoS class `rate` is 30 Mbps, and the best-effort class `rate` is 10 Mbps. The tokens are given to excess QoS and best-effort traffic in the ratio of 3:1. So, apart from the guaranteed rate of 30 Mbps, the QoS flows are allowed to send excess packets three times more than the best-effort flow. Any packet that does not get any token is delayed or dropped.

In the system with prioritization, it happens differently. After the guaranteed rates for QoS flows are satisfied, only best-effort packets are allowed to borrow bandwidth. In a network with 100% utilization, it means that all excess QoS packets are delayed or dropped. Consequently, the average throughput for best-effort flows is higher, increases 20% from the previous scenario.

From this experiment, we can see that the prioritization system give benefit to the best-effort traffic because it has more priority than the excess QoS traffic. In the case of congestion, excess QoS packets are the first to be dropped, allowing QoS flows to get higher throughput. This result confirms that our controller works as the design consideration in Section 4-2.

6-4 Multi-queues system suitability

In our models, OpenFlow queues in the switch's ports are created when the switch first connected to the controller. The queues are created with the `ovs-vsctl` command. Figure 6-9 shows the time needed to create a various number of queues.

The queue creation time is nearly linear. For 1000 queues, a total time of 800 ms is needed to complete the queues creation. Although it seems to be lengthy, it is important to note that

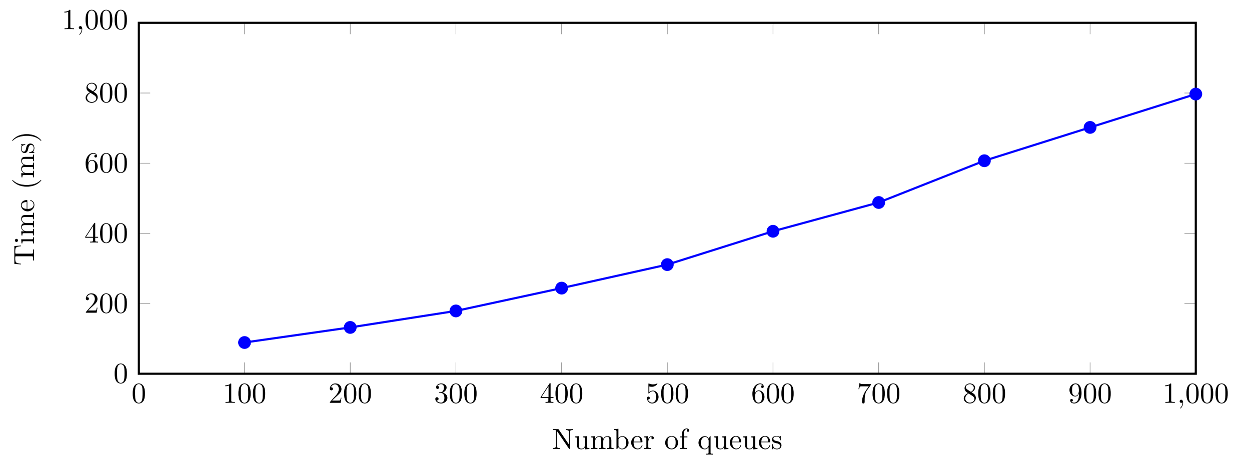


Figure 6-9: Queues creation time for various number of queue

this operation only performed once.

Compared to the "single queue" system, in which for multiple QoS flows are aggregated via a single queue, the multi-queues system presented in this chapter requires a higher number of queues. For every switch in the path, a flow requires one queue for bandwidth guaranteeing. In this sense, the multi-queues system does not solve the scalability problem; flow states are still stored in all switches in the path. However, if we want to allow excess QoS traffic in Open vSwitch, this is the only way to guarantee a certain amount of bandwidth in every link in the path.

In the next chapter, we will examine the aggregation concept using OpenFlow meter table. The aggregation reduces the number of queues used. Because Open vSwitch does not support meter table, the model is deployed using Pica8 hardware switch.

Chapter 7

Meter Table and Flow Aggregation

In the previous chapter, we see that our model requires one queue for every flow in every switch in the path. To reduce the number of the queue, the QoS flows should be aggregated. In this chapter, we explore the possibility of using meter table for flow aggregation. A proof of this concept is presented by conducting an experiment with Pica8 switches.

7-1 Metering and Aggregation Concept

The basic idea is to classify a QoS flow into two different flows, i.e. conformant and excess QoS flows. These two flows are then forwarded via different queues with different priorities.

Metering operation with *dscp_remark* allows the switch to split the flows by altering the DSCP/ToS bits of excess packets. Each QoS flow is passed through a meter table entry with the meter's rate equal to the guaranteed rate. A QoS flow with an actual rate exceeding the meter rate will have its excess packets' DSCP/ToS bits remarked. This operation will change the DSCP/ToS bits of the non-conformant packets while the conformant packets' DSCP bits remain the same.

Conformant QoS traffic from multiple flows are aggregated and forwarded via a single queue. Aggregation also performed for excess QoS and best-effort traffic. The aggregation is possible because conformant and excess packets now have distinct ToS bits; thus, they can be easily identified. By using three separate queues for conformant QoS, excess QoS, and best-effort traffic, we can easily arrange the priority for these traffic.

We took advantage of the multi-table pipeline processing in OpenFlow, which allows the switch to process the packets multiple times in a sequential order. In this case, the metering is performed in flow *Table 0*. Then, flow *Table 1* matches ToS bits and forwards the packets to the aggregation queue. With aggregation, every switch port will only need three queues, i.e. one for conformant QoS, one for excess QoS, and one for best-effort traffic.

1. ***Conformant queue.*** The queue for conformant traffic of all QoS flows forwarded via a particular switch port. This queue has the highest priority.

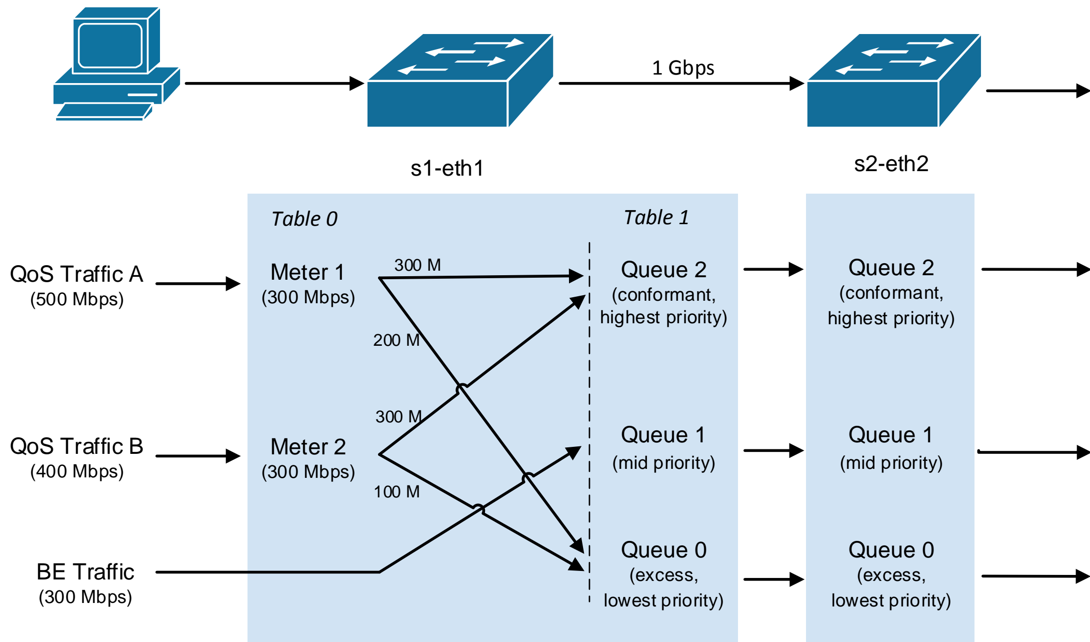


Figure 7-1: Metering-aggregation concept

2. ***Excess queue.*** The queue for excess QoS traffic of all QoS flows forwarded via a particular switch port. This queue has the lowest priority, as we want to give more priority to the best-effort traffic.

3. ***Best-effort queue.*** The queue for best-effort traffic of all QoS flows forwarded via a particular switch port. This queue has a priority between the two queues above.

Using this setting, we no longer have to set the `min-rate` for the queues. As now each traffic types are forwarded via exclusive queues, we only have to set the priority of the queues.

