

Guaranteeing service level agreements in a congested IP-network

Networks and Distributed Systems

[illegible]

Aalborg University
Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg



AALBORG UNIVERSITY

STUDENT REPORT

Department of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
<http://es.aau.dk>

Title:

Flow-based Dynamic Queue Selector
- *Guaranteeing service level agreements in a congested IP-network*

Theme:

Master thesis

Project Period:

4th semester
Networks and Distributed Systems
February 3rd – June 4th, 2014

Project Group:

14gr1022

Participants:

Niels Fristrup Andersen
Mads Holdgaard Vejøl

Supervisors:

Jimmy Jessen Nielsen
Andrea Fabio Cattoni

Copies: 3**Page Numbers:** 78**Date of Completion:** June 2, 2014**Abstract:**

This work presents a solution for increasing link utilisation in a backbone while adhering to Service Level Agreements (SLAs) for different service types using the new networking paradigm called Software-Defined Networking (SDN). The work is based on emulating a network, using lightweight virtualisation of individual network hosts and virtual switches. Traffic is generated synthetically according to three different service types; Video, Voice and Best Effort, where each service type has a set of SLA parameters, consisting of Throughput, End-to-End delay and Packet loss rate. A solution is proposed in the form of a control loop, based on known methods from Cognitive Networks. The control loop dynamically moves flows between different queues on an output port in a switch and drops flows, in order to ensure that the SLA parameters are fulfilled for each individual flow. Results for the proposed solution show that it is capable of guaranteeing SLAs on a per flow basis for link utilisations of up to 80 [%] for the flows with the most demanding SLAs.

Preface

This report documents a master thesis project, for the Networks and Distributed Systems programme at the Department of Electronic Systems, Aalborg University.

The purpose of this work is two-fold; first it must *explore* the possibilities that the newest technologies in communication networks bring, secondly it must *apply* this new knowledge, in order to innovate current network management solutions.

The performed work is based on emulations of a real network, using standard communication protocols, such that the developed control solution is prepared for deployment on physical network equipment.

This report also contains a DVD in the back, which includes all source code for the project. The content is separated into three folders; *TestBed/*, *Floodlight/* and *OpenDayLight/*. *TestBed/* includes all scripts and source code used for setting up the test bed and executing tests, as well as MATLAB scripts for parsing test results.

Floodlight/ contains the Floodlight controller as well as source code for the module developed for that controller.

OpenDayLight/ contains the OpenDayLight controllers as well as the source code for the module created for this specific controller.

Aalborg University, June 2, 2014

Niels Frstrup Andersen
<nfan10@student.aau.dk>

Mads Holdgaard Vejøl
<mvejlo09@student.aau.dk>

Contents

1	Introduction	1
1.1	Problem statement	3
2	Preliminary Analysis	5
2.1	General usage of the Internet	5
2.2	Guaranteeing a good experience	7
2.2.1	Voice	7
2.2.2	Video	8
2.2.3	Best Effort	9
2.2.4	Quality of Service requirements for Service Level Agreements	9
2.3	Use case	10
2.3.1	Architectural overview	10
2.3.2	Service usage	11
2.3.3	Use case performance measurements	12
3	Analysis	15
3.1	Software-Defined Networking	15
3.1.1	SDN in relation to Cognitive Networks	15
3.1.2	What is SDN?	18
3.1.3	SDN controllers	19
3.2	OpenFlow	19
3.2.1	Flow Table	20
3.2.2	Match	21
3.2.3	Action	22
3.2.4	Flow statistics	22
3.3	System as a feedback loop	24
3.3.1	Network feedback	24
3.3.2	Network actions	27
4	Design	29
4.1	The controller	29
4.1.1	Controller modules	29
4.2	Definition of a flow	31
4.3	Flow-based Dynamic Queue Selector module design	32
4.4	State Maintainer design	34
4.4.1	OpenFlow statistics	34

4.4.2	Delay estimation	35
4.4.3	Active flows	35
4.5	Decision Logic design	37
4.5.1	Drop utility function	40
4.5.2	Delay utility function	41
5	Implementation	43
5.1	Virtual switches	43
5.2	Dynamic queues	43
5.3	Delay estimation	44
6	Evaluation	45
6.1	Test bed	45
6.2	Test method	46
6.2.1	Test parameters	46
6.2.2	Generating traffic	47
6.2.3	Create topology and start traffic	49
6.3	Results	51
6.3.1	Utilisation: 70 [%]	51
6.3.2	Utilisation: 80 [%]	55
7	Conclusion	61
	Bibliography	63
A	Result tables	65
B	Resource allocation techniques	71
C	Generating traffic	75

Used acronyms

API Application Programmers Interface

AF Assured Forwarding

CDF Cumulative Distribution Function

DiffServ Differentiated services

DSCP DiffServ Code Point

E2E End-To-End

EF Expedited Forwarding

FDQS Flow-based Dynamic Queue Selector

FE Forwarding Element

FEC Forwarding Equivalence Class

IDT Inter-Departure Time

IoT Internet of Things

IP Internet Protocol

IntServ Integrated services

ISP Internet Service Provider

LER Label Edge Router

LSP Label Switched Path

LSR Label Switched Router

MAC Media Access Control

MPLS Multi-Protocol Label Switching

MTU Maximum Transmission Unit

NIC Network Interface Card

OODA Observe Orient Decide and Act

OS Operating System

OVS Open vSwitch
PHB Per Hop Behaviour
QoS Quality of Service
RSVP Resource reSerVation Protocol
RTT Round-Trip-Time
SDN Software-Defined Networking
SLA Service Level Agreement
TCP Transmission Control Protocol
TE Traffic Engineering
VoIP Voice over IP

1 Introduction

This chapter introduces and motivates the problem, of having a fixed amount of resources in the network. The concept of SDN is introduced as a new method for approaching the problem, which is stated in the end of the chapter.

Today's modern communication networks have made the world globally interconnected, where information can be transported over longer geographical distances in a matter of seconds. This connectivity constantly inspires and leads to new applications and services that provide new opportunities, which again pushes the boundaries for what the Internet users expect and demand.

Users that are having a good experience with a service, is an ideal situation for all concerned; the users are satisfied and so is the service provider. However, this scenario relies on the performance of the connectivity between the users and the service. If the communication is exposed to long waiting times or information losses, neither the users nor the service providers are satisfied. This increases the pressure on the network providers in that both the users and service providers implicitly expect the network to perform whenever a user requests a service. Hence, both users and service providers demand a certain level of Quality of Service (QoS).

According to estimations by Cisco [2013], the number of Internet-connected objects as of 2013 was 10 billion. These objects account for users, places, processes and things in general, also conceptually referred to as the Internet of Things (IoT). In 2020, it is expected that this number will be up to 50 billion Internet-connected objects.

Since the underlying network does not provide unlimited resources, simultaneous service requests may consume all the available capacity, thus leading to degradations in communication performance. With this increasing use of the Internet as a converged communication platform, the network providers need to adapt their infrastructure accordingly, in order to satisfy their customers.

The simple approach to solving this problem, is to ensure the network is over-provisioned regarding capacity. No need for the network providers to deal with other things, than monitoring the amount of available resources and buy some more before running out. The advantage of that is the avoidance of complex QoS set-ups with expensive and sophisticated hardware and the accompanying management of them, and instead posting the money in more resources. However, sometimes it might be practically impossible to upgrade the pool of resources, for instance if it would involve trenching down new cables, which is a

rather expensive process. In such cases, the network provider would need some kind of service prioritisation management anyway.

Even though, the costs of more bandwidth seem to be decreasing in the years to come [Cisco, 2013], the question is; will the strategy of over-provisioning remain sustainable in the future, with the above-mentioned estimated growth of connected objects? Indeed the strategy is simple, but it is a waste of valuable resources, which is not ideal from both an engineering and an ecological perspective.

If the network providers were able to improve the efficiency of their current network deployment, without disturbing the users' experience of the services, then they might be able to lower the costs of cable trenching and network equipment, while being more friendly to the environment.

But how to accomplish that? One possibility is SDN. SDN is a new computer networking paradigm, that has emerged and matured in recent years, which enables network operators, to develop new ways of managing and controlling network resources.

SDN introduces a layered abstraction of the network, where the data plane is separated from the control plane, as depicted in figure 1.1.

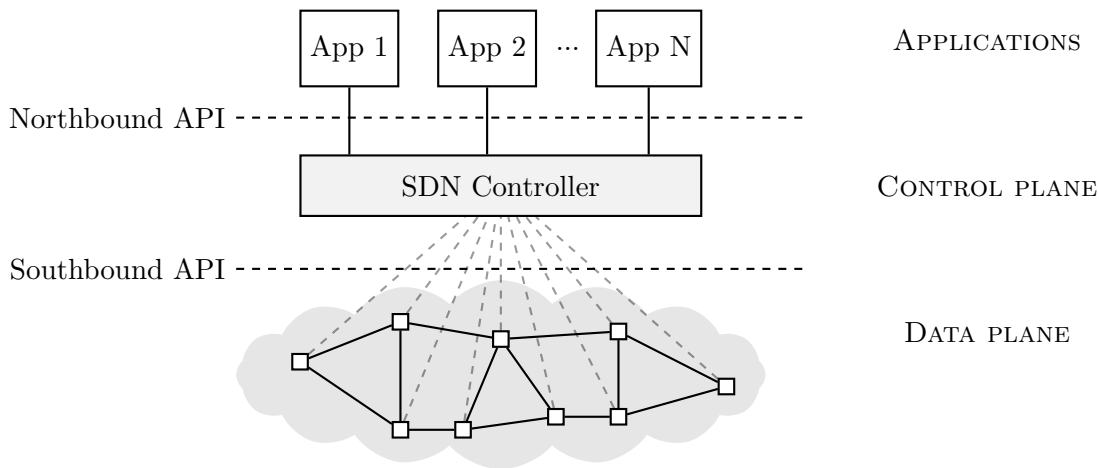


Figure 1.1: The layered abstraction of Software-Defined Networking (SDN).

This abstraction thereby decouples the forwarding hardware from the control software, which means that the control mechanism can be extracted from the Forwarding Elements (FEs) and logically centralised in the network, on the SDN Controller. The controller creates an abstraction of the underlying network, and thereby provides an interface to the higher-layer applications. These can then interact with the controller, hence indirectly control the packet forwarding in the FEs. This centralised control-structure makes it possible to implement a coordinated behaviour of the network, making it more deterministic.

By introducing layers and using standardised interfaces both north- and southbound to the controller, SDN brings a modular concept that enables the network to be developed independently on each of the three layers. Most notably, it creates the opportunity for

others than networking hardware vendors, to innovate and develop control software and applications for the network.

The SDN controller can be placed on commodity computing hardware while providing an open software environment, in contrast to the proprietary control plane software running on the FEs in non-SDN networks. This means that the FEs could be reduced to "dumb" packet forwarders, which would reduce the complexity and thereby the cost of data plane hardware. Furthermore, it transforms the task of configuring a network, from being decentralised use of vendor-specific console syntaxes, into a programmable task for a software engineer, who can deploy the software to the whole network, just by interfacing with the logically centralised controller.

Being able to program the network using standardised Application Programmers Interfaces (APIs), ultimately enables the network operators to automate tasks, hence improving the manageability. It also makes the network providers more adaptable to business needs, since network features can now be independently developed, without the need to wait for the hardware vendors to provide the feature. But it also makes it possible to create custom implementations that supports a specific need, thereby avoiding unnecessary complexity in the control software.

SDN is interesting, because it introduces a new networking architecture, which enables the development of networking algorithms that are based on a centralised control structure of the network. This centralised structure and the possibilities it brings, is considerably different from current networks, where the logic is generally distributed amongst all of the network devices.

1.1 Problem statement

As motivated above, the case of having a fixed amount of resources in the network, is a problem which network providers and users most likely will be faced with sooner or later, if not already. The task for the network operator is then to use the available resources and make sure, that the negotiated SLAs are still met. If otherwise the SLAs are not met, the users may experience a bad QoS, that could result in a lost customer.

This project takes a new approach to QoS and network resource efficiency, by applying the concept of SDN, in order to explore and demonstrate the potential of next-generation network control and -management.

This leads to the following problem statement:

"How to increase link utilisation in a Software-Defined Network, while adhering to constraints given by Quality of Service parameters for different service types?"

The following chapter will open the problem, by exploring the service types of today's Internet and their associated QoS parameters.

2 Preliminary Analysis

This chapter serves as an initial exploration of the problem statement. First, the different service types and their traffic characteristics, in terms of Quality of Service requirements, are defined. Afterwards a use case scenario is presented, in order to test and demonstrate the specified problem.

2.1 General usage of the Internet

This section seeks to open the problem statement, presented in section 1.1, by exploring which types of services that exist in Internet Protocol (IP)-based communication networks today. The goal of this analysis is to come up with a limited set of categories, that covers the most widely used services on the Internet.

To determine how the traffic is distributed among the different services, Internet usage statistics are explored. The Global Internet Phenomena Report by Sandvine [2013] gives a strong indication of the current development in Internet usage.

Rank	Upstream		Downstream		Aggregate	
	Application	Share	Application	Share	Application	Share
1	BitTorrent	48.10%	YouTube	28.73%	YouTube	24.21%
2	YouTube	7.12%	HTTP	15.64%	BitTorrent	17.99%
3	HTTP	5.74%	BitTorrent	10.10%	HTTP	13.59%
4	Skype	4.96%	Facebook	4.94%	Facebook	4.65%
5	Facebook	3.54%	Netflix	3.45%	Netflix	3.33%
6	Netflix	2.83%	MPEG - Other	3.10%	MPEG - Other	2.57%
7	SSL	2.47%	RTMP	2.82%	RTMP	2.42%
8	eDonkey	1.12%	Flash Video	2.56%	Skype	2.32%
9	Dropbox	1.12%	SSL	1.91%	Flash Video	2.16%
10	RTMP	0.85%	PutLocker	1.25%	SSL	2.03%
		77.83%		73.23%		75.25%




Figure 2.1: Table of internet usage statistics for Europe [Sandvine, 2013].

From the table in figure 2.1 it is clear that video on-demand streaming services such as YouTube and Netflix, as well as other MPEG and Flash based services, are responsible for the majority of downstream data in Europe. Based on this, video streaming is an obvious category to focus on, because of its dominant role in Internet traffic, henceforth this category will be referred to as *Video*.

Further inspection of figure 2.1 reveals Skype traffic, which represents Voice over IP (VoIP) both with and without video. This is a traffic category that does not consume a large amount of bandwidth, compared to video streaming, but it is interesting because of VoIPs impact on regular telephony. Skype is responsible for approximately 55 billion minutes of international Skype-to-Skype calls compared to approximately 35 billion minutes of international calls using regular telephony, according to TeleGeography [2014]. The use of regular telephony for international calls has been decreasing for the last 3 years, while the use of Skype continues to grow. This illustrates the trend of how services converge onto the same platform.

Voice traffic will also present the network with different and more strict QoS requirements compared to the video category in terms of delay. Using a voice service implies that users interact with each other, hence the service is rather sensitive to delays and variations in the communication, due to the bi-directional nature of a teleconference or voice call. For instance, if the user must wait too long for the other user to respond, then the conversation can end up being awkward, thus the experience is affected.

The category includes teleconferences and calls with and without video, and will simply be referred to as *Voice*.

Finally, a category which covers service types based on explicit file transfers. This is services such as HTTP, FTP, BitTorrent, Dropbox, etc. Common for file transferring services is that they are designed to work under various network conditions, in order to deliver the expected outcome, namely that the file is transferred successfully. Hence, they are robust regarding network traffic conditions and not nearly as sensitive to varying network conditions as voice and video are. This category will be referred to as *Best Effort*.

This leads to three categories:

- Video
- Voice
- Best Effort

These three categories represent a simplified set of services, that are used on the Internet. They will be the basis for the traffic to study, when determining the requirements for the network, such that all users have a good experience when using any of the three service types.

Requirements for the network in each category, will be determined in the following section.

2.2 Guaranteeing a good experience

The goal is to guarantee that the users have a good experience with the services they use across a network. But how does a network guarantee a good experience? This is done by guaranteeing certain QoS requirements for the different traffic types.

The QoS requirements covers different aspects of packet delivery in a network. First of all, there can be requirements to the *End-To-End (E2E) delay* between a user and a service. The E2E delay is the time it takes from a packet is transmitted and until it arrives at the receiving end. The next QoS parameter is *packet loss*, which is the number of packets lost relatively to the total number of packets transmitted for the traffic flow. The final QoS parameter is *throughput*, which describes the average amount of bandwidth a service requires in order to give the user a good experience.

This leaves us with three QoS parameters that must be determined for each of the three categories presented in the previous section.

2.2.1 Voice

Voice applications need to be able to play streamed audio as close to real-time as possible, in order to give the user the best experience. Voice traffic in general is very sensitive to delay. Delay variations can be difficult to avoid, which is why, most voice applications use buffering mechanisms, to mitigate the effects of delay variations. However, buffers have the drawback, that they increase the E2E delay between users communicating, because data must be buffered up, such that they can mitigate the delay variations.

In terms of packet loss, voice traffic is very dependent on the coding scheme being used. Packet loss will in most cases result in a degradation of application performance and output quality. The performance degradation can be noticed through sound glitches because of the codec trying to compensate for missing packets, when retransmission is not used. Packet loss can also be noticed through increased E2E delay, or a need for more buffering, because retransmission is used for the lost packets. A low packet loss ratio is therefore desired.

The above is all general thoughts on QoS requirements for voice traffic, but it is necessary to determine specific QoS requirements. Both Szigeti and Hattingh [2004] and Chen et al. [2004] have similar QoS requirements for voice traffic, because both of them are based on the ITU-T (International Telecommunication Union) G.114¹ recommendations for QoS in communication systems. Both Szigeti and Hattingh [2004] and Chen et al. [2004] presents E2E delays between 100 and 150 [ms], where ITU-Ts recommendation is E2E delays below 150 [ms]. These E2E delay requirements represents the maximum tolerated values, i.e. the mouth-to-ear delay, which is the time it takes the voice signal to travel from the senders microphone to the receivers speaker, thus including signal processing delay at both ends. The network transmission goal must therefore be, to deliver a service such as voice as fast as possible, in order to retain a proper delay margin for signal processing. Therefore, a target E2E delay of 30 [ms] is selected as the requirement for voice traffic.

¹G.114 is a set of recommendations regarding transmission quality of a telephone connection for TRANSMISSION SYSTEMS AND MEDIA, DIGITAL SYSTEMS AND NETWORKS

Both sources agree that the packet loss rate should be less than 1 [%], which is selected as the targeted packet loss rate for voice.

According to Skype, 100 [Kb/s] is the recommended amount of throughput required to make a Skype voice call², while 300 [Kb/s] is needed to make a Skype video call. Szigeti and Hattingh [2004] recommend up to 384 [Kb/s] throughput per voice call. A throughput requirement of 300 [Kb/s] is therefore selected for the voice category.

Table 2.1 in section 2.2.4 shows the QoS requirements for voice traffic based on these sources.

2.2.2 Video

This category covers video streaming from on-demand services such as Netflix and YouTube, and does not include services such as video conferencing, which is considered a real-time video service. Video streaming applications uses buffering mechanisms, which makes the video streaming robust in terms of delay and delay variations as well as packet loss.

As a category, video will generally require a higher amount of throughput than voice, but does not have as high QoS requirements because it is not real-time traffic.

Szigeti and Hattingh [2004] presents the following QoS requirements for video streaming: 4 – 5 [s] E2E delay, 5 [%] packet loss rate. However Joseph and Chapman [2009] presents these QoS requirements for video streaming: 400 [ms] E2E delay, 1 [%] packet loss rate.

The two sources do not agree on the requirements for E2E delay, 400 [ms] is a bit low considering that this is not a real-time traffic category, 4 – 5 [s] on the other hand, is a very loose requirement.

Guaranteeing an E2E delay of 4 – 5 [s] means that a delay of up to 4 – 5 [s] is acceptable when starting a video stream. This may be tolerable, for the user, in relation to a movie with a duration of two hours. However it also means that a user would have to wait up to 4 – 5 [s] before e.g. the streaming of a YouTube clip can start, which is expected to have a considerable lower duration than a movie. An E2E delay of 1 [s] may be more appropriate in this case, as it ensures less waiting time when starting the streaming of a new video clip, while still not being as strict a delay requirement as what was presented for voice traffic. 1 [s] is therefore selected as the E2E delay QoS parameters for video.

The two sources slightly disagree on the tolerated packet loss rate for video streaming. Which of the two sources that is most correct most likely depends of the application in question. A packet loss rate of 5 [%] is therefore selected as one of the QoS parameters for video.

The throughput requirements for video streaming can be anywhere from less than 1 [Mb/s] to more than 20 [Mb/s], dependent on the codec and the resolution of the content. Netflix recommends between 1.5 [Mb/s] for the lowest quality of streaming and 7 [Mb/s] for their Super HD content. Based on this, a throughput requirement of 7 [Mb/s] is selected for the video category.

The QoS requirements for video is shown in table 2.1 in section 2.2.4.

²<https://support.skype.com/en/faq/fa1417/how-much-bandwidth-does-skype-need>

2.2.3 Best Effort

This category covers a broad range of network protocols. Creating a specific set of requirements for this category is therefore deemed unrealistic, however some general requirements can be made. The best effort category contains the HTTP protocol which is generally used for web browsing, which would suggest that some timeliness is desired for this category. Chen et al. [2004] has suggested that HTTP browsing traffic should not have an E2E delay above 400 [ms].

Packet drops will cause packets to be retransmitted when data is transferred using TCP, which ensures that all data reaches the user, but packet loss will increase the E2E delay because of the retransmission. Packet loss is therefore acceptable, but a too high loss rate will give the user a bad experience. The acceptable packet loss rate is therefore set to be 5 [%].

With respect to throughput, best effort traffic is considered robust in that, it works both at high or low throughput rates. Therefore it is set lower than video, but higher than voice traffic, thus a value of 2 [Mb/s] is selected as its throughput QoS requirement.

The selected QoS requirements for best effort traffic is presented in table 2.1 in section 2.2.4.

2.2.4 Quality of Service requirements for Service Level Agreements

Table 2.1 shows the QoS requirements for the three different traffic categories.

Service type	E2E delay [ms]	Packet loss [%]	Throughput [Mb/s]
Voice	< 30	< 1	0.3
Video	< 1000	< 5	7
Best effort	< 400	< 5	2

Table 2.1: QoS requirements for each of the three traffic categories.

These QoS requirements must be fulfilled by the network in order to guarantee a good experience when using the different services. The QoS requirements are also referred to as Service Level Agreements (SLAs) throughout this project documentation, since they represent the agreement between the users and services, of how the different services should be delivered.

2.3 Use case

In order to demonstrate the problem of guaranteeing QoS requirements, in a network with limited link capacity, this section presents a use case, which exemplifies the relation between link utilisation and E2E delay. The use case acts as the reference scenario throughout the work of this project.

Among current techniques for guaranteeing QoS in IP networks, there are Integrated services (IntServ), Differentiated services (DiffServ) and Multi-Protocol Label Switching (MPLS). IntServ suffers from not being scalable in larger backbone networks, due to its need for state information for every flow. DiffServ schedules packet transmission on a configured per-hop basis, hence it cannot guarantee E2E requirements. MPLS can set up tunnels throughout the network, in order to allocate bandwidth E2E, but only based on a simple counter value and not from a view of the global network state.

However, combining MPLS and DiffServ can provide a synergy, where MPLS handles the path selection and DiffServ determines how to schedule packets along the way. But this also introduces the need for further distributed state information to map between Label Switched Paths (LSPs) and output queue, which again will decrease the scalability aspect of the solution. Furthermore, the technique is still not aware of the global network state and cannot adapt to the current networking conditions.

The three techniques are all described in more detail in appendix B.

The use case is based on the assumption, that an Internet Service Provider (ISP) is creating one tunnel per user, with a single scheduling policy for the tunnel, in order to maintain scalability as discussed above. That is, no configured per-hop behaviour for different traffic types within the tunnel, hence aggregating all traffic from one user into a single queue.

To stress the fact that the network has limited link capacity, the network topology for the use case only has a single link in the backbone, thus no option for re-routing the traffic along another and less congested path. Combined with the assumption above, all traffic is transmitted via the backbone using a single output queue only, since a queue per user in the backbone is also not a viable solution in an ISP network.

These descriptions are used for creating the network topology, which is introduced in the following section.

2.3.1 Architectural overview

Figure 2.2 shows the architecture of the scenario investigated by the use case. This scenario is also referred to as the *static* scenario, in this project documentation. The network is deployed and emulated on a test bed, using the Mininet framework Lantz et al. [2010]. This set-up forms the basis for the implementation for this project, and is further described in chapter 6.

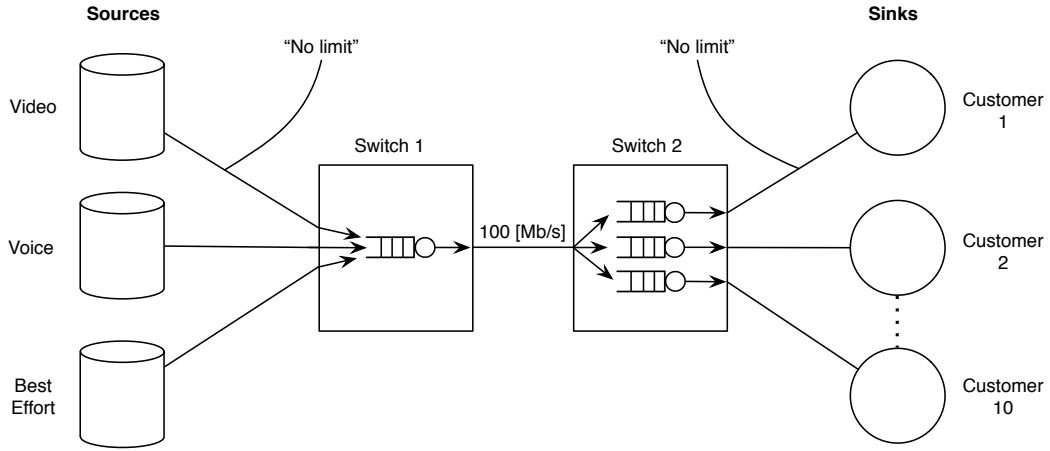


Figure 2.2: Use case architecture with IP switches interconnecting services and users. Also referred to as the *static* scenario.

The scenario consists of a wired IP network with two Ethernet switches, connected using a 100 [Mb/s] link, forming the backbone of the network. The network has 10 customers (or users), attached to switch 2, each having a stationary user endpoint, that connects to the network through an Ethernet cable. The users have access to three different services in the network; *Voice*, *Video* and a *Best Effort*. The user and service endpoints are all connected, using links with "No limit" on the capacity, hence no emulation of link data rates. This is done to avoid that significant bottlenecks will occur at the endpoints, instead of where the focus of this project lies, namely at the backbone link. All links in the network are configured to run full-duplex communication.

The network architecture of this use case scenario has a single route only, hence the traffic between the different users and services will traverse the same path in the network, thus increasing the chance of a congestion happening, when a service sends a burst of traffic into the backbone. All traffic going through the backbone is aggregated into one queue. This may be an over-simplification of what a real backbone network would look like, however the entire scenario has been scaled down, such that it is possible to emulate it. A backbone link of 100 [Mb/s] is also not realistic, but it reduces the required number of flows needed to be generated in order to utilise the link at different levels.

2.3.2 Service usage

The services used in the use case are all defined as client-server structures, with services located on the servers and the user endpoints connect as clients. While using a service, there will be a traffic stream between the user and the server for the particular service. It is only interesting to investigate the traffic going in the congested direction of the network, for that reason all traffic is generated with the service nodes as sources and the user nodes as sinks. This, in essence, is the same as only analysing the traffic going in the congested direction, but it reduces the overall load on the computer performing the emulation.

Assuming that all traffic in the network, has constant Inter-Departure Time (IDT) and

payload size, no congestions would happen, as long as the average utilisation stays below 100 [%]. It is therefore necessary to generate bursty traffic in order to increase the risk of congestions to happen. When a service bursts traffic into the backbone, a small congestion will happen in a switch, because it receives packets faster than it is capable of dispatching them.

Switches generally use queues to handle bursts, but as the number of packets in such a queue increases, so will the delay experienced by each next-coming packet increase, because the last packet in the queue has to wait for all other packets to be dispatched first.

In order to generate these bursts in the network, both video and best effort traffic will be bursty traffic. Video will be generated according to a Markovian ON-OFF process, with exponentially distributed ON-OFF times and constant payload. Best effort will be generated with exponentially distributed payload sizes and exponentially distributed IDTs. Voice traffic will be generated as a flow with constant bit rate. For more information on how traffic is modelled and generated, see appendix C.

2.3.3 Use case performance measurements

The use case scenario has been emulated on a test bed, as previously mentioned. The method for how the measurements are performed, including traffic generation, is documented in section 6.2 Test method.

The results from the emulation of the use case is shown in figure 2.3 as empirical Cumulative Distribution Functions (CDFs) for voice traffic at different utilisation levels on the backbone link.

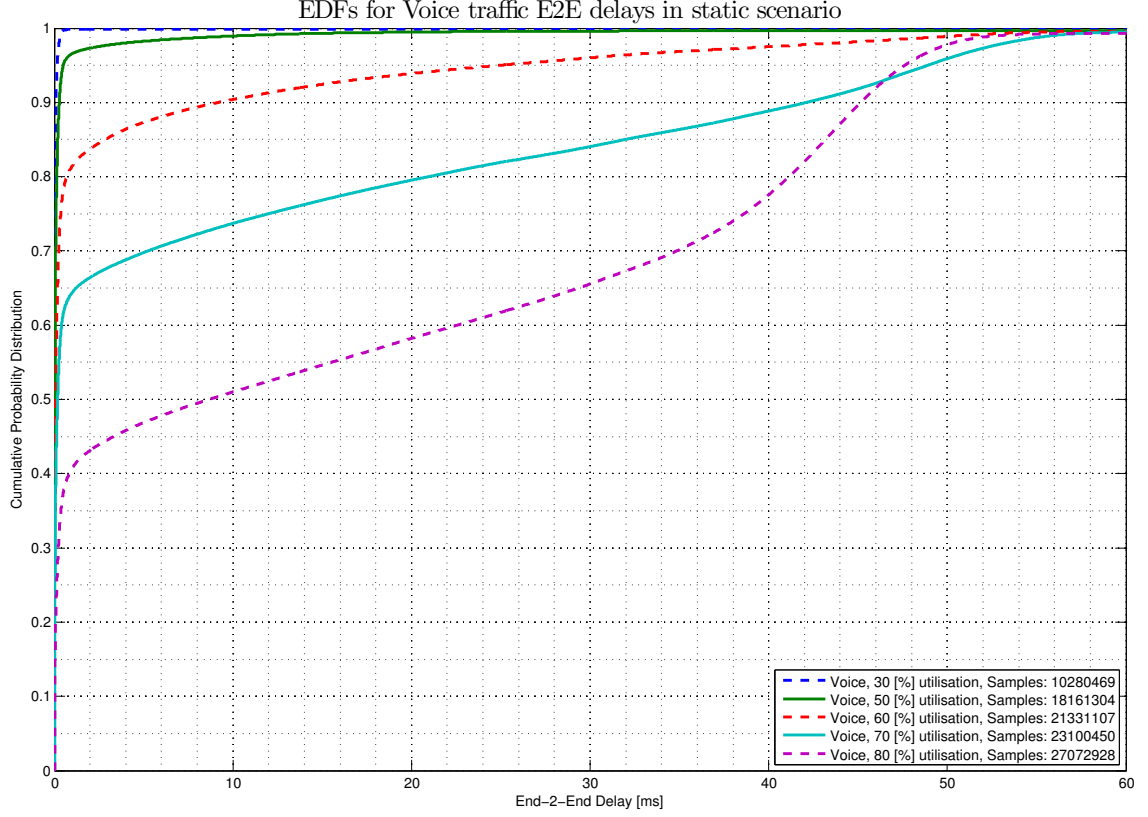


Figure 2.3: EDFs for voice traffic delay distributions for the use case scenario, aka. static scenario. More detailed statistics are listed in table A.1, page 66.

The figure shows that an increase in the average utilisation of the network leads to an increase in the delay experienced by each individual voice packet. This is also to be expected, because whenever video or best effort services send a burst of traffic into the network, it is going to affect all other traffic, until the burst of packets has been served. When the average utilisation is increased, so is the mean number of flows for each service, leading to a higher probability of multiple burst occurring within a relative short period of time. Hence, a situation which leads to a congestion.

This concludes the preliminary analysis, which stated the problem, defined the service types and the corresponding QoS requirements. To explore the problem characteristics, a use case was explored, resulting in an expected relationship between E2E delay and link utilisation. The next chapter continues the problem analysis, beginning with an investigation of Software-Defined Networking.

3 Analysis

This chapter describes the concept of Software-Defined Networking (SDN), first in relation to Cognitive Networks and after that in a more technical perspective. This includes an analysis of the OpenFlow protocol and its functionality for both monitoring and configuring the network. The last part of the chapter elaborate on what state information is needed from the network, and how the network can be controlled based on this.

3.1 Software-Defined Networking

The SDN concept was introduced during the introduction in chapter 1, this section seeks to give a more thorough introduction to the SDN concept.

3.1.1 SDN in relation to Cognitive Networks

The general SDN concept is not a new concept in terms of networking. A similar concept which has existed for several years before the invention of SDN is called cognitive networks, which was first introduced by Thomas et al. [2005]. The concept of cognitive networks is shown in figure 3.1.

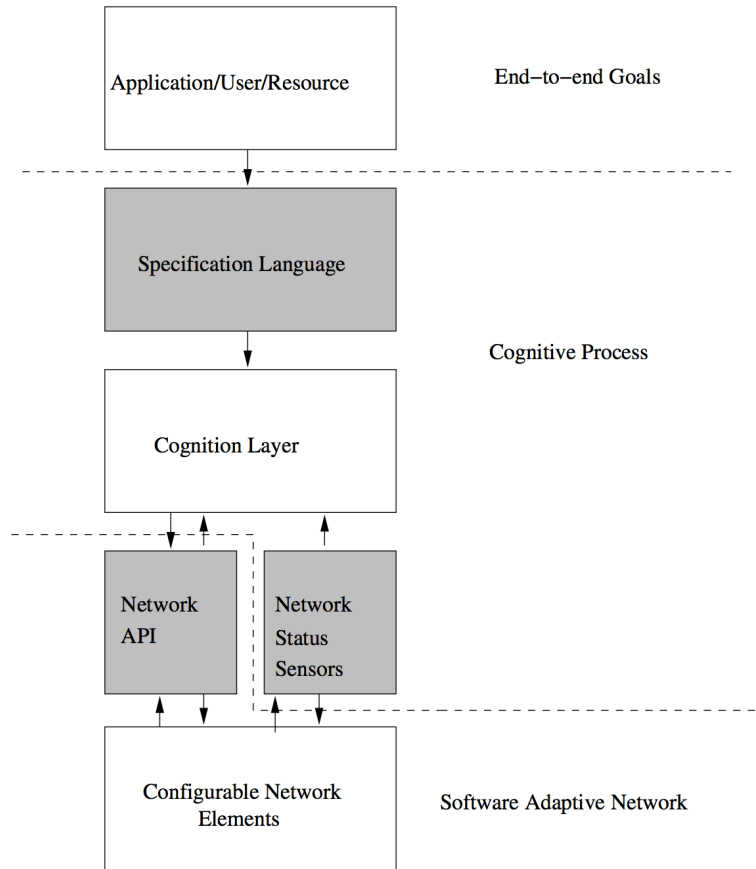


Figure 3.1: Cognitive networks model [Thomas et al., 2005].

The idea is to be able to guarantee E2E goals for traffic in a network. This is achieved by creating a layered abstraction model of the network, such that applications in the top layer become independent of the deployment and physical hardware used in the network. The applications present the system with a set of requirements, which the rest of the system, then must seek to fulfil. The bottom layer, which is called Software Adaptive Network, is a software-based network, which makes it possible to change configurations in individual switches in real-time. The Cognitive Process, which is the middle layer, handles the interpretation between the E2E goals and the actions that are applied to the network in order to fulfil the goals. In Thomas et al. [2005] it is proposed to implement the cognition using a feedback loop called Observe Orient Decide and Act (OODA).