Figure 7-1 gives an illustration of this arrangement. The two QoS flows (ToS 64), flow A and flow B, are passed to meter table entries. The rate of the meter is 300 Mbps, which is equal to the guaranteed rate of these QoS flows. The conformant packets keep the ToS of 64, while excess packets change the ToS to 4. After the metering, all the packets are processed further in the flow *Table 1* using `goto_table` instruction. In *Table 1*, the packets are matched against the ToS bits and then processed accordingly. Packets with ToS 64 are queued in queue 2 which has the highest priority, while packets with ToS 4 are queued in queue 0 which has the lowest priority.

The following is the flow entries of QoS Flow A in the first switch.

```
ovs-ofctl -O Openflow13 add-flow br0
table=0,dl_type=0x800,nw_src=10.31.32.101,nw_dst=10.30.128.109,nw_tos=64
actions=meter:1,goto_table:1
```

```
ovs-ofctl -O Openflow13 add-flow br0
table=1,dl_type=0x800,nw_src=10.31.32.101,nw_dst=10.30.128.109,nw_tos=64
actions=set_queue:2,output:5

ovs-ofctl -O Openflow13 add-flow br0
table=1,dl_type=0x800,nw_src=10.31.32.101,nw_dst=10.30.128.109,nw_tos=4
actions=set_queue:1,output:5
```

The aggregation significantly cuts down the number of queues that is used in the system. In multiple-queues system, presented in the previous chapters, every single QoS flow uses one queue in every switch in the path. As the number of flows grows, the number of queues also increases. In contrast, with aggregation, we only need two queues for all QoS flows. The complexity of the system becomes much lower and while the system's scalability increases.

The aggregation system presented in this chapter makes a distinction between conformant and non-conformant (excess) QoS flow. This aggregation is different from the model proposed in [12]. In [12], a single queue is used to aggregate all QoS flows; there is no distinction between the two types of traffic. In this system, it is not possible for a QoS flow to send more than the guaranteed rate. In fact, there is a maximum limit for the queue. If there is one flow in the aggregation exceeding its guaranteed rate, it will contend with other flows.

By differentiating three types of traffic with its DSCP/ToS bits, we can easily arrange the traffic prioritization. In our model, we give priority to the best-effort over the excess QoS traffic. But if we want different prioritization setting, it can be easily arranged. For example, by forwarding best-effort and excess QoS traffic to a single queue, the two types of traffic will have the same priority.

7-2 Metering in Pica8 switch

To prove the concept of metering and aggregation, we perform an experiment using Pica8 switches on the SURFnet SDN testbed [44]. The testbed uses Pica8 P5101 switches, running PicOS 2.7.1. This hardware switch is based on the Broadcom Trident II ASIC (Application-Specific Integrated Circuit) [25]. The switches are connected to OpenStack virtual machines as hosts. All the virtual machines run Ubuntu 14.04 with 4 GB RAM and 2 VCPUs.

Figure 7-2 shows the network setup for the experiment. UDP traffic is sent from vm1-vm4 to vm5, generated with iperf. The capacity of the link between two switches is 1 Gbps. The traffic generated by each host is shown in the Figure 7-2. This experiment is analogous to the one we conducted in Section 5-2.

Unfortunately, we found that the ASIC implementation of OpenFlow in Pica8 switch has limitations in its functionality. First, the multi-table implementation is still limited [45]. The Pica8 switch uses a Broadcom Trident II ASIC, which is designed for a specific networking application. This chip is mainly used in legacy networking (non-OpenFlow) devices and designed to be used for legacy protocols. Its implementation in OpenFlow has allowed more control on the forwarding planes; however, there are still several limitations such as in the

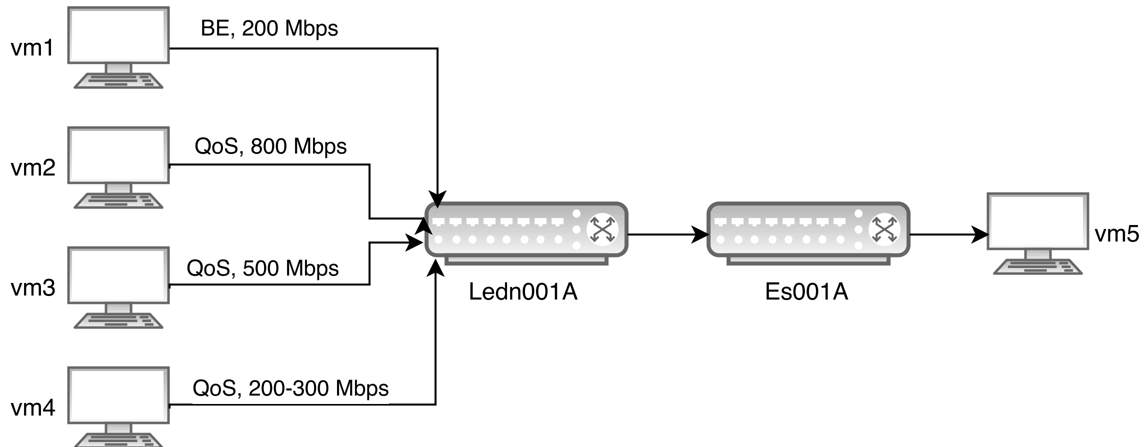


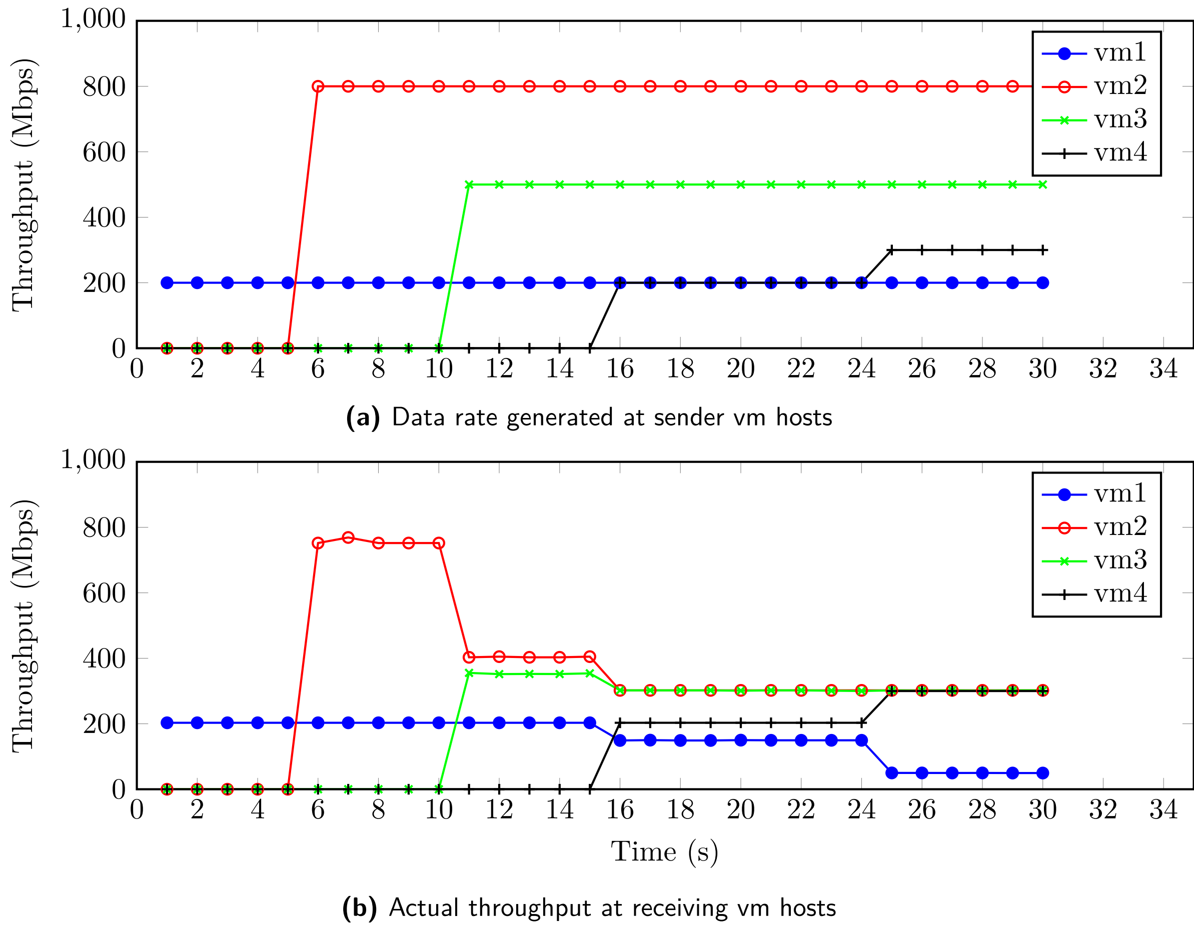
Figure 7-2: Topology for the metering and aggregation experiment

multi-tables. It does not allow recirculation of packets via multi-tables pipeline. Thus, we can not pass the result of metering to *Table 1* of the ingress switch; the DSCP-remarked packets will only appear in the next switch.