The OODA concept is shown in figure 3.2.

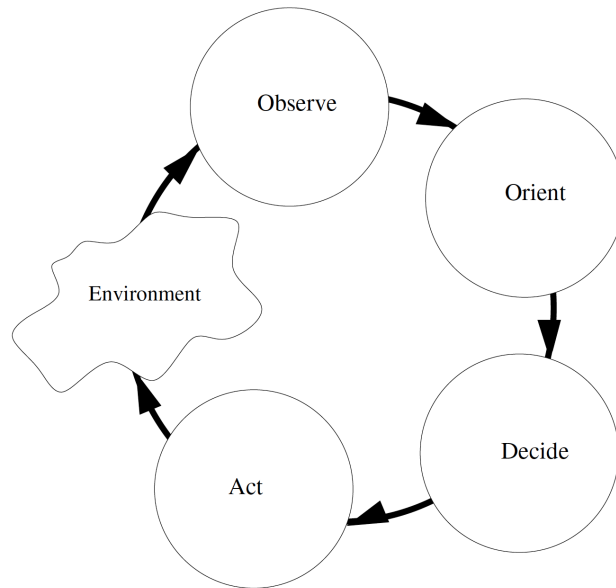


Figure 3.2: The OODA model as presented in [Thomas et al., 2005].

The OODA loop consists of four processes, where Observe is the first one. During Observe the state of the environment is obtained. Orient will then process the observed state of the system. When Orient has processed the state of the system, the Decide process must determine if and what actions to apply to the environment based on the outcome of the Orient process. Finally, the Act process must carry out the actions that the Decide process determined were necessary to apply to the environment.

When cognitive networks were presented in 2005 it was not possible to create an actual Software Adaptive Network, due to technical limitations, which limited the usefulness of cognitive networks at that time. SDN is however a realisation of the Software Adaptive Network, which makes it possible to apply the concepts from cognitive networks to an actual network, and not just a simulated network.

3.1.2 What is SDN?

Figure 3.3 shows the concept of SDN where the control plane is extracted from the individual switches and gathered in one logically centralised controller, which all switches in the network are connected to.

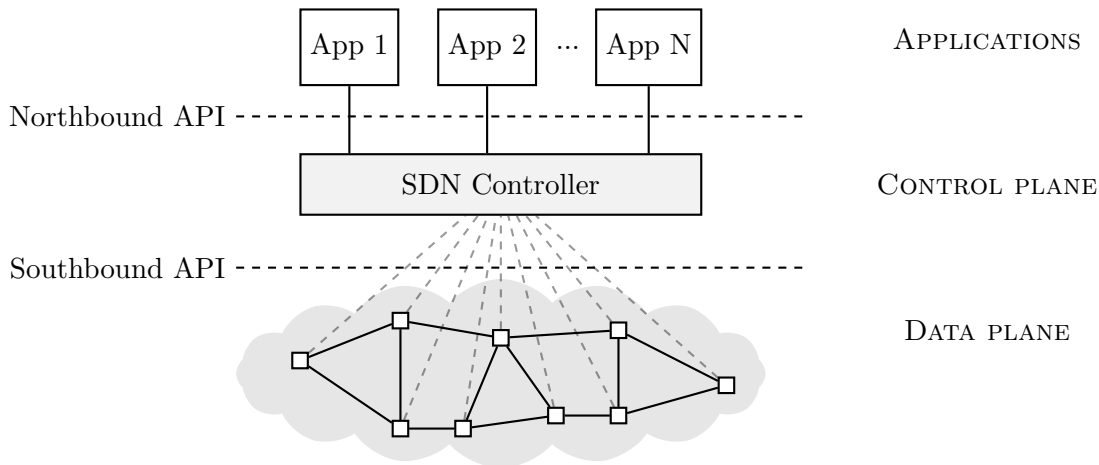


Figure 3.3: The layered abstraction of Software-Defined Networking (SDN).

The purpose of extracting the control plan is to create a layered abstraction of the network, such that developers can create network applications independent of the actual switches deployed in the network. Because all switches connect to a centralised controller, the applications will be able to extract information about the state of the entire system. This enables the applications to make global optimal decisions, based on the entire system, instead of making local optimal decisions based on the individual switches and their conception of the system.

The SDN controller has two APIs, one southbound to the switches and one northbound to the applications. The northbound API provides abstract information about the network to the applications and is used by the applications to instruct the network. The controller then translates the application commands into commands for the specific switches.

The southbound API is used for communication between the controller and the switches. This API can use different standard network management protocols such as SNMP or various other management protocols. The evolution of the SDN concept started when the work done by McKeown et al. [2008] was first presented. They presented OpenFlow, which is both a communication protocol as well as a switch specification. The OpenFlow protocol is further detailed in section 3.2, and is considered the de facto standard for southbound communication in SDN networks. The OpenFlow protocol can be seen as the catalyst for SDN, because it brings the possibility for other than the network equipment vendors, to develop custom control plane software for the network. It is a break with the vendor-specific configuration methods, which adds to the complexity of managing larger networks.

3.1.3 SDN controllers

There are currently a lot of different SDN controllers on the market, backed by different companies. One of the largest controller projects is the OpenDayLight project¹ which is a project managed by Cisco, HP, IBM, Microsoft and other large companies.

But there are also other controllers developed by individual companies such as the Floodlight² controller developed by BigSwitchNetworks, or the RYU³ controller developed by the Japanese Nippon Telegraph and Telephone Corporation.

The main difference between the different controllers is the language they are implemented in, the level of documentation provided, the number of features or standard applications they provide and in general the complexity of the code bases.

The southbound interface for all of these controllers are based on the OpenFlow protocol, but there are variations in terms of which protocol versions the different controllers support.

3.2 OpenFlow

The OpenFlow protocol and switch specification was first presented in by McKeown et al. [2008].

The concept of OpenFlow in relation to a regular switch is shown in figure 3.4.

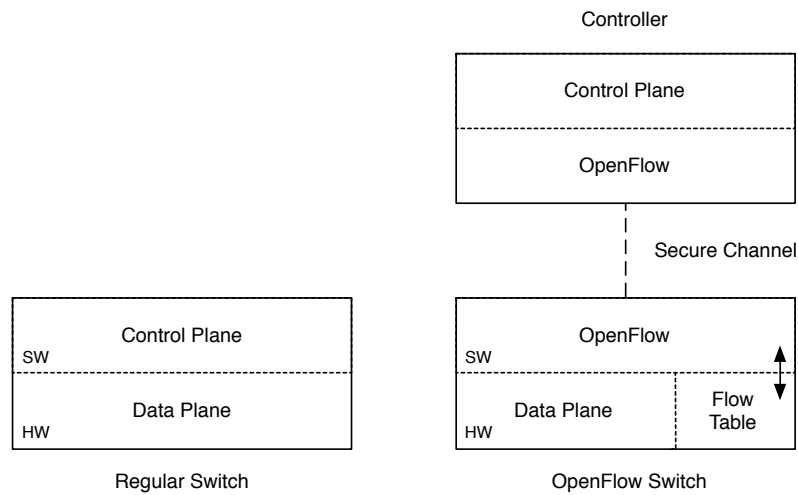


Figure 3.4: A regular switch compared to a OpenFlow-enabled switch.

¹<http://www.opendaylight.org>

²<http://www.projectfloodlight.org/floodlight/>

³<http://osrg.github.io/ryu/>

The meaning of OpenFlow is two-fold, first of all it is a communications protocol used by SDN controllers to retrieve information and modify switch configurations. Secondly, it is a switch specification, i.e. a definition of what a switch must be capable of, in order to support OpenFlow.

Figure 3.4 shows the general definition of a switch on the left-hand side, where both the data plane and control plane is located on the switch itself. The right-hand side of the figure shows, the architecture of an OpenFlow-enabled switch. OpenFlow replaces the regular control plane in a switch with a OpenFlow-control plane, such that the data plane configuration of a switch can be controlled from an external controller. This is primarily achieved by letting the OpenFlow control plane read and write flow entries to the flow tables within a switch.

The majority of OpenFlow-enabled switches and controllers on the market currently supports OpenFlow version 1.0 [Open Networking Foundation, 2009] (from Dec. 2009), which is why this explanation of OpenFlow will focus on that specific version. However, newer versions of the protocol exist, where the most recent specification is version 1.4 (from Oct. 2013), but even version 1.3 (from Jun. 2012) is still in experimental "support" in software implementations which are freely available today. This bears witness of the rapid evolvement the protocol is undertaking, and that SDN as concept is still in the early stage of its implementation.

The OpenFlow specification in [Open Networking Foundation, 2009] defines a set of required and optional features, that a switch must and can implement in order to support OpenFlow. The description of OpenFlow in the following, focuses on the required features, as these can be expected to be present in all OpenFlow capable equipment.

3.2.1 Flow Table

The flow table, which OpenFlow reads from and writes to in switches consists of entries that are formatted as shown in figure 3.5.

Header Fields	Counters	Actions
---------------	----------	---------

Figure 3.5: A flow table entry in a switch supporting OpenFlow version 1.0 [Open Networking Foundation, 2009].

A table entry consists of three parts; Header Fields, Counters and Actions. The Header Fields act as the key for the entry, i.e. when a packet is received from the network, the switch will try to match the header fields of the received packet to one of the flow table entries.

When there is a match between a packet and an entry, the associated Counters for that entry is updated and the associated Action is applied to the packet. All of these concepts will be explained further in the following sections.

When a flow entry is created and sent to a switch it is possible to specify different timeouts for the entry. One of these timeouts is the hard timeout, which removes a flow entry after

a specified period of time. The other timeout is the idle timeout, which removes a flow entry when no packets has matched the entry within the specified time.

3.2.2 Match

The ability to match packets to rules is one of the key features of OpenFlow. A packet can be matched based on any combination of the fields from the Ethernet header up to and including the transport layer protocol header. The input port a packet is received on can also be used for the match, e.g. such that all packets arriving on input port 1 can be forwarded to output port 2.

The fields, that are not needed for a match, can be set to wildcards such that everything will match them. This makes it possible to create both, very general matches, such as forward all Transmission Control Protocol (TCP) packets to output port X, but also very specific matches, such as the specific flow between two MAC addresses on a specific TCP port.

In the OpenFlow domain a flow is specified by its corresponding match in the flow table. This enables an SDN network to use a very generic and dynamic definition of a flow, which can change in real-time according to the requirements for the traffic in the network.

Figure 3.6 shows the logic that is executed in a switch, when a packet is received.

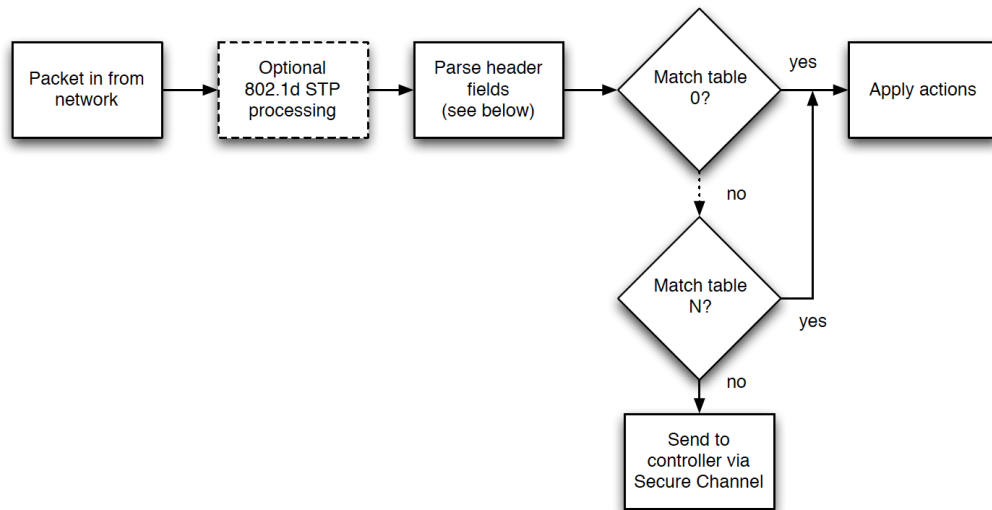


Figure 3.6: A flowchart showing the logic for processing incoming packets [Open Networking Foundation, 2009].

The specification defines that support for the spanning tree protocol (IEEE 802.1d) is optional, hence the dashed border in the figure. Afterwards the headers of the packet are parsed, based on the EtherType, IP version and protocol contained within the IP packet. The parsed header fields are then matched against the flow table entries, $1, \dots, N$, starting from the entry with the highest priority. If a match is found the corresponding action is applied, otherwise the packet is sent to the controller for processing.

3.2.3 Action

OpenFlow supports a wide range of actions that can be applied to a flow of packets. The possible actions range from outputting the packet to a specific port in the switch, enqueueing the packet in a specific queue for an output port, to re-writing certain fields in the packets header. An action can also be to drop all packets that matches a flow entry.

Table 3.1 gives an overview of the different actions that can be applied to a packet, once it matches a flow entry.

OpenFlow 1.0 Actions
Drop
Output to port
Set VLAN ID
Set VLAN ID priority
Strip 802.1q header
Set MAC source
Set MAC destination
Set IP source
Set IP destination
Set DSCP field
Set TCP/UDP source port
Set TCP/UDP destination port
Enqueue in queue at output port

Table 3.1: Overview of OpenFlow actions.

The OpenFlow specification defines that when adding/removing or changing an existing flow table entry, these instructions must be send as an OpenFlow FlowMod message. Such a message simply contains the match that is used as key for the entry, i.e. the definition the flow to interact with, and which action to apply when receiving a packet from such a flow.

3.2.4 Flow statistics

OpenFlow can extract traffic statistics on different abstraction levels, from individual switches, based on the counters in the flow tables. Generally, there are three different kinds of statistics; flow, port and queue statistics.

When a flow statistics request is send to a switch, it is replied with the values shown in table 3.2.

OpenFlow 1.0 Flow Statistics

Rx packet count
 Rx byte count
 Duration in sec.
 Duration in nano sec.

Table 3.2: Overview of a flow statistics reply.

Table 3.3 shows a list of the values that are obtained, when requesting the statistics for a specific port in a switch. The port statistics reply contains both Rx and Tx values for the specified port.

OpenFlow 1.0 Port Statistics

Rx packet count
 Tx packet count
 Rx byte count
 Tx byte count
 Rx packet drop count
 Tx packet drop count
 Rx errors
 Tx errors
 Rx overruns

Table 3.3: Overview of a port statistics reply.

It is also possible to request statistics for a queue, within the switch. Queue statics can be requested for a specific queue, identified by a port number and a queue id, or queue statistics can be requested for all queues within the switch.

Table 3.4 shows the content of a queue statistics reply.

OpenFlow 1.0 Queue Statistics

Tx packet count
 Tx byte count
 Tx errors

Table 3.4: Overview of a queue statistics reply.

When requesting statistics for more than a single flow/port/queue, the reply will be formatted as an array, such that the statistics for each individual flow/port/queue can be extracted uniquely.

3.3 System as a feedback loop

SDN can easily be seen as a system with a feedback loop because of the capabilities of OpenFlow. Using OpenFlow, the state of the system can be obtained, by extracting statistics for flows/ports/queues, and the system can be controlled by applying different OpenFlow actions to different flows.

Similar stats can also be obtained using SNMP, sFlow or some other management/monitoring protocol. However, introducing more protocols increases the complexity of the network set up, which will result in a higher management cost in the long run. Therefore, the work presented in this report, is based solely on the use of OpenFlow as the network interface protocol.

This is feasible, since OpenFlow as a standalone protocol, makes it possible to both extract information about the network and to apply specific low-level configuration in individual switches. Thus, enabling the feedback loop.

The beginning of the analysis explained the relationship between SDN and cognitive networks, where it was demonstrated that SDN is a realisation of the lowest layer in the cognitive network model, namely the Software Adaptive Network. This is further investigated in this section, where the network feedback will be discussed first, and then the actions that can be applied to the network will be described.

3.3.1 Network feedback

The network feedback must provide the SDN controller with information about the current state of the network. The obtained network state information must be sufficient in order to determine if the SLA for a flow is fulfilled. This means that the network feedback must provide information about E2E delays, throughput and packet loss rate.

End-2-End delay

E2E delay information can be obtained in several different ways. It can be obtained by performing active delay measurements in the network, using packet injections. The work done by Phemius and Bouet [2013] indicates that this method is quite possible, and that the accuracy of it is within 1 – 2 [%] of a ping measurement, while using 81 [%] less bandwidth compared to ICMP. However the work presented by van Adrichem et al. [2014] indicates that packets injected to the network through the OpenFlow control plane in the switches yields inaccurate results, due to software scheduling in the Open vSwitch⁴, which is the same switch model used in this work. van Adrichem et al. [2014] suggests that a dedicated measurement VLAN is created between the SDN controller and each of the switches, such that the controller can inject packets directly and receive them directly, without using the OpenFlow protocol and control plane to transport the packets.

Ping would be another solution, but it is only capable of measuring Round-Trip-Time (RTT), which makes it quite difficult to determine if the link delays are distributed asymmetrically. Ping will also leave a significant footprint in the network, making it a more intrusive approach.

⁴Open virtual Switch: <http://openvswitch.org/>

A third way to determine link delays is estimation based on queue lengths within the switches. This requires an interface to the switches such that the current queue lengths for any/all queues can be obtained. Furthermore, it requires a model of the scheduling algorithm within the switch, if there are multiple queues for the same output port. Depending on the complexity of the queue set-ups in the switches, this approach may result in a complex estimation model.

Throughput

The throughput for a flow can be estimated using OpenFlow statistics. The average throughput, over the entire life-time of a flow, can be calculated using the flow statistics counters *Duration in seconds* and *RxBytes* presented in section 3.2. The average throughput can be calculated as shown in equation 3.1.

$$T_{Flow, Life-timeaverage} = \frac{Rx [Byte]}{D [s]} \quad (3.1)$$

where, Rx is the total number of bytes received for a specific flow, and D is the time the flow entry has existed in a switch, i.e. how long the flow has existed in the network.

In terms of calculating the utilisation of the system it is necessary to be able to calculate the current throughput of a port. This can be done as shown in equation 3.2.

$$T(t)_{Port, Instantaneous} = \frac{Tx(t) [Byte] - Tx(t-1) [Byte]}{Time(t) [s] - Time(t-1) [s]} \quad (3.2)$$

where $Tx(t)$ is the latest total number of bytes transmitted over a port at time t and $Tx(t-1)$ is the previous sample. $Time(t)$ is the time at which the value of $Tx(t)$ was received from a switch, and likewise $Time(t-1)$ is the time at which $Tx(t-1)$ was received.

Packet loss rate

In order to be able to fulfil the packet loss rate SLA of a flow, it is necessary to be able to obtain this information for each individual flow in the network. None of the OpenFlow flow statistics describe the packet loss rate experienced by a flow. OpenFlow does have packet loss rate per port though, however there is no obvious way to map this information to each individual flow.

It is therefore necessary to find a way of estimating the packet loss rate of a flow. One way, is using state information of a flow, e.g. when statistics are polled for a flow, the statistics will be processed according to the actual flow state. The flow state is, in this context, defined from the action of the flow. If it is set to *Drop*, it is in drop state, otherwise it is determined as in active state.

A statistics reply for a flow contains absolute values i.e. number of bytes and packets the switch has received for a particular flow. These absolute values can be converted to incremental values by taking the latest value and subtracting the previous value, as shown

in equation 3.3. Whenever a new incremental drop value is calculated it is added to the previous values, as the updates are incremental. The incremental values combined with the state information of a flow, i.e. if the flow is in drop state, can be used to estimate the packet loss rate as shown in equation 3.4,

$$\hat{R}x(t)_{total,drop} = \hat{R}x(t-1)_{total,drop} + \overbrace{Rx(t)_{total} - Rx(t-1)_{total}}^{\text{Incremental packet drops}} \quad (3.3)$$

$$\hat{P}(t)_{total} = \frac{\hat{R}x(t)_{total,drop}}{Rx(t)_{total}} \quad (3.4)$$

where $\hat{P}(t)_{total}$ is the estimated packet loss rate, $\hat{R}x_{total,drop}$ is the estimated number of dropped packets and $Rx(t)_{total}$ is the total number of received packets. For equation 3.3 $\hat{R}x(t-1)_{total,drop}$ is the old estimate of dropped packets, $Rx(t)_{total}$ is the latest absolute number of received packets and $Rx(t-1)_{total}$ is the previous absolute number of received packets.

This is however not a flawless way of estimating the packet loss rate of a flow, because estimation errors occurs when flows change state from being in active state where packets are forwarded to dropped state where packets are actively being dropped, this is illustrated in figure 3.7

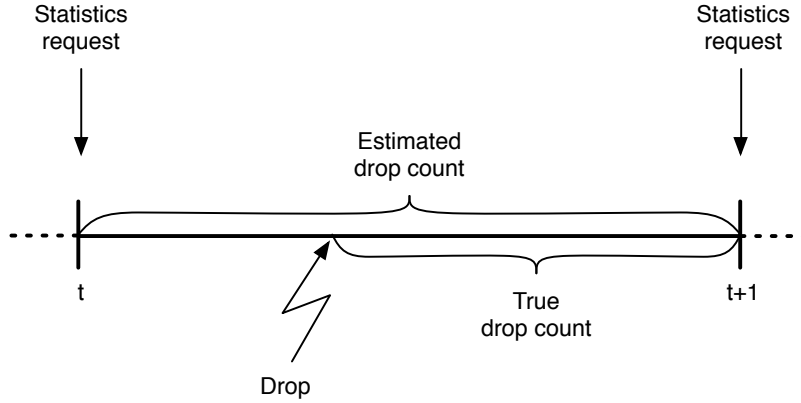


Figure 3.7: Exemplification of the estimation error introduced by the proposed estimation method.

If a flow changes state between two statistic polling, the estimated number of dropped packets will include the number of transmitted packets during the first part of the period, plus the actual number of dropped packets in the last part. The reverse estimation error occurs when a flow changes state from drop to active, because the actual dropped packets would then be counted as forwarded packets.

Another issue with this packet loss rate estimation method is that it is not capable of registering packet drops caused by buffer overflows, because OpenFlow does not register when this happens for a flow. This is only possible for a port or for a queue, as listed in table 3.3 and table 3.4. So if this information is required on a per-flow granularity, it will need to be extracted through a different interface to the switches, if the actual switch supports reading of this metric.

3.3.2 Network actions

In terms of actions to apply to the system, OpenFlow provides the most basic functionality as described in section 3.2.3, i.e. drop flows, send to output port and enqueue in queue at some output port. However, it is therefore necessary to investigate, what is actually needed in order to be able to guarantee SLAs.

In this work, the SLAs consists of three parameters, E2E delay, throughput and packet loss. The two latter of these parameters are dependent upon each other, because an increase in packet loss rate will result in a decrease in throughput. Like wise if the throughput of flow exceeds its nominated SLA throughput it can result in a need to drop from that flow, leading to an increase in packet loss rate.

Because of the dependency between throughput and packet loss rate, these two SLA parameters need to be treated together. This makes it possible to create a solution where one part focuses on the SLA delay parameter and another part that focuses on the SLA throughput and packet loss rate parameters.

End-To-End delay

The first part of the SLA that must be guaranteed is the network delay. The problem with the use case presented in section 2.3 is that all traffic through the backbone is aggregated in one queue, meaning that flows with strict delay requirements will have to wait whenever a crossing flow has bursted traffic into the backbone. This scenario is depicted in figure 3.8.



Figure 3.8: Exemplification of a voice packet which experiences a queue delay because of a video burst.

This is the same problem that technologies such as DiffServ attempts to solve, by using the DiffServ Code Point (DSCP) field to classify traffic into specific queues. Deploying an effective DiffServ configuration require thorough analysis of the traffic in the network, in order to determine how to prioritise queues, and decide how much bandwidth each queue should be guaranteed. This is a complex and time-consuming task, which might be aged out even before it is deployed, due to the constant change in available services in the network.

To avoid that, a simpler approach to queue configuration is desired. This can be achieved by creating a solution based on having multiple prioritised queues in the backbone, and

then utilise these queues based on the needs of each individual flow. That is, there is no need to prioritise a flow, if the delay through the default lowest priority queue is sufficient for that flow. Such a set up makes it possible to create a solution, where flows are dynamically moved between queues depending on their needs. This is what is depicted in figure 3.9.

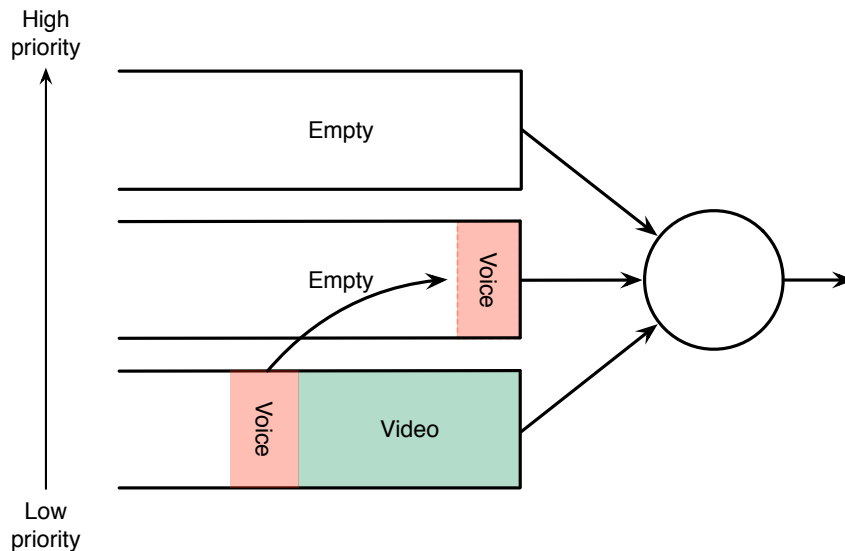


Figure 3.9: Exemplification of how a voice flow can be moved from one queue to another, to overcome queueing delay, due to a burst of video traffic.

The figure shows that the delay through the lowest prioritised queue is increased because of a burst from a video flow. This change in delay is observed and results in the voice flow being upgraded to a higher prioritised queue, with a lower delay, such that the voice packets can be dispatched faster.

Throughput and packet loss rate

As discussed before, there is a dependency between throughput and packet loss rate. If a flow is not receiving the throughput it should, it is most likely caused by a congestion in the network. This can be solved by actively dropping other flows, such that the traffic load in the network is reduced. However when actively dropping flows the packet loss rate will increase, which makes it a delicate balance when determining how much can be dropped from a flow.

Another thing to note is that if a flow is generating more traffic than its target throughput, and a congestion occurs in the network, it becomes necessary to drop flows to free up bandwidth. An obvious choice is to start dropping from flows that are exceeding their throughput SLA. In that case, the increase in packet loss rate is justified, because the flow is generating too much traffic.

This ends the analysis chapter. The following chapter describes the design of the implemented solution.

4 Design

This chapter covers the design for the implemented solution. First the software architecture of the applied SDN controller is presented, followed by a definition of a flow in the system. Thereafter, the design of the Flow-based Dynamic Queue Selector (FDQS) module is explained. It includes the State Maintainer and the Decision Logic components, which comprises the Delay- and Drop utility functions.

The solution to be designed, must be able to solve the problem stated in section 1.1, which is to be able to increase link utilisation, while adhering to the SLA constraints for different service types. These constraints are given by the QoS requirements, defined in section 2.2.4 Quality of Service requirements for Service Level Agreements.

As explained in section 3.3.2 the solution can split into two parts; one that handles throughput and packet loss and another which focuses on the E2E delay.

Before the design of the solution can begin, it is necessary to determine what the definition of a flow should be like in the system. It is also necessary to determine which controller the solution should be implemented on, as this will impose some limitations on the design of a solution.

4.1 The controller

SDN controllers were discussed in section 3.1.3. Based on the many options regarding controllers it has been decided to use the Floodlight¹ controller, because it provides the most basic features of an SDN controller such as providing topology information, which can help when routing decision needs to be made. Furthermore the Floodlight controller allows any modules to create and dispatch OpenFlow messages with relative ease, compared to some of the more complex controllers such as the OpenDayLight controller.

4.1.1 Controller modules

For the Floodlight project, new modules are created directly in the code base of the controller as a new Java package. In order for a module to start up it must be added to a configuration file that defines which modules are launched when the controller is started. When the controller starts, it will start the different modules, and set references between them according to the inter-module dependencies.

¹<http://www.projectfloodlight.org/floodlight/>

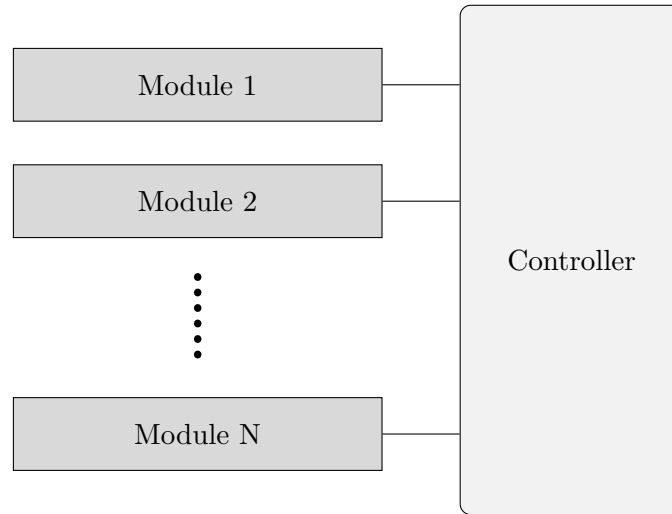


Figure 4.1: Floodlight controller with modules.

The architectural overview of the floodlight controller and its modules is shown in figure 4.1. The Floodlight controller is based on an event driven architecture, meaning that modules must subscribe to the messages they want to receive, i.e. in order for a module to receive OpenFlow Statistics Replies, the module must subscribe for such messages. When the controller receives an OpenFlow message from a switch it will dispatch that message to all modules that has subscribed for that specific message type. This is what is shown in figure 4.2.

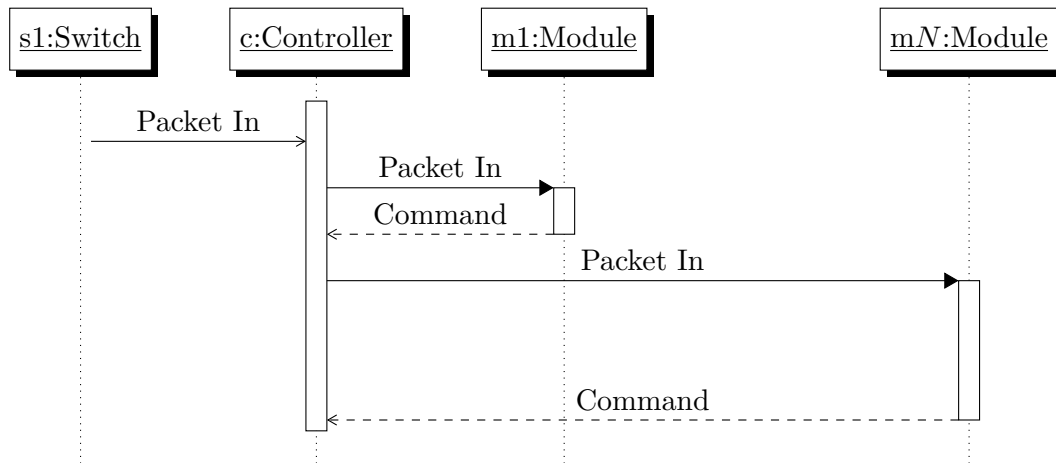


Figure 4.2: Sequence diagram showing how the controller dispatches messages to modules.

The figure shows that the controller receives a PacketIn message from a switch. This message is then dispatched to the first module that has subscribed for this message type. The message will not be dispatched to the next module before the first one called is done with the packet. A module must return a Command, which indicates if the message should be dispatched to the remaining modules or if the controller should stop dispatching it.

If the controller receives multiple messages from one or more switches, these messages are enqueued for dispatching because the controller only supports dispatching one message at a time. This means that the performance of the controller is dependent on the time it takes to process a message in each individual module, as the processing time of a single message is equal to the sum of all the processing time done in each individual module.

4.2 Definition of a flow

OpenFlow makes it possible to have a dynamic understanding of what a flow is, using all of the different header fields from the Ethernet frame to the transport layer header. The ability to use any of these fields including wildcards, makes OpenFlows match action and table entries very scalable for large scale networks. This makes it possible to set general flow table entries that direct the majority of the traffic in the network, whilst making it possible to set more specific flow table entries, that dictates what to do, with specific flows that needs specific actions.

As presented in section 2.2, this work focuses on three traffic types, Voice, Video and Best effort. It is therefore necessary to use a flow definition such that these traffic types can be differentiated, such that special actions can be applied for each of them. One way of doing this, would be to use the DSCP field in the IP header, which is the field that solutions such as DiffServ are based on. However the traffic generator used, in this work, does not support setting the DSCP field for its generated traffic.

Another solution would be for the application that is going to use the network for a specific service, e.g. a YouTube application, to negotiate with the network controller before starting its flow. The application and the controller would then be able to agree upon a SLA such that both parties are satisfied. This is a more futuristic approach, but indeed a possible option for developing more agile and maybe on-demand SLAs. However, that will not be pursued further in this work.

A simpler, but also more un-realistic approach is used instead, which relies on knowing MAC addresses the different services are located on, such that the controller modules knows that traffic going to or coming from the Voice node in the network is actually Voice traffic. This clearly imposes limitations on the presented results, but the focus of the project is not to be able to classify traffic.

A flow is therefore defined using the following tuple: {Ethernet Type, Media Access Control (MAC) Source, MAC destination, Transport Layer type, Transport Layer Destination Port}, where either the source or destination MAC can be used to determine which service type the flow is.

4.3 Flow-based Dynamic Queue Selector module design

The analysis in section 3.3 showed that SDN using OpenFlow will be capable of providing the information necessary in order to be able to apply the concepts of Cognitive Networks to this work. Cognitive Networks were also discussed in the analysis, where the OODA loop was presented as part of the explanation of Cognitive Networks. The OODA loop is what makes Cognitive Networks “cognitive”, i.e. be intelligent and have memory. The design of the controller module thereby becomes a matter of designing a module that contains the different parts of the OODA loop.

Figure 4.3 presents a way of dividing the OODA loop into smaller components.

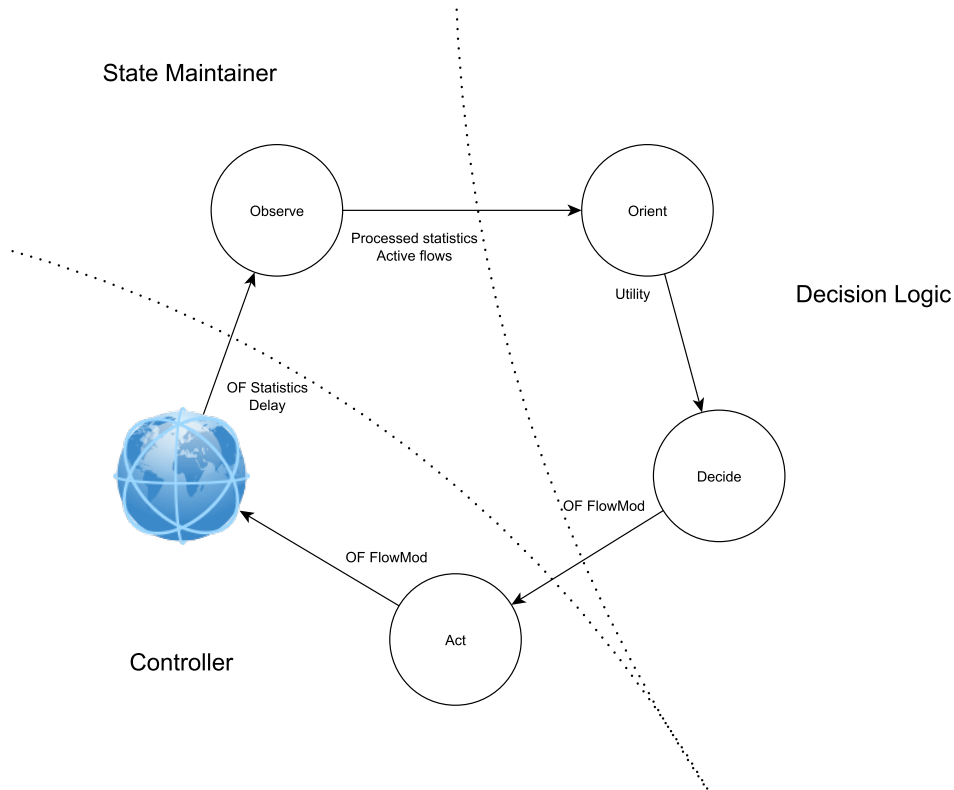


Figure 4.3: OODA loop applied to SDN and partitioned into components.