We also found a problem when forwarding multiple flows originating from a single ingress port. Consider a scenario in which two flows are entering the same switch port. These two flows are matched with two different flow entries and instructed to use two different queues. Despite these explicit settings, the flows are still forwarded via a single queue (the one with less priority). This anomaly is inconsistent with the OpenFlow standard. The `set_queue` action sets the queue in which a packet should be processed, disregarding how the packet is matched. As we demonstrate in Chapter 5, this problem does not occur in the Open vSwitch.

To overcome these problems, we use a workaround by making a self-loop in around the ingress switch. Instead of being processed in multi-tables, the packets are processed in multiple switches before entering the ingress switch again. When a flow exceeds the meter-rate, the two different DSCP bits (original and remarked) appear at the egress port of the ingress switch. We also arrange that the conformant and excess packets re-enter the ingress switch from different ports to prevent them from using the same queue. The workaround setup is described in Appendix A.

Figure 7-3 shows the data rate received in vm5. This result is similar to the one obtained in Section 5-2. The only difference occurs in second 10-15. In the Section 5-2 experiment (Figure 5-4b) flows from h2 and h3 get an equal rate. It happens because each QoS flow uses an exclusive queue, and the idle bandwidth is distributed equally to the excess QoS traffic using round-robin scheduling. In this experiment, it happens differently. All QoS flows share a common *excess queue*. Since the excess traffic from vm2 is higher than the one from vm3, more packets from vm2 are passed through the queue, resulting a higher received rate at vm5. Nevertheless, the difference does not conflict our design consideration, since the main goal of giving a 300 Mbps guarantee to the QoS flows is already achieved.



(a) Data rate generated at sender vm hosts

(b) Actual throughput at receiving vm hosts

Figure 7-3: Prioritization test with metering in Pica8 testbed

Analysis

The aggregation significantly reduces the number of queues needed in the intermediate switches. No matter how many flows we have in the network, all switches in the network will only need three queues. Other than the flow entries, there are no additional flow states stored in the intermediate switches. The flow entries itself is an absolute OpenFlow requirement that exists in any OpenFlow applications. Additional complexities only present in the first switch in the form of meter table and pipeline processing which have to be added for every individual QoS flow. Nevertheless, the additional complexity is relatively trivial compared to a multiple-queues system. It also less complex compared to IntServ in which flow states are stored in every switch in the path. Therefore, we can conclude that the model is scalable.

From the experiment, we see that the firm bandwidth guarantees are still holding when the traffic are aggregated. It is important to notice that the problem we presented at Section 6-1 does not appear in our aggregation model. Although both our model and the model presented in [12] use aggregation, our model differentiates conformant and excess traffic and puts them in different queues. The queue differentiation provides separation between the two traffic; ensuring excess traffic from one QoS flow will not interfere with other QoS flows.

Chapter 8

Conclusion and Future Work

8-1 Conclusion

In this thesis, an end-to-end bandwidth guarantee model is proposed. The guaranteeing effort is implemented in both controller and switch level. In the controller, bandwidth for QoS flows is guaranteed using an admission process. The controller always chooses a path that can accommodate the required rate. If none available, the controller rejects the flow. In the switch, OpenFlow's queue guarantees data rate to QoS flows by using traffic shaping and policing. From the simulation, we see that the model gives a per-flow level bandwidth guarantees.

Our contribution in this thesis is a new model in which QoS flows are allowed to send more than the guaranteed rate. In other OpenFlow hard-QoS guaranteed models, such as presented in [11] and [12], QoS flows' rates are strictly regulated. In these models, QoS flows' rates are not only guaranteed, but the maximum rates are also limited. QoS flows are explicitly not allowed to send more than the guaranteed rate. The reason is because excess QoS traffic might disturb best-effort and conformant traffic from other QoS flows, as they compete for resources in the case of congestion. However, in a network with low utilization, this limitation is a loss of opportunity. Allowing flows to send more than the guaranteed rate lets the flows to finish earlier, freeing network resources for other flows that appear later.

Our model also gives new insight on traffic prioritization in which excess (non-conformant) QoS traffic have lower priority than best-effort. This is a departure from IntServ's Guaranteed Service in which excess QoS traffic get a similar treatment as regular best-effort traffic. We take an advantage of the `min-rate` and `prioritization` parameters of OpenFlow queues for the prioritization. To put it in short: "the switch prioritizes a queue over other queues after min-rate for all queues are satisfied". This way, we can easily put a differentiation of guaranteed traffic and excess traffic of QoS flow in a single queue.

While QoS flows go through the admission process, best-effort flows are routed via shortest path and are allowed to use any of the idle bandwidth. However, when a new QoS flow appears, the bandwidth should be "returned" immediately to satisfy the QoS flow's guaranteed rate.

In the simulation, we found that Linux TC's traffic shaping is reliable enough to provide bandwidth guarantee and traffic prioritization. The excess QoS and best-effort traffic do not have adverse effects on the guaranteed flow's performance. In addition, we also found that the bandwidth borrowing concept in Linux HTB increases the network utilization by letting other flows using the unused reserved bandwidth.

In Chapter 5 we explore the possibility of using meter table with *dscp_remark* for traffic aggregation. The metering alters the DSCP/ToS bits of the excess packets. Having different ToS bits for different traffic types (conformant QoS, excess QoS, and best-effort), we can easily aggregate the traffic using only three queues in each switch port. For the proof of concept, we conduct the experiment using Pica8 switches that support the OpenFlow meter table. Compared to Intserv, this model uses less reservation signaling messages, since it does not require periodic reservation state refreshment, as shown in Figure 4-1. By using aggregation, a switch does not need to store every single flow state it forwards. These two improvements solve the scalability problem commonly associated with IntServ. We believe that by having the scalability problem solved, end-to-end QoS guarantees is ready to be implemented in a larger network such as Internet.

Unfortunately, the different switch implementations of the OpenFlow standard are causing difficulties in the deployment. In our experiment with Pica8 switch, we found several OpenFlow features on the switch that are not working as they are supposed to. Also, Open vSwitch as the most popular OpenFlow switch does not support the OpenFlow meter table in its software implementation. In the future OpenFlow release, we strongly recommend the ONF and OpenFlow vendors to put an effort to resolve these issues. Then, the OpenFlow can have another advantage over the legacy IP network: a scalable end-to-end QoS guarantees.

8-2 Future work

In Chapter 5 we present a proof of concept of metering and aggregation using hybrid OpenFlow switches based on ASICs. This kind of hardware has limited OpenFlow 1.3 functionality and is proven to be problematic as there are several features that do not work according to the standard. A further experiment needs to be conducted using OpenFlow-only switches. An OpenFlow-only switch based on NPU (Network Processor Unit) based switches, such as NoviFlow [46], does not implement hybrid features; therefore, it has better supports for the features that we need in our model, i.e. multi-tables pipeline and meter table. The experiment also needs to be extended to include other QoS parameters such as delay and jitter.

Bibliography

[1] F. A. Kuipers. *Quality of Service Routing in the Internet. Theory, Complexity and Algorithms*. TU Delft, Delft University of Technology, 2004.

[2] R. Braden, D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview*. RFC 1633 (Informational). Internet Engineering Task Force, 1994. URL: http://www.ietf.org/rfc/rfc1633.txt.

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *An Architecture for Differentiated Services*. RFC 2475 (Informational). Updated by RFC 3260. Internet Engineering Task Force, 1998. URL: http://www.ietf.org/rfc/rfc2475.txt.

[4] A. P. Bianzino, C. Chaudet, D. Rossi, and J. L. Rougier. "A survey of green networking research". In: *Communications Surveys & Tutorials, IEEE* 14.1 (2012), pp. 3–20.

[5] *The ATM Forum*. URL: http://www.atmforum.com/.

[6] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*. RFC 2205 (Proposed Standard). Updated by RFCs 2750, 3936, 4495, 5946, 6437, 6780. Internet Engineering Task Force, 1997. URL: http://www.ietf.org/rfc/rfc2205.txt.

[7] Y. Jinyao, Z. Hailong, S. Qianjun, L. Bo, and G. Xiao. "HiQoS: An SDN-based multipath QoS solution". In: *Communications, China* 12.5 (2015), pp. 123–133.

[8] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp. "OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks". In: *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*. IEEE. 2012, pp. 1–8.

[9] R. Wallner and R. Cannistra. "An SDN approach: quality of service using big switch's floodlight open-source controller". In: *Proceedings of the Asia-Pacific Advanced Network* 35 (2013), pp. 14–19.

[10] M. R. Celenlioglu and H. A. Mantar. "An SDN Based Intra-Domain Routing and Resource Management Model". In: *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE. 2015, pp. 347–352.

[11] S. Tomovic, N. Prasad, and I. Radusinovic. "SDN control framework for QoS provisioning". In: *Telecommunications Forum Telfor (TELFOR), 2014 22nd*. IEEE. 2014, pp. 111–114.

[12] S. Dwarakanathan, L. Bass, and L. Zhu. "Cloud Application HA using SDN to ensure QoS". In: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE. 2015, pp. 1003–1007.

[13] C. Doerr, R. Gavrila, F. A. Kuipers, and P. Trimintzios. "Good practices in resilient internet interconnection". In: *ENISA Report, Jun* (2012).

[14] N. L. van Adrichem, B. J. Van Asten, and F. A. Kuipers. "Fast recovery in software-defined networks". In: *2014 Third European Workshop on Software Defined Networks*. IEEE. 2014, pp. 61–66.

[15] N. L. van Adrichem, F. Iqbal, and F. A. Kuipers. "Computing backup forwarding rules in Software-Defined Networks". In: *arXiv preprint arXiv:1605.09350* (2016).

[16] B. J. van Asten, N. L. van Adrichem, and F. A. Kuipers. "Scalability and resilience of software-defined networking: An overview". In: *arXiv preprint arXiv:1408.6760* (2014).

[17] B. A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti. "A survey of software-defined networking: Past, present, and future of programmable networks". In: *IEEE Communications Surveys & Tutorials* 16.3 (2014), pp. 1617–1634.

[18] F. A. Kuipers, P. Van Mieghem, T. Korkmaz, and M. Krunz. "An overview of constraint-based path selection algorithms for QoS routing". In: *IEEE Communications Magazine, 40 (12)* (2002).

[19] P. Van Mieghem and F. A. Kuipers. "Concepts of exact QoS routing algorithms". In: *Networking, IEEE/ACM Transactions on* 12.5 (2004), pp. 851–864.

[20] S. Jain et al. "B4: Experience with a globally-deployed software defined WAN". In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 3–14.

[21] Open Networking Foundation. "OpenFlow Switch Specification version 1.3.0". 2012.

[22] N. McKeown and B. Girod. "Clean slate design for the internet". 2006.

[23] *Open vSwitch*. Accessed 2016. URL: http://openvswitch.org/.

[24] *OpenFlow 1.3 Software Switch*. Accessed 2016. URL: http://cpqd.github.io/ofsoftswitch13/.

[25] Inc. Pica8. "PicOS Overview - Whitepaper". 2014. URL: http://www.pica8.com/documents/pica8-whitepaper-picos-overview.pdf.

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

[27] *Open vSwitch database schema*. Accessed 2016. URL: http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf.

[28] B. Sonkoly et al. "On qos support to ofelia and openflow". In: *Software Defined Networking (EWSDN), 2012 European Workshop on*. IEEE. 2012, pp. 109–113.

[29] K. Jeong, J. Kim, and Y. Kim. "QoS-aware network operating system for software defined networking with generalized OpenFlows". In: *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE. 2012, pp. 1167–1174.

[30] *OpenFlow in Europe: Linking Infrastructure and Applications*. Accessed 2016. URL: http://www.fp7-ofelia.eu/.

[31] S. Sharma et al. "Implementing quality of service for the software defined networking enabled future internet". In: *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE. 2014, pp. 49–54.

[32] H. E. Egilmez and A. M. Tekalp. "Distributed QoS architectures for multimedia streaming over software defined networks". In: *Multimedia, IEEE Transactions on* 16.6 (2014), pp. 1597–1609.

[33] *Traffic Control HOWTO*. Accessed 2016. URL: http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html.

[34] I. Stoica, H. Zhang, and T. S. Ng. *A hierarchical fair service curve algorithm for link-sharing, real-time and priority services*. Vol. 27. 4. ACM, 1997.

[35] B. B. Grossman. "An overview of frame relay technology". In: *Computers and Communications, 1991. Conference Proceedings., Tenth Annual International Phoenix Conference on*. IEEE. 1991, pp. 539–545.

[36] S. Shenker, C. Partridge, and R. Guerin. *Specification of Guaranteed Quality of Service*. RFC 2212 (Proposed Standard). Internet Engineering Task Force, 1997. URL: http://www.ietf.org/rfc/rfc2212.txt.

[37] N. L. van Adrichem, C. Doerr, and F. A. Kuipers. "Opennetmon: Network monitoring in openflow software-defined networks". In: *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE. 2014, pp. 1–8.

[38] *Ryu SDN Framework*. Accessed 2016. URL: https://osrg.github.io/ryu/.

[39] P. Siripongwutikorn, S. Banerjee, and D. Tipper. "A survey of adaptive bandwidth control algorithms". In: *Communications Surveys & Tutorials, IEEE* 5.1 (2003), pp. 14–26.

[40] S. Terrasa, S. Sáez, J. Vila, and E. Hernández. "Comparing the utilization bounds of IntServ and DiffServ". In: *Universidad Politécnica de Valencia* (2004).

[41] A. Rao, A. Legout, Y. Lim, D. Towsley, C. Barakat, and W. Dabbous. "Network characteristics of video streaming traffic". In: *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM. 2011, p. 25.

[42] Q. Ma and P. Steenkiste. "On path selection for traffic with bandwidth guarantees". In: *Network protocols, 1997. proceedings., 1997 international conference on*. IEEE. 1997, pp. 191–202.

[43] *iPerf - The network bandwidth measurement tool*. Accessed 2016. URL: https://iperf.fr/.

[44] *SURFnet testbed*. Accessed 2016. URL: https://www.surf.nl/en/innovationprojects/the-open-programmable-network/software-defined-networking/surfnet-testbed/index.html.

[45] R. van der Pol. "Experiences with Programmable Dataplanes". The Networking Conference 2016. 2016. URL: https://tnc16.geant.org/core/presentation/663.

[46] *The Network Processor (NPU) Advantage*. Accessed 2016. URL: http://noviflow.com/the-network-processor-npu-advantage/.

Appendix A

Workaround for SURFnet Pica8 Testbed

There are two problems with the Pica8 switch when we implement our model. First, there is a problem with multi-tables pipeline. This problem does not allow us to get the *dscp_remarking* result in the same switch that performs the metering. Second, flows from the same ingress port are forced to use a single queue in the egress port. These problems occur due to the limited OpenFlow 1.3 functionality since the switch is ASIC based.

To overcome the problems, we make a workaround. The traffic is sent from a Lithium VM (connected to switch Ledn001A) to a Boron VM (connected to switch Es001A). A loop is created from Ledn001A to itself, so the traffic exits before re-renter the switch again. After the re-enter, the packets are sent to Es001A.

Figure A-1 shows the loop that is made in the SURFnet's Pica8 testbed. There are six switches connected with ten links in the testbed. Unfortunately, at the time we are conducting the experiment, there are several links down. These links are shown in red in the Figure.

The sender host generates packets with ToS 64. These packets are processed with a meter table in Ledn001A, resulting packet with ToS 64 and 4, which are sent to Gn001A. From Gn001A the packets go to Asd001A, then Es001A. Because we need ToS 64 and ToS 4 to re-enter Ledn001A from different ports, the path is split here. Packets with ToS 64 go through Dt001B, re-enter from port 11. Packets with ToS 4 go via link 5 -11 - 13 - 10 - 9, then re-enter Ledn001A from port 9.