The figure shows that the interface between the network and the module will be handled by the controller, i.e. this is abstracted away from the module, which is the general concept of SDN. The Observe part of the loop is contained in a component called State Maintainer, which, as the name suggests, will maintain the state of the network. The State Maintainer must be able to provide processed statistics as well as information regarding all active flows in the network.

The Orient and Decide part of the loop are gathered in a component called Decision Logic. Decision Logic must evaluate the current state of the system, i.e. how is each individual flow performing relative to its SLA. This evaluation result in a utility value for each flow. Based on the utility values for each individual flow, a decision must be made regarding which actions to apply to the system. The Decide part of the loop results in a set of OpenFlow FlowMods that must be applied to the network, these are passed on to the controller, which then carries out the Act part of the loop.

This OODA partitioning leads to the module design shown in figure 4.4.

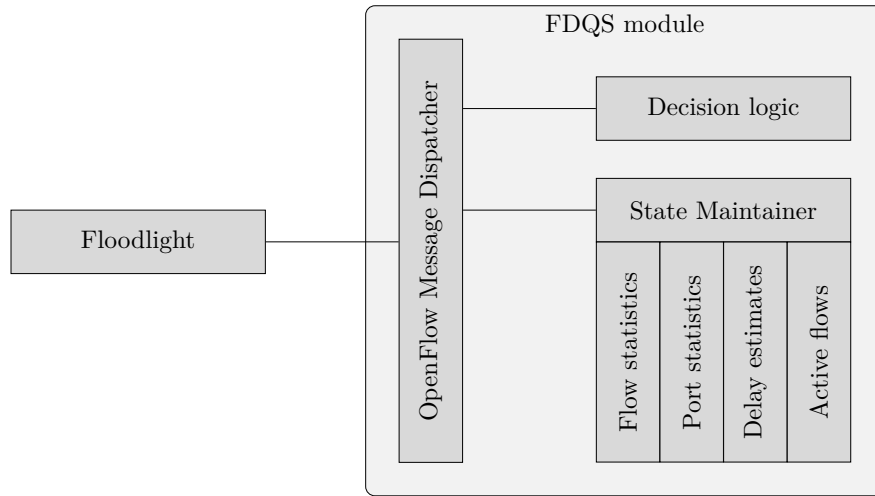


Figure 4.4: Module design for FDQS module to the Floodlight controller.

The module consists of the two components State Maintainer and Decision Logic, which both originates from the OODA partitioning. The State Maintainer has been further expanded with sub-modules for the different parts of the system it must maintain state for according to what was discovered in the analysis in section 3.3. The module design also contains a component called OpenFlow Message Dispatcher, which is a controller defined requirement to the module, such that the controller can dispatch messages to the module as explained in section 4.1.1.

The following sections will present a detailed design description of the internal components of the FDQS module.

4.4 State Maintainer design

The purpose of the State Maintainer is to collect information, process it and store it for later use. The Floodlight controller is event driven, which means that when statistics are requested from the individual switches it is not a blocking call in the module. But a response is received in the form of an event, at some later point in time.

4.4.1 OpenFlow statistics

In order not to overload the switches, statistics cannot be polled continuously, but should instead be polled with a certain time period. Statistics are polled per switch and there are two different statistics that are of interest, namely port and flow statistics. These are polled by generating two OpenFlow statistic request messages, one which requests all flow statistics on a switch, and one which requests all port statistics on a switch. This is done for each switch. The statistic replies come as events as described in section 4.1.1. It is necessary to create a queue where statistic replies can be offloaded too, such that the receive method of our module is not blocked more than necessary. This results in the need for a thread which can process the stored statistics replies. The design of this processing thread is shown in figure 4.5.

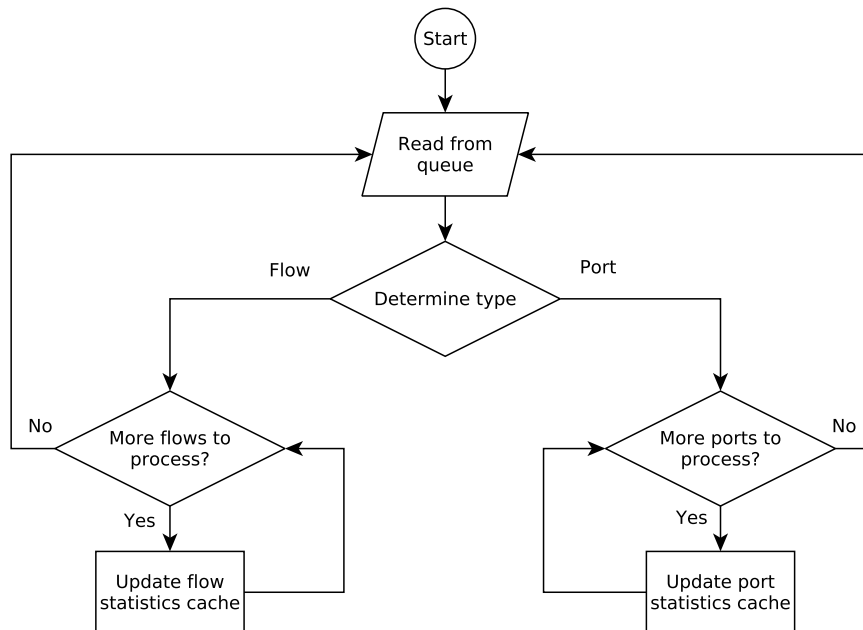


Figure 4.5: Flowchart showing the processing of statistic replies.

The statistics processor reads from a queue, this call is blocking, meaning that if there are no elements in the queue, the thread will sleep until an OpenFlow statistics message is put in the queue. When a message is read from the queue it is first classified, such that flow statistics can be processed differently than port statistics. When a flow statistics

message is processed, the flow stats cache is updated. Updating the flow stats cache means that the raw OpenFlow statistics are converted into SLA related values, according to the calculations presented in section 3.3.1. The processed statistics are then stored, according to which switch the statistics reply is from and which flow they belong to using the following key {Switch ID, Flow match}.

The procedure for processing port statistics is quite similar, except for that fact that only the throughput calculation from section 3.3.1 is applied, as this is the only relevant statistic for ports. The processed port statistics are also stored, but using a different key; {Switch ID, Port number}, which uniquely identifies each switch port combination.

4.4.2 Delay estimation

Section 3.3.1 in the analysis discusses different approaches to measuring or estimating delays in a SDN. Sources indicate that the packet injection method yields inaccurate results when injecting directly through the OpenFlow control plane. The proposed solution would however require larger modifications to the controller software, which makes delay estimations based on queue lengths more appropriate for this solution.

The State Maintainer must extract the queue length information from each switch periodically, such that the switches are not overloaded, similarly to what happens when statistics are requested. The key used for storing queue lengths are as follows; {Switch ID, Port number, Queue ID}, such that all queues are stored uniquely for each switch and port combination. The value that is stored is the delay estimate, based on equation 4.1

$$\hat{d} = q \cdot t_{\text{MTU}} \quad (4.1)$$

The delay estimate \hat{d} is the current queue length multiplied with the time t_{MTU} it takes to transfer a Maximum Transmission Unit (MTU) sized packet over a 100 [Mb/s] link. The MTU size is set to 1500 [Byte] , which results in a transmission time of 120 [\mu s] . The default queue lengths for Network Interface Cards (NICs) in Linux is 1000 packets, which results in a maximum queue delay of 120 [ms] . Using a transmission time based on MTU sized packets, makes the delay estimation a worst case estimate within a queue, however the estimation does not take queue prioritisation into consideration, meaning that the actual delay may be even higher, if there are packets in a higher prioritised queue, because these packet must be served before a lower priority queue.

4.4.3 Active flows

The state maintainer must also be aware of which flows are active in the system. New flows are discovered when a switch receives a packet that does not match any of its table entries, causing the packet to be send to the controller as an OpenFlow Packet In message. By letting the FDQS module register for these message types, new flows can be registered. The FDQS module must also decide how to route the flow in the network, since default routing module in the controller has been disabled, as it would otherwise conflict with the functionality of this module.

Setting up a new flow

When a packet is received from a switch, the FDQS module must decide how to route the packet and the following packets in the flow, the packet represents. There are two approaches to setting up a path, the first way is to decide what to do with the flow on the switch that has send the packet to the controller. This is then repeated for each switch the flow traverses in the network, when a switch receives the first packet in a flow. This is illustrated in figure 4.6, where the sequence of events is numbered, such that the flow of packets transmitted can be followed.

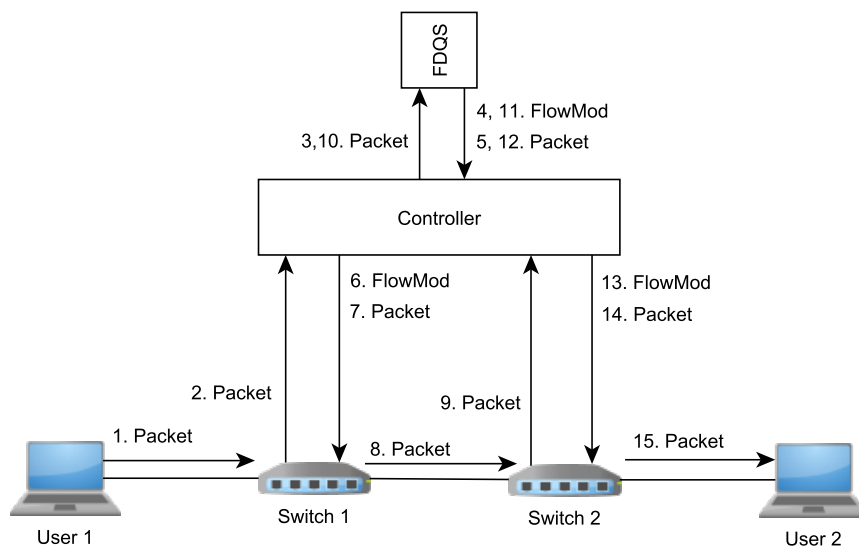


Figure 4.6: Sequence of events when setting up a path for a new flow.

The problem with the approach illustrated in figure 4.6 is that it takes time for the controller and its modules to process a packet, which means that the more times a packet has to be send to the controller, the larger its delay will become. Furthermore there is the possibility that a switch will receive more packets from a flow, before the first packet of the flow has been processed, creating a queue of packets to be send to the controller from a switch. This problem will become worse for each switch the flow traverses.

Setting up a new path can be done with considerable less overhead in terms of transmitted data as well as processing time using topology information stored in the controllers default topology module. The topology module is capable of reporting all links between OpenFlow switches in the network. In the link discovery module a link is defined as; {source switch, source port, destination switch, destination port}. Using this information it is possible to implement a shortest-path algorithm to find a path through network. If all links in the network are weighted equally, this results in the shortest-path algorithm finding the shortest hop count path through the network. This concept is illustrated in figure 4.7.

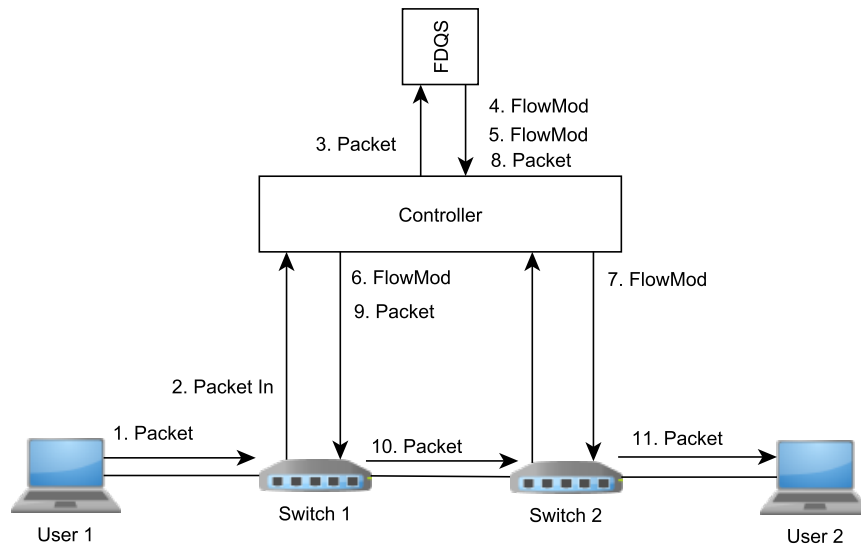


Figure 4.7: Sequence of events when setting up a path for a new flow, using topology information to apply rules to all switches in the path, the first time a packet is seen from a flow.

The main difference between figure 4.7 and 4.6 is that the first packet of a new flow is only sent to the controller once, from the first switch that encounters the flow. The FDQS module generates FlowMods for all switch and applies them to the switches, before allowing the flow to resume by returning the Packet to the switch it was received from.

Removing flows

The analysis of the OpenFlow protocol showed that flow table entries have timers attached to them, these timers can remove a flow entry based on both idle timeout as well as a hard timeout. When a flow table entry is removed, an OpenFlow FlowRemoved message is sent from a switch to the controller. This message can be used to remove flows from the State Maintainer, when they are no longer active.

4.5 Decision Logic design

The Decision Logic component must make decisions for each individual flow based on the current state of the flow. As previously discussed, the SLA can be considered in two parts; delay and throughput/packet loss rate. The functionality of Decision Logic thereby becomes two-fold. First of all it must evaluate the state of each flow and determine if the flow should be up- or downgraded according to its delay SLA and it must evaluate how desirable a flow is to drop if there is a congestion in the network. Then it must determine if there actually is a congestion in the network and start dropping flows to eliminate the congestion. The design of this logic is presented in the flowchart in figure 4.8.

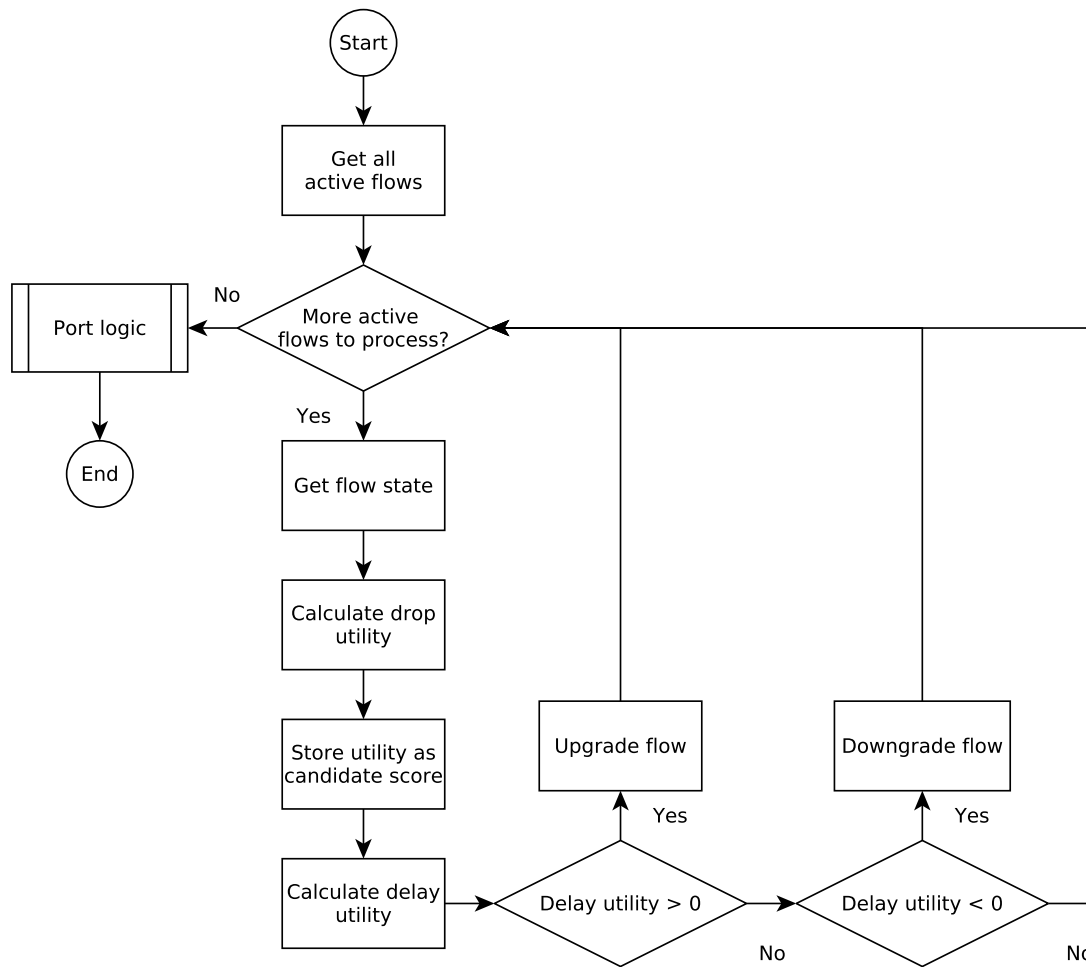


Figure 4.8: Flowchart showing the functionality of Decision Logic. The *Port logic* subroutine has its own flowchart, depicted in figure 4.9

Figure 4.8 shows a flowchart over the decision making process in Decision Logic. First all active flows are requested from the State Maintainer component. Then, for each active flow the following is done: Get the current state of a flow, which means that the processed statistics for the current flow is obtained from the State Maintainer. The state of the flow is then used as input to the drop utility function, which is explained later in section 4.5.1. The calculated drop utility value is stored in a sorted list, such that it can be used later when determining if and which flows should be dropped.

Then the delay utility is evaluated based on the current delay estimate for the flow, which is also obtained from the State Maintainer. If the delay utilisation is above 0, it means that the flow's delay SLA is being violated, the flow is therefore upgraded, by sending a OpenFlow FlowMod message to all switches that the flow passes through, instructing them to enqueue packets for the specific flow in a higher prioritised queue. If the delay utility value is below 0, then the flow is experiencing such low delays that it can be downgraded

to a lower prioritised queue, if one is available.

This design does not take into consideration, what happens if a flow is violating its delay SLA, but there are no higher priority queues to upgrade the flow to. This would most likely occur if the utilisation in the network is too high, or because too many flows has been upgraded to the highest priority queue.

When it is determined if a flow should be downgraded, the expected performance in the lower priority queue is not taken into consideration, because of this there may be situations where a flow is moved up and down between two queues, because one is too good and the other is too bad. This could be avoided by checking the performance of the queue being downgraded to, to see if it would meet the requirements of the flow being downgraded.

When all flows have been processed the next part of the Decision Logic functionality can be commenced. This part is shown in figure 4.9.

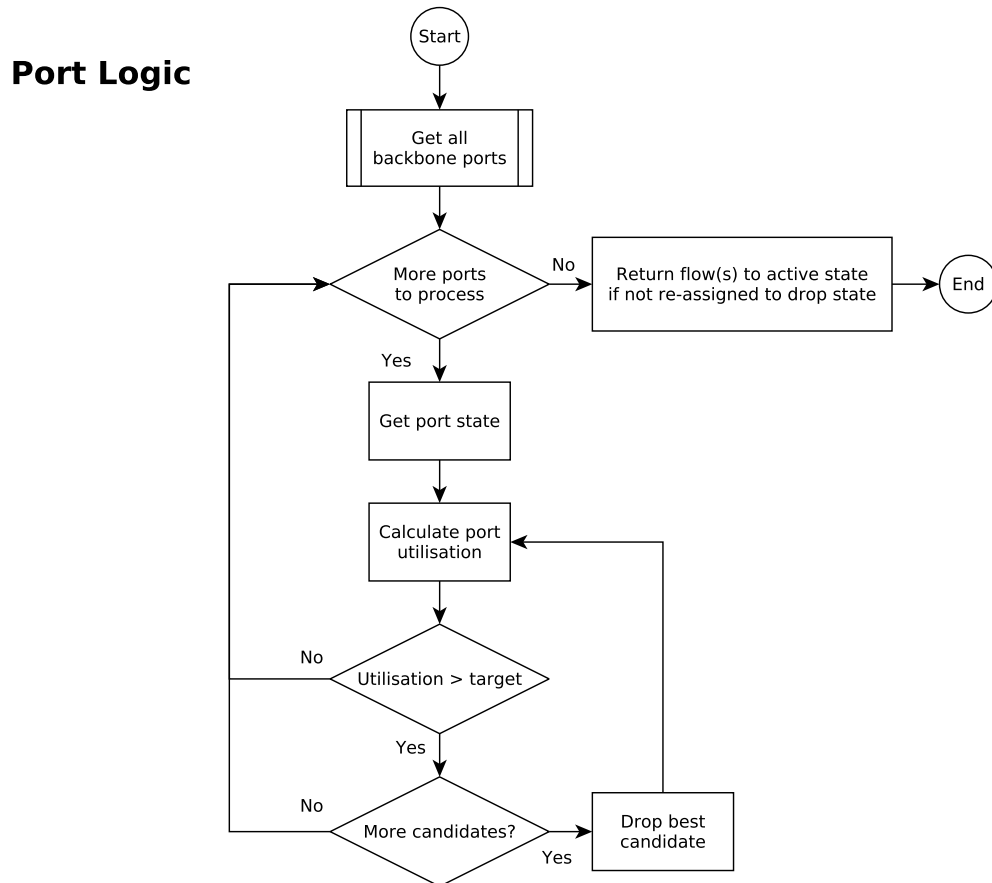


Figure 4.9: Flowchart showing the continued functionality of Decision Logic, relating to switch ports.

In this part of Decision Logic the state of each output port on each switch in the network will be processed. The ports in the backbone are found using the default topology module in the controller. For each port the following is performed: First, the state of a port is

obtained from the State Maintainer. The state of a port is the current throughput of an output port, this information is then used to calculate the utilisation of the port. If the port utilisation is above the target utilisation, the best flow candidate is found. This means that the flow with the highest drop utility value is found. This flow is then put into drop state, meaning that a OpenFlow FlowMod is transmitted to all relevant switches instructing them to drop all packets for this flow. Once a flow has been put into drop state, its traffic contribution across the entire network is subtracted, such that when the next port is evaluated, the dropped flow's contribution has already been subtracted. Then the utilisation of the port is evaluated again. This continues for as long as the port utilisation is above the target and there are flow candidates to drop.

Finally, all flows which were put in drop state during the last iteration are returned to active state if they were not instructed to be dropped during this iteration.

When all ports have been processed the Decision Logic component is done and can be stopped until next time the state of the system has to be evaluated.

4.5.1 Drop utility function

The drop utility function is used to determine how “attractive” a flow is, in terms of dropping it to free up bandwidth in the network. The drop utility is designed such that a higher drop utility value indicates that a flow is more desirable to drop than a flow with a lower utility value. Equation 4.2 shows how the utility is calculated,

$$U_{drop} = NR \cdot (0.2 \cdot (1 - PLR) + 0.8 \cdot (TR - 1)) \quad (4.2)$$

$$NR = \frac{Th}{Lc} \quad (4.3)$$

$$PLR = \frac{P}{SLA_P} \quad (4.4)$$

$$TR = \frac{Th}{SLA_{Th}} \quad (4.5)$$

where NR is the flow's current throughput Th normalised with respect to the link capacity Lc , which is shown in equation 4.3. PLR is the ratio between packet loss rate P of a flow and maximum allowed packet loss rate SLA_P defined for the flow's SLA, equation 4.4 shows how PLR is calculated. Equation 4.5 shows TR which is the throughput rate, which is a flow's current throughput Th relative to the expected throughput SLA_{Th} of the flow based on its SLA. The utility is calculated as the NR value weighted by the flows performance relative to its SLAs.

The weighting is based on prioritising throughput with a factor of 0.8 and packet loss rate with a factor of 0.2. The $(1 - PLR)$ part of the weighting ensures that as long as a flow stays below its SLA for packet loss rate, i.e. PLR below 1, it is more desirable to drop than a flow that is above its SLA, because the expression would then become negative when PLR is above 1.

The second part of the weight concerns the throughput of a flow relative to its SLA throughput. TR will go from 0 when a flow gets none of the throughput it is supposed too, to 1 when it gets the target throughput and above one when it has a throughput higher than its target. The $(TR - 1)$ thereby ensures that a flow, which has a throughput higher than its target is more desirable to drop than a flow which does not meet its SLA throughput.

The NR part of the equation biases the utility calculation, such that it is in general more desirable to drop flows with a high throughput compared to a flow with a lower throughput. This design choice has two sides to it, first of all it ensures that small flows in the network will not be the obvious choices when selecting a flow to drop. Secondly, it may also cause the algorithm that decides when to drop to be too aggressive, because large flows will be dropped first, which may not always be necessary if the utilisation target is only exceeded with a small fraction.

4.5.2 Delay utility function

The delay utility function is to be used when determining if a flow should be upgraded/downgraded to a higher/lower priority queue. The delay utility function is shown in equation 4.6,

$$U_{delay} = \begin{cases} 1 & \text{for } \hat{d} > SLA_{delay}, \\ -1 & \text{for } \hat{d} < \frac{1}{2}SLA_{delay}, \\ 0 & \text{otherwise} \end{cases} \quad (4.6)$$

where \hat{d} is the current delay estimate of the flow and SLA_{delay} is the maximum tolerated E2E delay for the service type the flow belongs to. The utility function is a step function ranging from -1 when a flow should be downgraded, to 0 when its current situation is sufficient, to 1 when the flow should be upgraded.

This concludes the documentation for the design of the Flow-based Dynamic Queue Selector (FDQS) module. The next chapter describes some of the aspects of the developed implementation.

5 Implementation

This chapter aims to explain some of the aspects relating to the implementation of the FDQS module as well as the test bed.

5.1 Virtual switches

This work is based on emulating a network on a single computer, which creates the need for virtual switches in order to setup the topology shown in figure 2.2 in section 2.3.

Open vSwitch (OVS)¹ is a production-level open-source implementation of a virtual switch, which supports the OpenFlow protocol version 1.0. Another open-source virtual switch that supports OpenFlow version 1.0 to 1.3 is the CPqD² switch. However CPqD is a user-space implementation of a virtual switch, which significantly increases the load on the CPU, compared to the OVS which is a kernel-space implementation. As the network scenario is emulated on a single machine, sharing of the computational resources among all nodes and links, is expected to have at least some effect on the end results. Therefore the OVS has been selected as the switch that will be used during the emulations.

5.2 Dynamic queues

In section 3.3.2 the idea of using multiple queues for guaranteeing delays was presented. In the implementation three queues are used, as this was deemed sufficient in order to be able to up- and downprioritise traffic in the network. The queues use the Hierarchical Token Bucket scheduling algorithm which is the default queue scheduler in the Open vSwitches. Because the Hierarchical aspect of the queues are not utilised, the scheduling algorithm can simply be viewed as Token Bucket scheduler. Each queue is allowed to use the full link capacity, which ensures that resources are not wasted, as they could otherwise have been if each queue was assigned a static amount of bandwidth. Each queue is also assigned a priority, which ensures that the highest prioritised queue is served first.

¹<http://openvswitch.org/>

²<http://cpqd.github.io/ofsoftswitch13/>

5.3 Delay estimation

During the implementation of a delay measuring algorithm, the switches were observed to add a significant amount of processing delay, upon reception of packets from the controller. This is also observed by van Adrichem et al. [2014], who also experienced this with the Open vSwitch.

When the switch receives a so-called PacketOut from the controller, it needs processing time to determine what to do next with the packet. This processing time was observed to vary in the range of 1 to 100 [ms], which is a relatively large delay contribution compared to the expected E2E delay in the system. Therefore, using PacketOut messages as probing packets from controller to the switches was found not to be a viable solution for estimating the E2E delays in the system.

In section 3.3.1 it was also proposed to use queue lengths in the switches to estimate the delays. However the Open vSwitches used in this work does not have an interface to extract such information. A workaround has therefore been found using the tc³ interface in the Linux kernel. This is possible because the virtual switches used in this project relies on Linux traffic control in order to be able to emulate different link speeds. The tc interface allows to read the current queue lengths for all queues in the switches.

³Linux kernel traffic control interface <http://lartc.org/manpages/tc.txt>

6 Evaluation

This chapter presents the test bed on which all tests have been conducted. The test bed description is followed by a description of what a test is and how it is conducted. Finally this chapter presents the results obtained using the solution that has been designed through this report. The obtained results are compared to the results obtained from studying the use case presented in section 2.3.

6.1 Test bed

The test bed consists of a Dell OptiPlex 7010 PC with the following hardware specifications:

CPU Intel(R) Quad Core(TM) i7-3770 CPU @ 3.40GHz

RAM 8 GB DDR3 RAM

HDD 250 GB, 7200 RPM Serial-ATA

and the following software installed:

OS Ubuntu 13.04 Server

Network emulator Mininet, version 2.1.0+

Virtual switch OVS version 2.1.0

SDN Controllers Floodlight version 0.90, OpenDayLight version 0.1.2-370

The network scenario is emulated using the Mininet framework [Lantz et al., 2010]. It uses OS features to instantiate lightweight virtualisation of network hosts, and interconnects them with virtual switches, according to a specified topology configuration. Effectively, what Mininet does, is to abstract this rather complex configuration task of the Operating System (OS), into a matter of high-level programming, which makes it ideal for rapid network prototyping.

6.2 Test method

This section describes how a test is performed. The process is depicted in figure 6.1, which is explained in more detail in the remainder of this section.

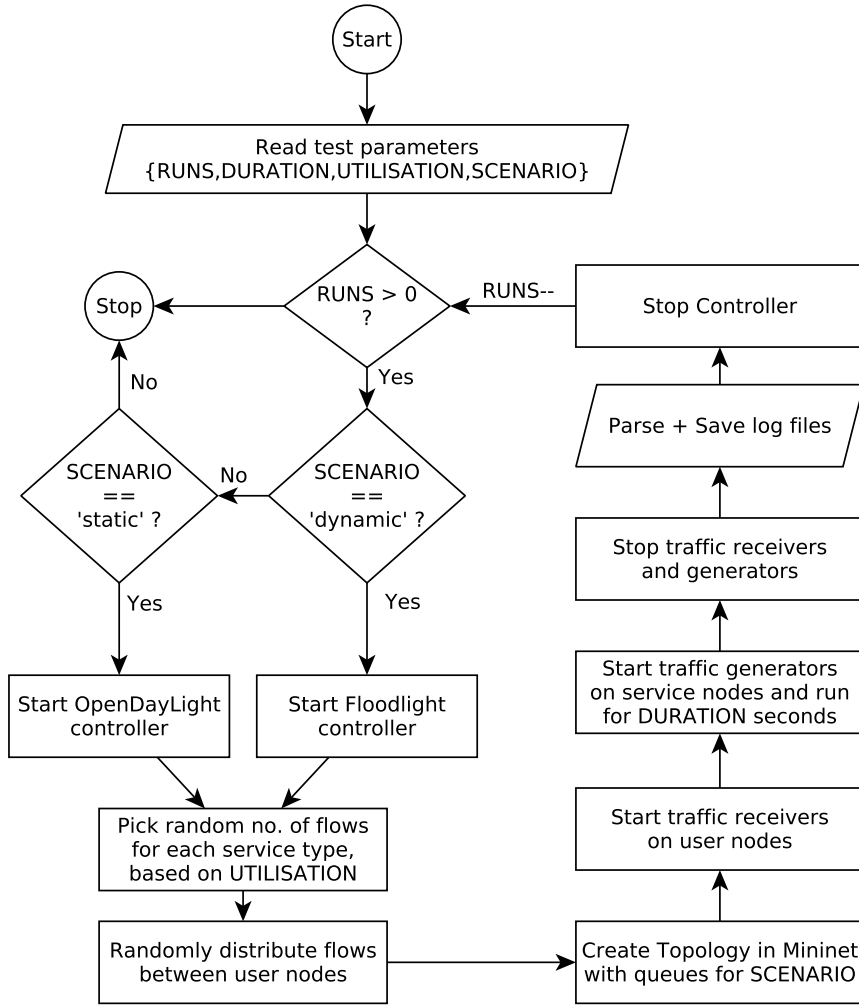


Figure 6.1: Flowchart describing the method for performing a test.

6.2.1 Test parameters

Each test is described via the parameters; **Runs**, **Duration**, **Utilisation**, **Scenario**. These are set before a test is performed, hence they do not change during a test.

Runs is the number of realisations within a test. The data for a single test thereby includes flow data from all the runs within that test. This decreases the likelihood of the data being overly biased by a single timing event in the OS scheduling. The number of runs is set to **30** [*runs*] for all performed tests in the presented work, this

was considered to be enough runs to allow the average utilisation converge towards the target utilisation.

Duration is the size of the time windows for the traffic generation within a run. This should be long enough to encompass the variability of the modelled traffic. The duration time is set to **60** [s] for all the performed tests.

Scenario can be either **static** or **dynamic**. *Static* refers to the reference scenario, which was first presented in section 2.3 Use case, where the network has no dynamic control algorithm applied to it and only one output queue per port in the backbone. Practically, this selects and starts the OpenDayLight controller, which sets up static flow entries in the switches, with Port-to-MAC mappings, just like a non-OpenFlow switch would set up its MAC table. *Dynamic* starts the Floodlight controller with the Flow-based Dynamic Queue Selector (FDQS) module, and sets up 3 output queues for each of the backbone ports.

Utilisation specifies the target utilisation of the backbone link, i.e. the aggregated amount of throughput in the network. The number is specified in percent, and is varied in intervals of 10 [%].

6.2.2 Generating traffic

The Utilisation parameter for a test, specifies the target value for utilisation, i.e. aggregated throughput, on the backbone link. This value is an average for the generated backbone utilisation, taken over all runs (realisations) in a test.

The traffic for each of the three service types is generated according to the traffic models, described in section 2.3.2 and appendix C. Based on the average throughput of each traffic type, which is shown in table 6.2 this results in an expected value $\mathbf{h}(\mathbf{s})$ for the throughput of a flow, for each of the service types s . That is

$$\mathbf{h}(\mathbf{s}) = \begin{bmatrix} h(s_1) \\ h(s_2) \\ h(s_3) \end{bmatrix} = \begin{bmatrix} 7 \\ 0.3 \\ 2 \end{bmatrix} [Mb/s] \quad (6.1)$$

for $s_1 = \text{video}$, $s_2 = \text{voice}$, $s_3 = \text{besteffort}$.

For each run, i in a test, a random number of flows $x_i(s)$ are generated for each of the service types s , i.e.

$$\mathbf{x}_i(\mathbf{s}) = \begin{bmatrix} x_i(s_1) \\ x_i(s_2) \\ x_i(s_3) \end{bmatrix} \quad (6.2)$$

for $s_1 = \text{video}$, $s_2 = \text{voice}$, $s_3 = \text{besteffort}$.

A realisation $\mathbf{x}_i(\mathbf{s})$, of the random number of flows, is weighted with the average generated throughput per flow, $\mathbf{h}(\mathbf{s})$, for service type s . This results in the total throughput in the backbone for run i

$$t_{i,backbone} = \mathbf{x}_i(\mathbf{s})^T \cdot \mathbf{h}(\mathbf{s}) [Mb/s] \quad (6.3)$$

The expected value of the backbone utilisation over multiple runs, is then

$$\mathbb{E}[t_{i,backbone}] = \mathbb{E}[\mathbf{x}_i(\mathbf{s})^T \cdot \mathbf{h}(\mathbf{s})] \quad (6.4)$$

$$= \mathbb{E}[\mathbf{x}_i(\mathbf{s})^T] \cdot \mathbf{h}(\mathbf{s}) \text{ (scalar)} \quad (6.5)$$

$$= \mathbb{E}[x_i(s_1)] \cdot h(s_1) + \mathbb{E}[x_i(s_2)] \cdot h(s_2) + \mathbb{E}[x_i(s_3)] \cdot h(s_3). \quad (6.6)$$

This results in three variables to be designed, namely

$$\mathbb{E}[x_i(s_1 = \text{video})] \quad (6.7)$$

$$\mathbb{E}[x_i(s_2 = \text{voice})]$$

$$\mathbb{E}[x_i(s_3 = \text{besteffort})].$$

The random number of flows $x_i(s)$ is generated from a uniform distribution with the parameters $a_{u,s}$ and $b_{u,s}$ for a target utilisation u for service s . That is,

$$x_i(s) \sim \mathcal{U}(a_{u,s}, b_{u,s}). \quad (6.8)$$

Hence

$$\mathbb{E}[x_i(s)] = \frac{a_{u,s} + b_{u,s}}{2} \quad (6.9)$$

The expected number of flows per service type can then be designed by selecting arbitrary values for $a_{u,s}$ and $b_{u,s}$, in order to generate traffic for a desired average backbone utilisation. The designed values are listed in table 6.1.