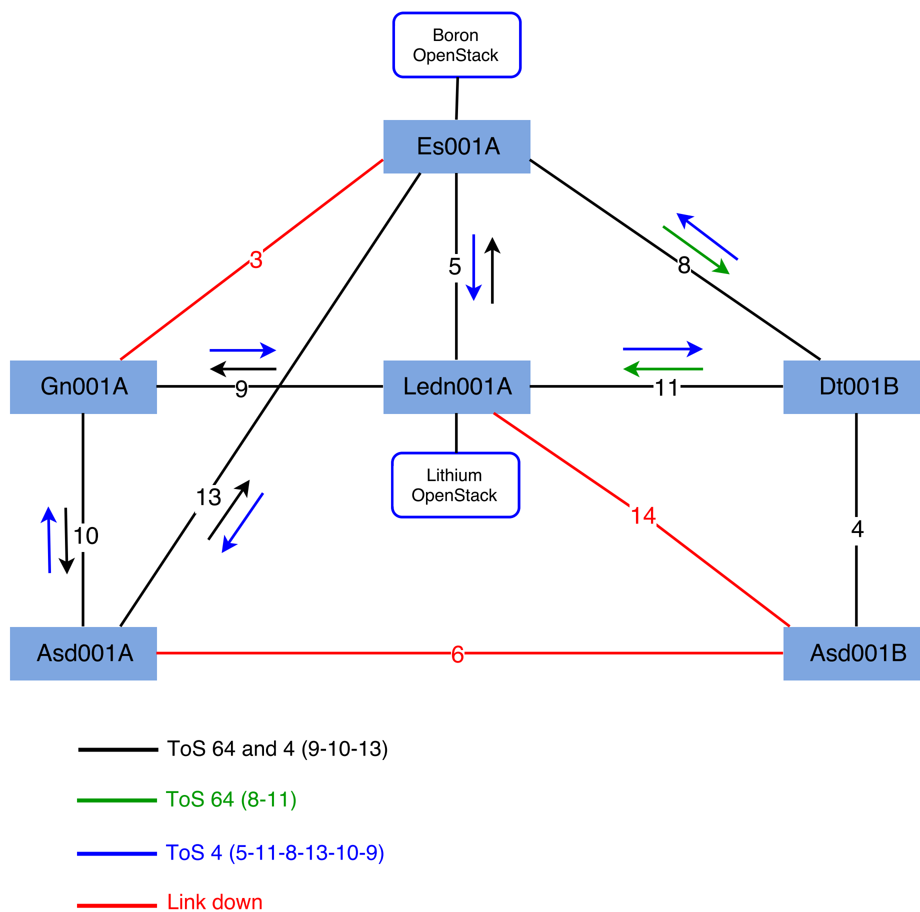


Figure A-1: Workaround for Pica8 testbed

Appendix B

Controller Source Code

The controller application consists of three files: shortest_route.py, network_aware.py and qos.py. The network awareness function is based on a patch for Ryu https://sourceforge.net/p/ryu/mailman/message/34275697/