Design parameters for random number of flows				
Utilisation, u [%]	Service type, s	$\mathbb{E}[x_i(s)]$ [flows]	$a_{u,s}$	$b_{u,s}$
30	Video	3	0	6
	Voice	12	9	15
	Best effort	3	0	6
50	Video	5	2	8
	Voice	20	17	23
	Best effort	5	2	8
60	Video	6	3	9
	Voice	24	21	27
	Best effort	6	3	9
70	Video	7	4	10
	Voice	28	23	29
	Best effort	7	4	10
80	Video	8	5	11
	Voice	32	29	35
	Best effort	8	5	11

Table 6.1: Traffic generation parameters for uniform random number of flows. The parameters are designed, such that the drawn random numbers all have the same variance.

The numbers in table 6.1 are designed, such that the variance of the drawn number of flows, remains constant for different utilisation levels and service types. That is, for all the pairs of $(a_{u,s}, b_{u,s})$, the resulting variance yields

$$\text{Var}(x_i(s)) = \frac{(b_{u,s} - a_{u,s} + 1)^2 - 1}{12} = 4.$$

The relation between the average number of flows for each service type, is kept constant over different utilisation levels. Video and Best effort have a 1 : 1 relation in generated flows, whereas the number of Voice traffic flows is 4 times larger than the number of Video flows. This is an arbitrary selected relation between the number of flows, which seemed a good fit, based on the average throughput for each service type flow. This also ensures that the number of flows are not skewed between utilisation levels, such that the relative number of samples for each service type is the same.

When a random number of flows has been generated, the flows are, 1-by-1, uniformly distributed among the 10 user nodes.

6.2.3 Create topology and start traffic

After the traffic realisations have been generated and distributed to the users, the Mininet topology is created. It deploys the two OVS switches, instantiates 10 virtual user nodes

and 3 service provider nodes and sets up the network connections between them according to the topology shown in figure 2.2 in section 2.3. At start-up, Mininet also orchestrates the connection set-up between the switches and the controller, which is already running in the background, such that the network is ready to be used.

When the topology is set-up, the backbone queue configuration is applied. In both scenarios, the output queues of the backbone interfaces are set to a maximum aggregated transmission rate of 100 [Mb/s], using configuration commands to the switches via the OVS configuration tool, *ovs-vsctl*. The static scenario creates one queue per interface, whereas the dynamic scenario sets-up 3 prioritised output queues per backbone interface as discussed in section 5.2. With the nodes and the network up and running, the relevant user nodes are assigned traffic receiver processes. When they are started, the traffic generator processes are instantiated at the service nodes, with the specified duration time for a run. After this, the traffic receiver and -generator processes are shut down, and the log files from all the traffic receivers are parsed and saved. The controller is also shut down, and the full procedure starts over, until 30 runs has been performed.

6.3 Results

This section presents the main results from the performed tests. The result discussion will primarily focus on the results generated with an average utilisation of 70 [%] and 80 [%] of the backbone link. The results include statistics for E2E delays and packet loss rates. Additional results can be found in appendix A which shows the results of running both the static and dynamic tests using 30, 50, 60, 70 and 80 [%] utilisation of the backbone link.

Table 6.2 shows the SLA requirements for the different traffic types, as determined in section 2.2.

Service type	E2E delay [ms]	Packet loss [%]	Throughput [Mb/s]
Voice	< 30	< 1	0.3
Video	< 1000	< 5	7
Best effort	< 400	< 5	2

Table 6.2: SLA requirements for each of the three traffic categories.

6.3.1 Utilisation: 70 [%]

Empirical CDF of E2E delays for Voice traffic.

Figure 6.2 shows the Empirical CDFs of E2E delays for voice traffic in the Static and Dynamic scenario, respectively. The plot is cropped to make the most interesting part apparent in the figure, thus the x-axis is set to the interval 0 – 100 [ms]. The legend shows the average load on the backbone, plus the number of packets included in the delay calculations for each of the CDFs.

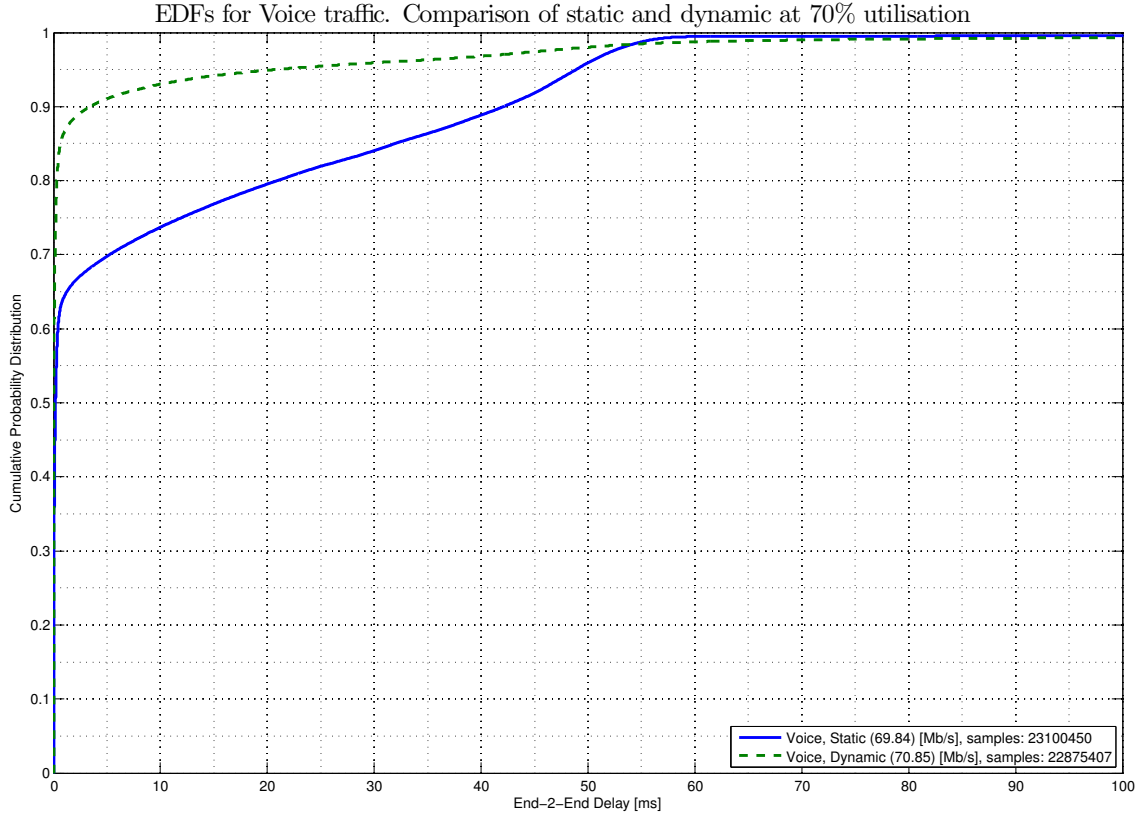


Figure 6.2: Delay distributions for Static and Dynamic at 70 % utilisation. Statistics are listed in table 6.3.

In both scenarios, the 50th percentile is below 1 [ms], but from around 60 [%] the curve for the static scenario starts to create its tail, whereas the tail for the dynamic curve does not start until 85 [%]. The specific statistics for the plot are listed in table 6.4. Comparing the 95th percentile (P_{95}) for voice, yields a margin of $48.92 - 20.75 = 28.17$ [ms] in favour of the dynamic scenario.

Further inspection of P_{95} values in table 6.4 also show, that the up-prioritisation of voice, has a cost regarding delays for video and best effort traffic. In the static scenario, all three services are near 50 [ms], but in the dynamic case the values for video and best effort increase. From the 95th percentile it can be seen that it is video that pays the highest price for lowering the delay of voice. Videos delay has been increased from around 50 [ms] up to 215 [ms], which is still below its SLA delay value of 1000 [ms]. For best effort the delay is increased from 50 [ms] up to around 80 [ms]. The reason why best efforts delay has not been increased as much as video is because of its SLA delay requirement of 400 [ms].

Both CDFs seem to be close to each other again from around 60 [ms]. That is, both scenarios are equally performing thereafter. This "upper bound" might be due to the nature of the delay measurements, in that only packets which are *not* lost can be included in the calculations. So the bound could indicate a saturation point of the network, and

thereby a worst-case delay for a packet, where the next packet in line will be tail-dropped due to a full queue.

Apart from queueing delay, measurement values can also be affected by the timing of OS kernel events, CPU scheduling, etc. This can be part of the explanation why the maximum delays are up to 1323 [ms].

Delay data at 70 % utilisation				
		Video	Best effort	Voice
Static	P₉₅ [ms]	51.65	49.50	48.92
	P₅₀ [ms]	0.80	0.26	0.19
	P₂₀ [ms]	0.03	0.03	0.03
	Avg. [ms]	14.82	11.56	10.68
	Min. [ms]	0.00	0.01	0.00
	Max. [ms]	1152.97	806.02	805.39
	Flows	208	224	774
	Packets	7727325	3380467	23100450
Dynamic	P₉₅ [ms]	214.29	82.18	20.75
	P₅₀ [ms]	0.33	0.07	0.04
	P₂₀ [ms]	0.03	0.02	0.02
	Avg. [ms]	46.01	12.97	4.31
	Min. [ms]	0.00	0.00	0.00
	Max. [ms]	1323.39	885.72	853.28
	Flows	218	218	765
	Packets	7960230	3239855	22875407

Table 6.3: Delay statistics for Static and Dynamic scenarios at 70 % utilisation. See CDF plot for voice in figure 6.2.

Empirical CDF of packet loss rates for traffic flows.

Figure 6.3 shows empirical CDFs of packet loss rates over all flows for each of the three service types in both static and dynamic scenarios. The legend carries the number of samples, i.e. flows, included for each CDF. Corresponding statistics for the figure is listed in table 6.4.

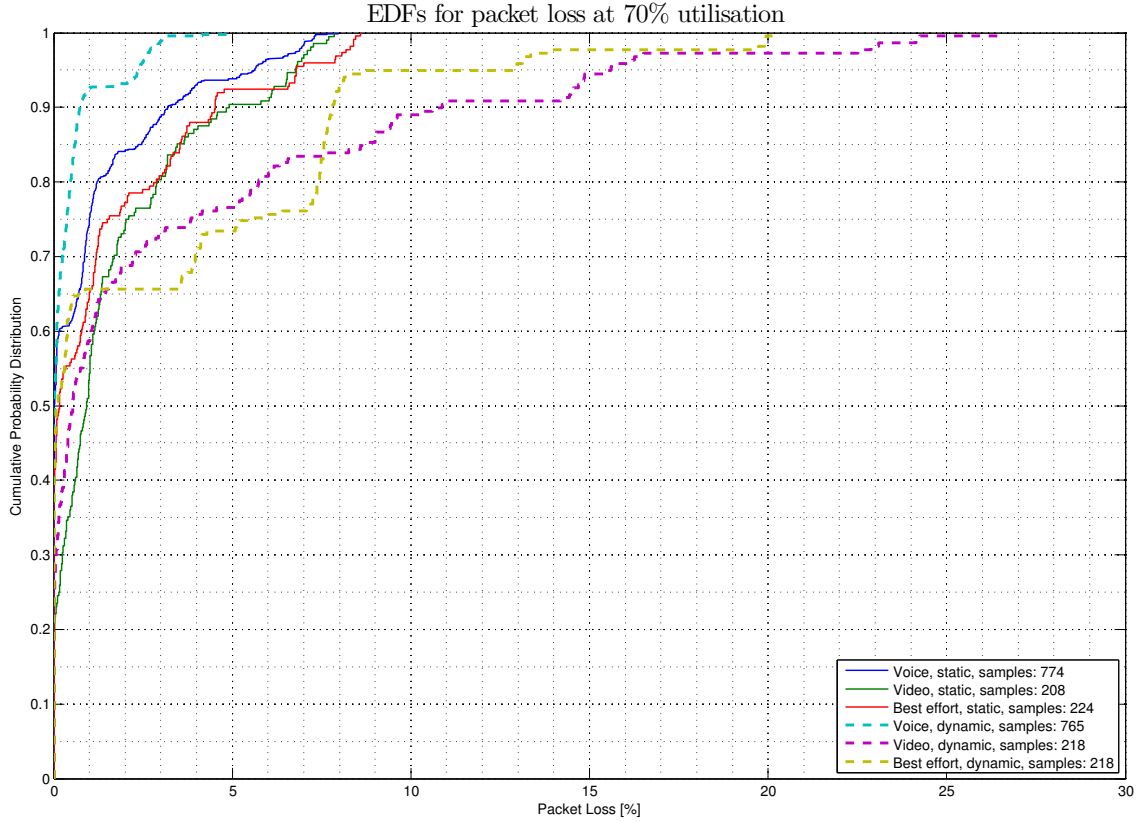


Figure 6.3: Packet loss distributions for all service types in both Static (—) and Dynamic (---) scenario at 70 [%] utilisation. Statistics are listed in table 6.4.

Figure 6.3 shows that in the static scenario, all three service type curves lie somewhat close to each other and within 5.62 – 6.76 [%] packet loss at the 95th percentile.

The dynamic scenario however, shows a notably higher packet loss rate for video and best effort traffic, while voice traffic performs better with a lower packet loss rate. This is indeed an expected result, and proves that the drop utility function (section 4.5) works as designed. That is, heavier flows, in terms of throughput, are more likely to be dropped than lighter flows, such as voice traffic. Furthermore, voice has a SLA packet loss rate of 1 [%] whereas video and best effort both have an SLA packet loss rate of 5 [%]. Looking at average packet loss rates for the dynamic scenario in table 6.4 it appears, that the SLA packet loss rates are fulfilled on average.

However P_{95} indicates that packet loss rates for all service types exceed their SLAs. This is also the case in the static scenario, where packet loss is evenly distributed amongst the service types. For the dynamic scenario the packet loss for voice has been reduced as a result of its more strict SLA requirement, video and best effort pay the price of reducing packet loss for voice traffic. Video pays the highest price as it is the service type that generates the largest amount of traffic, as the drop utility function was designed to.

Deeper analysis of the results reveals that the maximum observed packet loss rates occur

during realisations, where the generated traffic is very close to or exceeding the 100 [Mb/s] capacity limit of the backbone. Though it might seem like the FDQS is dropping packets a bit too aggressively when the utilisation of the backbone gets close to the target utilisation, which is set to 95 [%] for all dynamic results.

The dropping is based on the backbone port short-term throughput, which is calculated based on the two latest port rate samples. This results in large fluctuations in the measured throughput of a port. In order to make the FDQS module less aggressive in terms of dropping traffic, the short-term port throughput could be calculated as a moving average over several samples, as an attempt to mitigate the fluctuations in the measured throughput.

Packet loss data at 70 % utilisation				
		Video	Best effort	Voice
Static	P₉₅ [%]	6.76	6.79	5.62
	P₅₀ [%]	0.92	0.16	0.01
	P₂₀ [%]	0.03	0.00	0.00
	Avg. [%]	1.63	1.37	0.94
	Min. [%]	0.00	0.00	0.00
	Max. [%]	7.85	8.60	7.97
	Flows	208	224	774
	Packets	7727325	3380467	23100450
Dynamic	P₉₅ [%]	15.56	11.21	2.36
	P₅₀ [%]	0.53	0.11	0.02
	P₂₀ [%]	0.00	0.00	0.00
	Avg. [%]	3.22	2.78	0.33
	Min. [%]	0.00	0.00	0.00
	Max. [%]	26.55	20.18	4.93
	Flows	218	218	765
	Packets	7960230	3239855	22875407

Table 6.4: Packet loss statistics for Static and Dynamic scenarios at 70 % utilisation. CDF plot of packet loss rates for flows is depicted in figure 6.3

6.3.2 Utilisation: 80 [%]

Figure 6.4 shows the empirical CDFs for voice traffic at 80 [%] utilisation and figure 6.5 shows the empirical CDFs of packet loss for all service types in both the static and dynamic scenario.

What is interesting to see is that the performance of voice traffic is practically identical to what was seen at 70 [%] utilisation both in terms of delay and packet loss. Again it is video and best effort traffic that pays the price of maintaining the same performance level for voice traffic, but this time the price is significantly higher than what was seen for at 70 [%] utilisation because of the increase in traffic load.