shortest_route.py

```
1  import logging
2  import time
3  import struct
4  from operator import attrgetter
5  from ryu.base import app_manager
6  from ryu.controller import ofp_event
7  from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
8  from ryu.controller.handler import CONFIG_DISPATCHER
9  from ryu.controller.handler import set_ev_cls
10 from ryu.ofproto import ofproto_v1_3
11 from ryu.lib.packet import packet
12 from ryu.lib.packet import ethernet
13 from ryu.lib.packet import ipv4
14 from ryu.lib.packet import arp
15 from ryu.lib.packet import lldp
16 import qos
17 import uuid
18 from ryu.base import app_manager
19 import network_aware
20 import network_monitor
21
22 class Shortest_Route(app_manager.RyuApp):
23
24   OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
25   _CONTEXTS = {
26     "Network_Aware": network_aware.Network_Aware,
```

```
27       "Network_Monitor": network_monitor.Network_Monitor,
28     }
29 
30     def __init__(self, *args, **kwargs):
31       super(Shortest_Route, self).__init__(*args, **kwargs)
32 
33       self.network_aware = kwargs["Network_Aware"]
34       self.network_monitor = kwargs["Network_Monitor"]
35       self.mac_to_port = {}
36       self.datapaths = {}
37 
38       self.link_to_port = self.network_aware.link_to_port
39       self.access_table = self.network_aware.access_table
40       self.access_ports = self.network_aware.access_ports
41       self.graph = self.network_aware.graph
42       self.graphshort = self.network_aware.graphshort
43       self.shortest_graph = self.network_aware.shortest_graph
44 
45       self.route_db = {}
46       self.flow_db = self.network_monitor.flow_db
47       self.idle_time = 2
48       self.hard_time = 16000
49       self.queue_db = self.network_aware.queue_db
50       self.int_queue_db = self.network_aware.int_queue_db
51       self.int_queue_list = self.network_aware.int_queue_list
52       self.path_calc_counter = 0
53       self.timea=time.time()
54 
55     @set_ev_cls(ofp_event.EventOFPFlowRemoved, MAIN_DISPATCHER)
56     def flow_removed_handler(self, ev):
57       msg = ev.msg
58       dp = msg.datapath
59       ofp = dp.ofproto
60       tos=qos.dscp_to_tos(msg.match['ip_dscp'])
61       db_key = (msg.match['ipv4_src'], msg.match['ipv4_dst'], tos)
62       try:
63         first_sw=self.flow_db[db_key][3]
64       except KeyError:
65         return
66       out_port=self.flow_db[db_key][4]
67       path = self.flow_db[db_key][0]
68       bw = self.flow_db[db_key][2]
69 
70       if dp.id==path[0]:
71         if tos==64:
72           self.queue_db[(first_sw,out_port)][self.flow_db[db_key][1]]=None
73           qos.update_graph_release_bw(self.graph, path,bw)
74           if len(path)>2: #update intermediate queue if forwarded using
               more than 2 hops
75             for i in range(2,len(path)):
76               sw=path[i-1]
77               via_port = self.network_aware.get_port_from_link(path[i-1],
                   path[i])[0]
```

```
78             cur_rate=self.int_queue_db[(sw, via_port)]
79             if cur_rate==bw:
80                 cur_rate=bw+1000000  #avoid setting rate to 0
81             self.int_queue_db[(sw, via_port)]=cur_rate-bw   #reduce value
                   in int_queue_db
82             for i in range(5,15):
83               if self.int_queue_list[(sw, via_port)][i] == str(msg.match[
                     'ipv4_src']+msg.match['ipv4_dst']):
84                 self.int_queue_list[(sw, via_port)][i]=None
85       del self.flow_db[db_key]
86 
87   @set_ev_cls(ofp_event.EventOFPStateChange,
88         [MAIN_DISPATCHER, DEAD_DISPATCHER])
89   def _state_change_handler(self, ev):
90     datapath = ev.datapath
91     if ev.state == MAIN_DISPATCHER:
92       if not datapath.id in self.datapaths:
93         self.logger.debug('register datapath: %016x', datapath.id)
94         self.datapaths[datapath.id] = datapath
95     elif ev.state == DEAD_DISPATCHER:
96       if datapath.id in self.datapaths:
97         self.logger.debug('unregister datapath: %016x', datapath.id)
98         del self.datapaths[datapath.id]
99 
100  def add_flow(self, dp, p, match, actions, idle_timeout=0, hard_timeout
       =0):
101    ofproto = dp.ofproto
102    parser = dp.ofproto_parser
103    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
104                         actions)]
105    mod = parser.OFPFlowMod(datapath=dp, priority=p,
106                idle_timeout=idle_timeout,
107                hard_timeout=hard_timeout,
108                flags=ofproto.OFPFF_SEND_FLOW_REM,
109                match=match, instructions=inst)
110    drop=dp.send_msg(mod)
111 
112  def install_flow(self, path, flow_info, buffer_id, data, tos=0,
       queue_id=0, int_queues=[]):
113    hard_time = self.hard_time
114    idle_time = self.idle_time
115    if tos!=64:
116      int_queue_id=0
117      queue_id=0
118      idle_time=self.idle_time
119      hard_time = 3000
120    dscp=qos.tos_to_dscp(tos)
121 
122    print "install flow, queue_id", queue_id, " ", flow_info[1], " to ",
         flow_info[2]
123    in_port = flow_info[3]
124    assert path
125    datapath_first = self.datapaths[path[0]]
```

```
126       ofproto = datapath_first.ofproto
127       parser = datapath_first.ofproto_parser
128       out_port = ofproto.OFPP_LOCAL
129
130       if len(path) > 2:
131         for i in xrange(1, len(path) - 1):
132           port = self.get_link2port(path[i - 1], path[i])
133           port_next = self.get_link2port(path[i], path[i + 1])
134           timea=time.time()
135           if port:
136             src_port, dst_port = port[1], port_next[0]
137             datapath = self.datapaths[path[i]]
138             ofproto = datapath.ofproto
139             parser = datapath.ofproto_parser
140             actions = []
141
142             if tos==64:
143               for swqu in int_queues:
144                 if swqu[0]==datapath.id:
145                   int_queue_id=swqu[1]
146
147             actions.append(parser.OFPActionSetQueue(int_queue_id)) # Queue
                  for intermediate switches
148             actions.append(parser.OFPActionOutput(dst_port))
149
150             match = parser.OFPMatch(
151               in_port=src_port,
152               ip_dscp=dscp,
153               eth_type=flow_info[0],
154               ipv4_src=flow_info[1],
155               ipv4_dst=flow_info[2])
156             self.add_flow(
157               datapath, 1, match, actions,
158               idle_timeout=idle_time, hard_timeout=hard_time)
159             msg_data = None
160             if buffer_id == ofproto.OFP_NO_BUFFER:
161               msg_data = data
162
163             out = parser.OFPPacketOut(
164               datapath=datapath, buffer_id=buffer_id,
165               data=msg_data, in_port=src_port, actions=actions)
166             datapath.send_msg(out)
167
168       if len(path) > 1:
169         # the  first flow entry
170         port_pair = self.get_link2port(path[0], path[1])
171         out_port = port_pair[0]
172
173         actions = []
174         actions.append(parser.OFPActionSetQueue(queue_id)) #queue for
              originating switch
175         actions.append(parser.OFPActionOutput(out_port))
176         match = parser.OFPMatch(
```

```
177             in_port=in_port,
178             ip_dscp=dscp,
179             eth_type=flow_info[0],
180             ipv4_src=flow_info[1],
181             ipv4_dst=flow_info[2])
182           self.add_flow(datapath_first,
183                   1, match, actions, idle_timeout=idle_time, hard_timeout=
                          hard_time)
184 
185           # the last hop: tor -> host
186           datapath = self.datapaths[path[-1]]
187           ofproto = datapath.ofproto
188           parser = datapath.ofproto_parser
189           actions = []
190           src_port = self.get_link2port(path[-2], path[-1])[1]
191           dst_port = None
192 
193           for key in self.access_table.keys():
194             if flow_info[2] == self.access_table[key]:
195               dst_port = key[1]
196               break
197 
198           actions.append(parser.OFPActionOutput(dst_port))
199           match = parser.OFPMatch(
200             in_port=src_port,
201             ip_dscp=dscp,
202             eth_type=flow_info[0],
203             ipv4_src=flow_info[1],
204             ipv4_dst=flow_info[2])
205           self.add_flow(
206             datapath, 1, match, actions, idle_timeout=idle_time, hard_timeout
                =hard_time)
207 
208           # first pkt_out
209           actions = []
210           actions.append(parser.OFPActionOutput(out_port))
211           msg_data = None
212           if buffer_id == ofproto.OFP_NO_BUFFER:
213             msg_data = data
214 
215           out = parser.OFPPacketOut(
216             datapath=datapath_first, buffer_id=buffer_id,
217             data=msg_data, in_port=in_port, actions=actions)
218           datapath_first.send_msg(out)
219 
220           # last pkt_out
221           actions = []
222           actions.append(parser.OFPActionOutput(dst_port))
223           msg_data = None
224           if buffer_id == ofproto.OFP_NO_BUFFER:
225             msg_data = data
226 
227           out = parser.OFPPacketOut(
```

```
228         datapath=datapath, buffer_id=buffer_id,
229         data=msg_data, in_port=src_port, actions=actions)
230       datapath.send_msg(out)
231 
232     else:  # src and dst on the same
233       out_port = None
234       actions = []
235       for key in self.access_table.keys():
236         if flow_info[2] == self.access_table[key]:
237           out_port = key[1]
238           break
239 
240       actions.append(parser.OFPActionOutput(out_port))
241       match = parser.OFPMatch(
242         in_port=in_port,
243         ip_dscp=dscp,
244         eth_type=flow_info[0],
245         ipv4_src=flow_info[1],
246         ipv4_dst=flow_info[2])
247       self.add_flow(
248         datapath_first, 1, match, actions,
249         idle_timeout=idle_time, hard_timeout=hard_time)
250       msg_data = None
251       if buffer_id == ofproto.OFP_NO_BUFFER:
252         msg_data = data
253 
254       out = parser.OFPPacketOut(
255         datapath=datapath_first, buffer_id=buffer_id,
256         data=msg_data, in_port=in_port, actions=actions)
257       datapath_first.send_msg(out)
258 
259   def get_host_location(self, host_ip):
260     for key in self.access_table:
261       if self.access_table[key] == host_ip:
262         return key
263     self.logger.debug("%s location is not found." % host_ip)
264     return None
265 
266   def get_link2port(self, src_dpid, dst_dpid):
267     if (src_dpid, dst_dpid) in self.link_to_port:
268       return self.link_to_port[(src_dpid, dst_dpid)]
269     else:
270       self.logger.debug("Link to port is not found.")
271       return None
272 
273   def stats_req(self, datapath):
274     self.logger.debug('send stats request: %016x', datapath.id)
275     ofproto = datapath.ofproto
276     parser = datapath.ofproto_parser
277     req = parser.OFPFlowStatsRequest(datapath)
278     datapath.send_msg(req)
279     req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
280     datapath.send_msg(req)
```

```
281
282   @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
283   def _packet_in_handler(self, ev):
284     msg = ev.msg
285     datapath = msg.datapath
286     ofproto = datapath.ofproto
287     parser = datapath.ofproto_parser
288     in_port = msg.match['in_port']
289     pkt = packet.Packet(msg.data)
290
291     eth_type = pkt.get_protocols(ethernet.ethernet)[0].ethertype
292     arp_pkt = pkt.get_protocol(arp.arp)
293     ip_pkt = pkt.get_protocol(ipv4.ipv4)
294     lldp_pkt = pkt.get_protocol(lldp.lldp)
295
296     if isinstance(arp_pkt, arp.arp):
297       arp_src_ip = arp_pkt.src_ip
298       arp_dst_ip = arp_pkt.dst_ip
299       self.network_aware.register_access_info(datapath.id, in_port,
          arp_src_ip)
300
301       result = self.get_host_location(arp_dst_ip)
302       if result:  # host record in access table.
303         datapath_dst, out_port = result[0], result[1]
304         actions = [parser.OFPActionOutput(out_port)]
305         datapath = self.datapaths[datapath_dst]
306
307         out = parser.OFPPacketOut(
308           datapath=datapath,
309           buffer_id=ofproto.OFP_NO_BUFFER,
310           in_port=ofproto.OFPP_CONTROLLER,
311           actions=actions, data=msg.data)
312         datapath.send_msg(out)
313       else:     # access info is not existed. send to all host.
314         for dpid in self.access_ports:
315           for port in self.access_ports[dpid]:
316             if (dpid, port) not in self.access_table.keys():
317               actions = [parser.OFPActionOutput(port)]
318               datapath = self.datapaths[dpid]
319               out = parser.OFPPacketOut(
320                 datapath=datapath,
321                 buffer_id=ofproto.OFP_NO_BUFFER,
322                 in_port=ofproto.OFPP_CONTROLLER,
323                 actions=actions, data=msg.data)
324               datapath.send_msg(out)
325
326     if isinstance(ip_pkt, ipv4.ipv4):
327       time1=time.time()
328       ip_src = ip_pkt.src
329       ip_dst = ip_pkt.dst
330       tos = ip_pkt.tos
331
332       try:
```

```
333         a = self.flow_db.get((ip_src, ip_dst, tos))[1]    # check queue_id
              in flow_db
334       except TypeError:
335         a = None
336       if a is not None:    #Flow already Exists
337         return
338
339       result = None
340       src_sw = None
341       dst_sw = None
342
343       src_location = self.get_host_location(ip_src)
344       dst_location = self.get_host_location(ip_dst)
345
346       if src_location:
347         src_sw = src_location[0]
348
349       if dst_location:
350         dst_sw = dst_location[0]
351
352       if src_sw==dst_sw:  #return if sender-sink belong to same switch
353         flow_info = (eth_type, ip_src, ip_dst, in_port)
354         path=[src_sw]
355         print "same sw"
356         self.install_flow(path, flow_info, msg.buffer_id, msg.data)
357         return
358
359       bw = qos.tos_to_bw(tos) # map tos to bw
360       if tos==64:
361         result = qos.widest_shortest_path(self.graph, src_sw, dst_sw, tos
              )
362       else:
363         result = qos.shortest_path(self.graphshort, src_sw, dst_sw)
364       path = result
365       if result=="NEB64":
366         dscp=qos.tos_to_dscp(tos)
367         actions = []
368         datapath_first = self.datapaths[src_sw]
369         match = parser.OFPMatch(
370           in_port=in_port,
371           ip_dscp=dscp,
372           eth_type=eth_type,
373           ipv4_src=ip_src,
374           ipv4_dst=ip_dst)
375         self.add_flow(datapath_first,
376               1, match, actions, idle_timeout=self.idle_time,
                    hard_timeout=self.hard_time)
377         return
378
379       if result:
380         self.path_calc_counter +=1
381         print "calc counter", self.path_calc_counter, " bw: ", bw, ", tos
              : ", tos
```

```
382         flow_info = (eth_type, ip_src, ip_dst, in_port)
383         src_sw_out_port = self.network_aware.get_port_from_link(src_sw,
                path[1])[0]
384         if a is None:   #flow doesn't exists yet
385           unused_queue=qos.check_unused_queue(self.queue_db[(src_sw,
                  src_sw_out_port)])    #find unused queue
386           self.flow_db[(ip_src, ip_dst, tos)] = (path, unused_queue, bw,
                  src_sw, src_sw_out_port) #update flow_db
387           int_queues=[]
388           if unused_queue==None and tos==64:  #if queue is full
389             logging.critical("not enough queue %s %d"
390                     % (self.queue_db[(src_sw, src_sw_out_port)], time.
                            time()))
391             return
392           if tos==64:  #for QoS flow update queue_db
393             self.queue_db[(src_sw, src_sw_out_port)][unused_queue]=1   #
                    update queue_db (flag)
394             qos.update_graph_reserve_bw(self.graph, path, tos)
395             if len(path)>2: #update intermediate queue if forwarded using
                    more than 2 hops
396               for i in range(2,len(path)):
397                 sw=path[i-1]
398                 via_port = self.network_aware.get_port_from_link(path[i
                        -1],path[i])[0]
399                 cur_rate=self.int_queue_db[(sw, via_port)]
400 
401                 selected_int=qos.check_unused_int_queue(self.
                        int_queue_list[(sw, via_port)])
402                 int_queues.append((sw,selected_int))
403                 self.int_queue_list[(sw, via_port)][selected_int]=str(
                        ip_src+ip_dst)
404           self.install_flow(path, flow_info, msg.buffer_id, msg.data, tos
                  , unused_queue, int_queues)
405 
406       else:
407         self.network_aware.get_topology(None)    # Refresh the topology
                database.
408       self.logger.info("-PATH[%s --> %s],%s" %(ip_src, ip_dst, path))
```

network_aware.py

```
1  import logging
2  import struct
3  from operator import attrgetter
4  import qos
5  import copy
6  
7  from ryu.base import app_manager
8  from ryu.controller import ofp_event
9  from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
10 from ryu.controller.handler import CONFIG_DISPATCHER
11 from ryu.controller.handler import set_ev_cls
12 from ryu.ofproto import ofproto_v1_3
```

```
13 from ryu.lib import hub
14 from ryu.topology import event, switches
15 from ryu.topology.api import get_switch, get_link
16
17 SLEEP_PERIOD = 2
18 IS_UPDATE = True
19 queue_num=4
20
21 class Network_Aware(app_manager.RyuApp):
22   OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
23   _NAME = 'network_aware'
24
25  def __init__(self, *args, **kwargs):
26     super(Network_Aware, self).__init__(*args, **kwargs)
27     self.name = "Network_Aware"
28     self.topology_api_app = self
29     self.link_to_port = {}
30     self.access_table = {}
31     self.switch_port_table = {}  # dpid->port_num
32     self.access_ports = {}
33     self.interior_ports = {}
34     self.outer_ports = {}
35     self.graph = {}
36     self.graphshort = {}
37     self.shortest_graph = {}
38     self.pre_link_to_port = {}
39     self.pre_graph = {}
40     self.pre_access_table = {}
41     self.port_qos = {}
42     self.discover_thread = hub.spawn(self._discover)
43     self.queue_db = {}
44     self.int_queue_db = {}
45     self.int_queue_list = {}
46
47   def _discover(self):
48     i = 0
49     self.show_topology()
50     while True:
51       if i == 5:
52         self.get_topology(None)
53         i = 0
54       hub.sleep(SLEEP_PERIOD)
55       i = i + 1
56
57   @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
58   def switch_features_handler(self, ev):
59     datapath = ev.msg.datapath
60     ofproto = datapath.ofproto
61     parser = datapath.ofproto_parser
62     msg = ev.msg
63     self.logger.info("switch:%s connected", datapath.id)
64     match = parser.OFPMatch()
65     actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
```

```
66                          ofproto.OFPCML_NO_BUFFER)]
67      self.add_flow(datapath, 0, match, actions)
68
69    def add_flow(self, dp, p, match, actions, idle_timeout=0, hard_timeout
        =0):
70      ofproto = dp.ofproto
71      parser = dp.ofproto_parser
72
73      inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
74                          actions)]
75
76      mod = parser.OFPFlowMod(datapath=dp, priority=p,
77                  idle_timeout=idle_timeout,
78                  hard_timeout=hard_timeout,
79                  match=match, instructions=inst)
80      dp.send_msg(mod)
81
82    def get_switches(self):
83      return self.switches
84
85    def get_links(self):
86      return self.link_to_port
87
88    def get_port_from_link(self, src_sw, dst_sw):
89      #get port number from link of two switches
90      return self.link_to_port[(src_sw, dst_sw)]
91
92    def get_graph(self, link_list):
93      for src in self.switches:
94        for dst in self.switches:
95          self.graph.setdefault(src, {dst: 0})
96          self.graphshort.setdefault(src, {dst: float('inf')})
97          if src == dst:
98            self.graph[src][src] = float('inf')
99            self.graphshort[src][src] = 0
100         elif (src, dst) in link_list:
101           try:
102             weight = self.graph[src][dst]   # check queue_id in flow_db
103           except KeyError:
104             weight = None
105           if not weight:  #avoid update at switch_enter event
106             self.graph[src][dst] = qos.capacity
107             self.graphshort[src][dst] = 1
108         else:
109           self.graph[src][dst] = 0
110           self.graphshort[src][dst] = float('inf')
111
112     self.shortest_graph = copy.deepcopy(self.graph)
113     return (self.graph, self.shortest_graph, self.graphshort)
114
115   def create_port_map(self, switch_list):
116     for sw in switch_list:
117       dpid = sw.dp.id
```

```
118       self.switch_port_table.setdefault(dpid, set())
119       self.interior_ports.setdefault(dpid, set())
120       self.access_ports.setdefault(dpid, set())
121
122       for p in sw.ports:
123         self.switch_port_table[dpid].add(p.port_no)
124       self.switch_port_table[dpid].add(4294967294)
125
126   def create_interior_links(self, link_list):
127     for link in link_list:
128       src = link.src
129       dst = link.dst
130       self.link_to_port[(src.dpid, dst.dpid)] = (src.port_no, dst.port_no
            )
131       if link.src.dpid in self.switches:
132         self.interior_ports[link.src.dpid].add(link.src.port_no)
133       if link.dst.dpid in self.switches:
134         self.interior_ports[link.dst.dpid].add(link.dst.port_no)
135
136   def create_access_ports(self):
137     for sw in self.switch_port_table:
138       self.access_ports[sw] = self.switch_port_table[
139         sw] - self.interior_ports[sw]
140
141   def create_outer_port(self):
142     pass
143
144   @set_ev_cls(event.EventSwitchEnter)
145   def get_topology_sw_enter(self, ev):
146     #create qos and queue1 at switch enter event
147     msg = ev.switch.to_dict()
148
149     for port in msg['ports']:
150       dpid=int("0x"+msg['dpid'],0)
151       self.queue_db.setdefault((dpid,int(port['port_no'])), {1:None})
152       self.int_queue_list.setdefault((dpid,int(port['port_no'])), {5:None
            })
153
154       for i in range(1,queue_num):
155         self.queue_db[(dpid,int(port['port_no']))].update({i:None})
156         self.int_queue_db[(dpid,int(port['port_no']))]=0
157
158       for i in range(5,15):
159         self.int_queue_list[(dpid,int(port['port_no']))].update({i:None})
160
161     switch_list = get_switch(self.topology_api_app, None)
162     self.create_port_map(switch_list)
163     self.switches = self.switch_port_table.keys()
164     links = get_link(self.topology_api_app, None)
165     self.create_interior_links(links)
166     self.create_access_ports()
167     self.get_graph(self.link_to_port.keys())
168
```

```
169     events = [#event.EventSwitchEnter,
170             event.EventSwitchLeave, event.EventPortAdd,
171             event.EventPortDelete, event.EventPortModify,
172             event.EventLinkAdd, event.EventLinkDelete]
173 
174     @set_ev_cls(events)
175     def get_topology(self, ev):
176       switch_list = get_switch(self.topology_api_app, None)
177       self.create_port_map(switch_list)
178       self.switches = self.switch_port_table.keys()
179       links = get_link(self.topology_api_app, None)
180       self.create_interior_links(links)
181       self.create_access_ports()
182       self.get_graph(self.link_to_port.keys())
183 
184     def register_access_info(self, dpid, in_port, ip):
185       if in_port in self.access_ports[dpid]:
186         if (dpid, in_port) in self.access_table:
187           if ip != self.access_table[(dpid, in_port)]:
188             self.access_table[(dpid, in_port)] = ip
189         else:
190           self.access_table[(dpid, in_port)] = ip
191 
192     def show_topology(self):
193       switch_num = len(self.graph)
194       if self.pre_graph != self.graph or IS_UPDATE:
195         print "---------------------Topo Link---------------------"
196         print '%10s' % ("switch"),
197         for i in xrange(1, switch_num + 1):
198           print '%10d' % i,
199         print ""
200         for i in self.graph.keys():
201           print '%10d' % i,
202           for j in self.graph[i].values():
203             print '%10.0f' % j,
204           print ""
205         self.pre_graph = self.graph
206 
207       if self.pre_link_to_port != self.link_to_port or IS_UPDATE:
208         print "---------------------Link Port---------------------"
209         print '%10s' % ("switch"),
210         for i in xrange(1, switch_num + 1):
211           print '%10d' % i,
212         print ""
213         for i in xrange(1, switch_num + 1):
214           print '%10d' % i,
215           for j in xrange(1, switch_num + 1):
216             if (i, j) in self.link_to_port.keys():
217               print '%10s' % str(self.link_to_port[(i, j)]),
218             else:
219               print '%10s' % "No-link",
220           print ""
221         self.pre_link_to_port = self.link_to_port
```

```
222
223     if self.pre_access_table != self.access_table or IS_UPDATE:
224       print "----------------Access Host------------------"
225       print '%10s' % ("switch"), '%12s' % "Host"
226       if not self.access_table.keys():
227         print "  NO found host"
228       else:
229         for tup in self.access_table:
230           print '%10d:  ' % tup[0], self.access_table[tup]
231       self.pre_access_table = self.access_table
```

qos.py

```
1  import subprocess
2  import copy
3  import logging
4  import operator
5  import network_aware
6  import time
7  capacity = 100000000
8  resv_bw = 30000000
9
10 def port_translation(port):
11   switcher = {
12     1:0,
13     2:1,
14     3:2,
15     4:3,
16     5:5,
17   }
18   return switcher.get(port, 0)
19
20 def clean_arp(datapath):
21   server="node0"+str(datapath)
22   if datapath==10:
23     server="node10"
24   shell_cmd="sshpass -p ****** ssh hedi@%s sudo ip -s -s neigh flush all"
       %(server)
25   vsctl(shell_cmd)
26
27 def check_unused_queue(queue_db):
28   for i in queue_db:
29     if queue_db[i] == None:
30       return i
31   return
32
33 def check_unused_int_queue(int_queue_list):
34   for i in range(5,15):
35     if int_queue_list[i] == None:
36       return i
37   return
38
39 def tos_to_dscp(tos):
```

```
40   if tos==0: dscp=0
41   elif tos==64: dscp=16
42   elif tos==192: dscp=48
43   return dscp
44
45 def dscp_to_tos(dscp):
46   if dscp==0: tos=0
47   elif dscp==16: tos=64
48   elif dscp==48: tos=192
49   return tos
50
51 def tos_to_bw(tos):
52   # map tos to bw reservation
53   switcher = {
54     64: resv_bw,
55   }
56   return switcher.get(tos, 0)
57
58 def update_graph(graph, flow_db):
59   for x in graph:
60     for y in graph:
61       if x == y:
62         graph[x][y] = float('inf')
63       elif graph.get(x).get(y) == 0:
64         graph[x][y] = 0
65       elif get_rsv_bw(flow_db, x, y) is None:
66         graph[x][y] = capacity - 0
67       else:
68         graph[x][y] = capacity - get_rsv_bw(flow_db, x, y)
69
70 def update_graph_release_bw(graph,path, bw):
71   for i in range(1,len(path)):
72     sw1=path[i-1]
73     sw2=path[i]
74     graph[sw1][sw2] = graph[sw1][sw2]+bw
75
76 def update_graph_reserve_bw(graph,path, tos):
77   bw=tos_to_bw(tos)
78   for i in range(1,len(path)):
79     sw1=path[i-1]
80     sw2=path[i]
81     graph[sw1][sw2] = graph[sw1][sw2]-bw
82
83 def is_flow_exists(flow_db, ip_src, ip_dst):
84   for key in flow_db.keys():
85     if key[0]==ip_src and key[1]==ip_dst and flow_db[key][0] != [0]:
86       print "flow exists, flow_db:", flow_db
87       return 1
88
89 def widest_shortest_path(graph, source, dest, tos):
90   bw = tos_to_bw(tos)
91   graphcopy = copy.deepcopy(graph)
92
```

```
93     for key, value in graphcopy.iteritems():
94       for subkey, subvalue in graphcopy[key].iteritems():
95         if key == subkey:
96           graphcopy[key][subkey] = 0
97         elif subvalue < bw:
98           graphcopy[key][subkey] = float('inf')
99         elif subvalue == 0:
100          graphcopy[key][subkey] = float('inf')
101        else:
102          graphcopy[key][subkey] = 1
103    d = {}
104    p = {}
105    m = []
106    q = {}
107    c = {}
108
109    for r in list(graphcopy.keys()): #range(1, len(graphcopy) + 1):
110      d[r] = float('inf')
111    for r in list(graphcopy.keys()): #range(1, len(graph) + 1):
112      c[r] = 0
113    d[source] = 0  # the capacity of the best path from source to v
114    q[source] = 0  # insert source as initial candidate
115    c[source] = float('inf')
116
117    while q:
118      a={}
119      b={}
120      z={}
121      for key in q.iterkeys():
122        a[key]=d[key]
123        b[key]=c[key]
124      min_a = min(a.itervalues())  # best candidate in q
125      for key, value in a.iteritems():
126        if value == min_a:
127          z[key]=b[key]
128      x = max(z.iteritems(), key=operator.itemgetter(1))[0]
129      m.append(x)
130      q.pop(x)
131
132      for key, value in graphcopy[x].iteritems():
133        if value != 0 and value != float('inf') and key not in m:  # for
             each x's neighbor that is not in M:
134          minvalue = min(c[x], graph[x][key])
135          if d[x] + graphcopy[x][key] < d.get(key):
136            d[key] = d[x] + graphcopy[x][key]
137            c[key] = minvalue
138            p[key] = x
139          if not q.get(key):
140            q[key] = value
141    if not p.get(dest) and tos!=64:
142      logging.critical("NEB. install. %s to %s. tos: %d. %d", source, dest,
             tos, time.time())
143      return "NEB0"
```

```
144     if not p.get(dest) and tos==64:
145       logging.critical("NEB. reject. %s to %s. tos: %d. %d", source, dest,
            tos, time.time())
146       return "NEB64"
147     path=[]
148     while 1:
149       path.append(dest)
150       if dest==source: break
151       dest = p.get(dest)
152     path.reverse()
153     return path
154 
155 
156   def shortest_path(graphshort, source, dest):
157     graphcopy2 = copy.deepcopy(graphshort)
158     d = {}
159     p = {}
160     m = []
161     q = {}
162 
163     for r in list(graphcopy2.keys()):
164       d[r] = float('inf')
165     d[source] = 0  # the capacity of the best path from source to v
166     q[source] = 0  # insert source as initial candidate
167 
168     while q:
169       a={}
170       for key in q.iterkeys():
171         a[key]=d[key]
172       x = min(a.iteritems(), key=operator.itemgetter(1))[0]  # best
            candidate in q
173       m.append(x)
174       q.pop(x)
175       for key, value in graphcopy2[x].iteritems():
176         if value != 0 and value != float('inf') and key not in m:  # for
              each x's neighbor that is not in M:
177           if d[x] + graphcopy2[x][key] < d.get(key):
178             d[key] = d[x] + graphcopy2[x][key]
179             p[key] = x
180           if not q.get(key):
181             q[key] = value
182     if not bool(p):
183       print " path not found"
184       return
185     path=[]
186     while 1:
187       path.append(dest)
188       if dest==source: break
189       dest = p.get(dest)
190     path.reverse()
191     return path
```