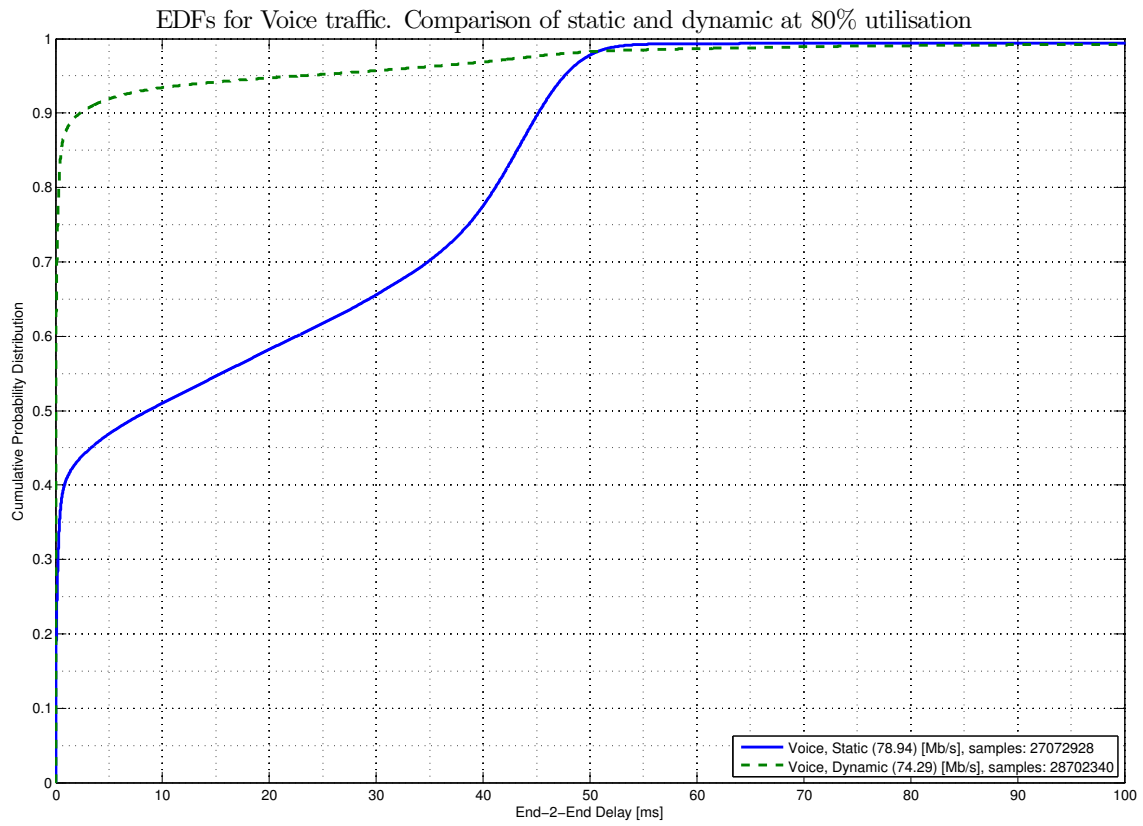


Figure 6.4: Delay distributions for Static and Dynamic at 80 [%] utilisation for 30 runs. Statistics are listed in table 6.5.

Delay data at 80 % utilisation				
		Video	Best effort	Voice
Static	P₉₅ [ms]	48.90	47.70	47.63
	P₅₀ [ms]	20.97	7.39	8.68
	P₂₀ [ms]	0.12	0.06	0.07
	Avg. [ms]	23.65	19.32	19.92
	Min. [ms]	0.00	0.01	0.01
	Max. [ms]	1187.51	817.28	904.69
	Flows	237	236	936
	Packets	8522857	3459213	27072928
	P₉₅ [ms]	238.84	140.81	22.76
	P₅₀ [ms]	1.07	0.08	0.03
Dynamic	P₂₀ [ms]	0.03	0.02	0.02
	Avg. [ms]	52.36	18.64	4.59
	Min. [ms]	0.00	0.01	0.00
	Max. [ms]	1032.58	913.98	916.25
	Flows	224	238	960
	Packets	8094772	3436081	28702340

Table 6.5: Delay statistics from Static and Dynamic scenario at 80 [%] utilisation from 30 runs. See figure 6.4 for CDFs of voice traffic.

It is worth noticing that the average throughput over 30 realisations for the dynamic scenario is 74.29 [Mb/s] while it is 78.94 [Mb/s] in the static scenario. This must be a result of not having generated enough realisations in order for the average throughput to converge to the target throughput. In terms of the results it means that a direct comparison between static and dynamic is unfair because they do not have the same average utilisation. However it is expected that the overall tendencies that can be observed for this test would have been the same, if the average throughput in both static and dynamic scenario had been the same.

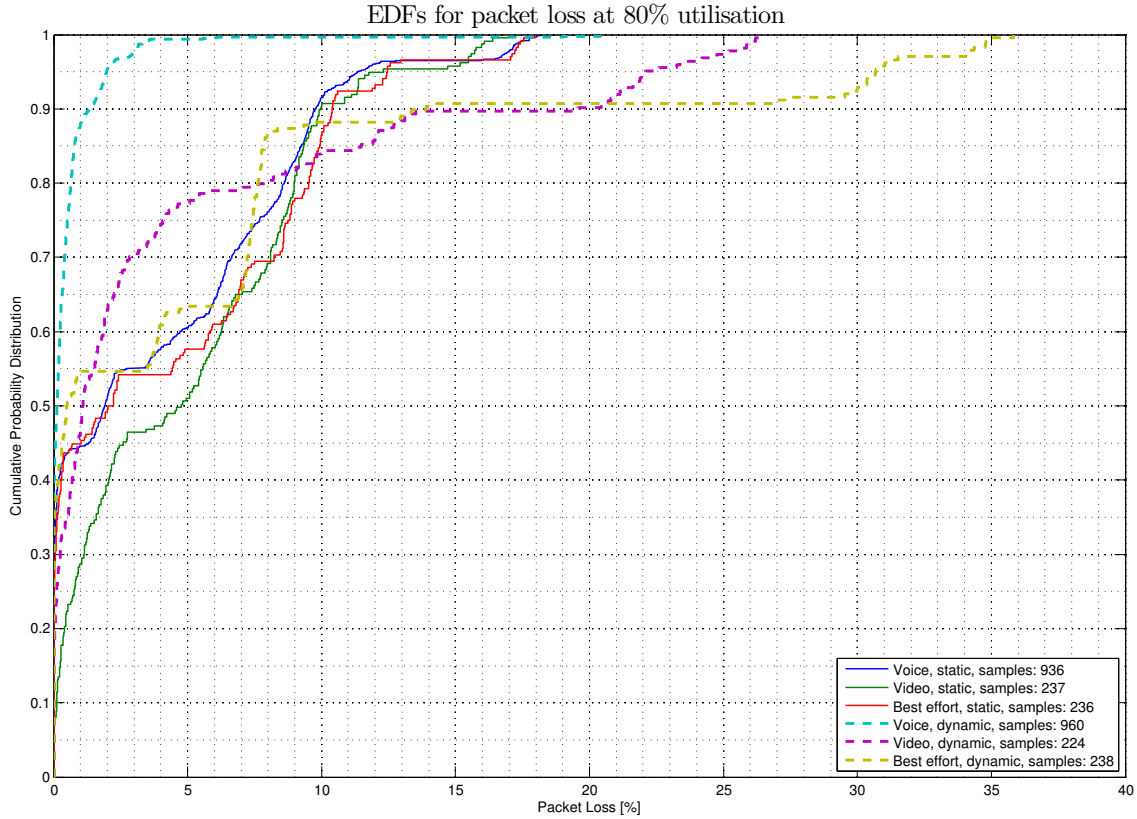


Figure 6.5: Packet loss distributions for Static and Dynamic at 80 [%] utilisation for 30 runs. Statistics are listed in table 6.6.

For packet loss, the average values of video and voice stay below their SLA packet loss values, but best effort exceeds its SLA values of 5 [%] slightly. Looking at the average values it can be seen that the dynamic scenario performs just as good as the static scenario for video and best effort, and that voice has been improved. However P_{95} show that non of the packet loss values are below their SLA values, but considering the somewhat extreme case a 80 [%] utilisation scenario is, this is to be expected when generating large amounts of bursty traffic. The packet loss SLAs were also not fulfilled in the 70 [%] utilisation test, so it would also not be expected that they would become any better by increasing the average utilisation.

The increase in packet loss rate does not necessary have to be a result of FDQS module actively dropping traffic, but because the interface buffers are not large enough to handle the bursts of traffic. This is what the packet loss values for the static scenario represents, because traffic is only dropped because of buffer overflows in that scenario.

Packet loss data at 80 % utilisation				
		Video	Best effort	Voice
Static	P₉₅ [%]	12.12	12.45	11.36
	P₅₀ [%]	4.78	2.13	1.91
	P₂₀ [%]	0.40	0.00	0.00
	Avg. [%]	4.95	4.43	4.01
	Min. [%]	0.00	0.00	0.00
	Max. [%]	16.97	17.98	18.15
	Flows	237	236	936
	Packets	8522857	3459213	27072928
	P₉₅ [%]	22.24	30.67	1.98
	P₅₀ [%]	1.08	0.49	0.13
Dynamic	P₂₀ [%]	0.04	0.00	0.00
	Avg. [%]	4.39	5.54	0.48
	Min. [%]	0.00	0.00	0.00
	Max. [%]	26.34	35.82	20.45
	Flows	224	238	960
	Packets	8094772	3436081	28702340

Table 6.6: Packet loss statistics from Static and Dynamic scenario at 80 [%] utilisation from 30 runs. See figure 6.5 for CDFs.

7 Conclusion

Based on the fact that the number of Internet-connected devices increases day by day, Internet Service Providers will have to keep up with this pace, to cope with the future customer needs. This demands new and innovative solutions, which is why this project focused on the problem statement:

“How to increase link utilisation in a Software-Defined Network, while adhering to constraints given by Quality of Service parameters for different service types?”

In a preliminary analysis in chapter 2, current trends of Internet traffic were inspected, to get an overview of the users needs in terms of service applications. This resulted in three main traffic categories; *Voice*, *Video* and *Best Effort*, which were characterised by individual Service Level Agreements (SLAs). To demonstrate the problem, a use case was set up, which clearly showed a correlation between End-To-End (E2E) delays and link utilisation.

After the preliminary analysis, a more in-depth analysis was performed in chapter 3, with particular focus on Software-Defined Networking (SDN), including aspects of how to interface with the network using the OpenFlow protocol. The analysis showed that the concept of Cognitive Networks would be an appropriate method for solving the problem, because it focuses on guaranteeing E2E goals for network traffic. Furthermore, the analysis showed that SDN is indeed a realisation of the Software Adaptive Networks, which Cognitive Networks rely on.

During the design in chapter 4, the OODA loop, which Cognitive Networks are based on, were explored. This exploration ended in the design of a controller module named Flow-based Dynamic Queue Selector (FDQS) consisting of two primary components; State Maintainer and Decision Logic. State Maintainer extracts and processes the state of the network using OpenFlow messages, while Decision Logic performs the actual control of the network based on the state of each individual flow and the state of the entire network. When Decision Logic makes a decision for a flow, it evaluates the state of the flow, in relation to its SLA using a delay utility function, such that it can ensure the fulfilment of the SLA End-To-End in the network. The delay requirement for a flow is fulfilled by moving flows between queues with different priorities. A flow is also evaluated using a drop-utility function. Its purpose is to find the best flow candidates to drop, when congestions happen in the network. The flow drop-candidates are then used when the evaluation of the entire network reveal congestion.

Finally, the performance of the proposed solution is evaluated in chapter 6, where the performance of the implemented FDQS module is compared to the results of the reference scenario, that was proposed as part of the use case scenario in section 2.3. The performance evaluation shows that the Decision Logic algorithm works as intended. It is capable of guaranteeing a delay requirement of 30 [ms] for up to 96 [%] of all voice traffic flows at 70 [%] average link utilisation, where the reference scenario is only capable of guaranteeing 84 [%]. This comes however at a price, which video and best effort traffic has to pay, by having an increased delay compared to the reference scenario, both are however still below their SLA requirements at the 95th percentile. The results show that best effort is performing better than video, which is a result of a SLA requirement of 400 [ms] delay for best effort and 1000 [ms] of delay for video.

In terms of packet loss the results show that the drop utility function ensures that video and best effort traffic is more likely to be dropped compared to voice, because voice traffic has a significantly smaller footprint in terms of throughput. Neither the reference solution nor the FDQS module is able to guarantee the SLA packet loss rates at the 95th percentile for any of the service types, but FDQS is able to achieve a packet loss rate less than 1 [%] for up to 92 [%] of all voice traffic flows. The results also show that the FDQS module have a tendency to be too aggressive when dropping flows, which is determined to be caused by port throughput measurements being too fluctuating.

In order to build intelligent control algorithms for an SDN network, a proper testing environment is needed. This includes an automated test bed, which both supports reproducibility and eliminates mistakes introduced by human interference. Therefore, a big part of the work during this project, has been to create an environment for automating tests of the FDQS module.

Because this work is based on an emulated network, with a real SDN controller, it would actually be possible to apply the proposed solution directly on a real network, using physical switches and links.

The presented work takes a new and innovative approach to QoS in an IP-based network. It has been shown that it is possible to create a solution based on SDN, which is capable of ensuring that different SLA requirements for different service types are fulfilled, by using the global network state as basis for the applied control plane logic. This ensures, that when the average link utilisation is increasing, the traffic flows are scheduled, such that the requirements for each individual flow is taken into account, to find a better overall configuration.

Bibliography

- Allied Telesis (2005). Resource reservation protocol (chapter 40). http://www.alliedtelesis.com/media/fount/software_reference/271/ar400/Rsvp.pdf. AR400 Software reference, Software Release 2.7.1, Document version: C613-03091-00 REV A.
- Botta, A., Dainotti, A., and Pescapè, A. (2012). A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531–3547.
- Chen, Y., Farley, T., and Ye, N. (2004). Qos requirements of network applications on the internet. *Inf. Knowl. Syst. Manag.*, 4(1):55–76.
- Cisco (2013). Connections counter: The internet of everything in motion. News blog: <http://newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342>. Accessed on 22. Mar 2014.
- Evans, J. W. and Filsfils, C. (2007). *Deploying IP and MPLS QoS for Multiservice Networks: Theory & Practice*. Morgan Kaufmann. eISBN: 9780080488684.
- Joseph, V. and Chapman, B. (2009). *Deploying QoS for Cisco IP and Next-Generation Networks: The Definitive Guide*. Morgan Kaufmann, 1st edition. ISBN: 978-0-12-374461-6.
- Kotani, D. and Okabe, Y. (2012). Packet-in message control for reducing cpu load and control traffic in openflow switches. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 42–47.
- Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA. ACM.
- McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.
- Nanduri, M., Kasten, M., and Kitajima, N. (2012). Mpls traffic engineering with auto-bandwidth: Operational experience and lessons learned. http://meetings.apnic.net/_data/assets/pdf_file/0020/45623/MPLS-Traffic-Engineering-with-Auto-Bandwidth.pdf. Accessed on 25. Mar 2014.

- Open Networking Foundation (2009). Openflow switch specification. On-line:<https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>. Accessed on 22. May 2014.
- Phemius, K. and Bouet, M. (2013). Monitoring latency with openflow. In *Network and Service Management (CNSM), 2013 9th International Conference on*, pages 122–125.
- Rao, A., Legout, A., Lim, Y.-s., Towsley, D., Barakat, C., and Dabbous, W. (2011). Network characteristics of video streaming traffic. In *Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 25:1–25:12, New York, NY, USA. ACM.
- Sandvine (2013). Global internet phenomena report - h2 2013. <https://www.sandvine.com/downloads/general/global-internet-phenomena/2013/2h-2013-global-internet-phenomena-report.pdf>.
- Steenbergen, R. A. (2013). Mpls rsvp-te auto-bandwidth: Practical lessons learned. NANOG 58 meeting - Slides:<https://www.nanog.org/sites/default/files/tues.general.steenbergen.autobandwidth.30.pdf> and Presentation:https://www.nanog.org/sites/default/files/mpls_rsvp-te_auto-bandwidth_-_lessons_learned.mp4. Accessed on 25. Mar 2014.
- Szigeti, T. and Hattingh, C. (2004). *End-to-End QoS Network Design: Quality of Service in LANs, WANs, and VPNs*. Cisco Press, 1st edition.
- Tanenbaum, A. S. and Wetherall, D. J. (2011). *Computer Networks*. Prentice Hall Professional Technical Reference, 5th edition. ISBN: 978-0-13-255317-9.
- TeleGeography (2014). Skype traffic continues to thrive. <http://www.telegeography.com/press/press-releases/2014/01/15/skype-traffic-continues-to-thrive/index.html>. Accessed on 26. Feb 2014.
- Thomas, R., DaSilva, L., and MacKenzie, A. (2005). Cognitive networks. In *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 352–360.
- van Adrichem, N. L., Doerr, C., and Kuipers, F. A. (2014). Opennetmon: Network monitoring in openflow software-defined networks.

A Result tables

A.1 Static scenario at different utilisation levels

This section contains the data from the use case scenario in section 2.3.

Delay data for static scenario at different utilisations				
		Video	Best effort	Voice
Util: 30 [%]	P_{95} [ms]	0.16	0.24	0.16
	P_{50} [ms]	0.04	0.05	0.05
	P_{20} [ms]	0.03	0.03	0.03
	Avg. [ms]	0.69	0.14	0.19
	Min. [ms]	0.01	0.01	0.01
	Max. [ms]	894.17	281.13	371.24
	Flows	84	78	346
	Packets	3146825	1180345	10280469
Util: 50 [%]	P_{95} [ms]	2.20	0.63	0.51
	P_{50} [ms]	0.04	0.05	0.04
	P_{20} [ms]	0.03	0.03	0.03
	Avg. [ms]	1.39	0.64	0.89
	Min. [ms]	0.00	0.00	0.00
	Max. [ms]	940.98	631.55	650.69
	Flows	141	137	608
	Packets	5288380	2091369	18161304
Util: 60 [%]	P_{95} [ms]	33.90	29.44	24.91
	P_{50} [ms]	0.08	0.09	0.07
	P_{20} [ms]	0.03	0.03	0.03
	Avg. [ms]	5.49	4.28	3.92
	Min. [ms]	0.00	0.01	0.00
	Max. [ms]	957.89	599.43	759.51
	Flows	184	197	711
	Packets	6932082	3002749	21331107
Util: 70 [%]	P_{95} [ms]	51.65	49.50	48.92
	P_{50} [ms]	0.80	0.26	0.19
	P_{20} [ms]	0.03	0.03	0.03
	Avg. [ms]	14.82	11.56	10.68
	Min. [ms]	0.00	0.01	0.00
	Max. [ms]	1152.97	806.02	805.39
	Flows	208	224	774
	Packets	7727325	3380467	23100450
Util: 80 [%]	P_{95} [ms]	48.90	47.70	47.63
	P_{50} [ms]	20.97	7.39	8.68
	P_{20} [ms]	0.12	0.06	0.07
	Avg. [ms]	23.65	19.32	19.92
	Min. [ms]	0.00	0.01	0.01
	Max. [ms]	1187.51	817.28	904.69
	Flows	237	236	936
	Packets	8522857	3459213	27072928

Table A.1: Delay statistics for the static scenario at different utilisation levels.

A.2 Dynamic scenario at different utilisation levels

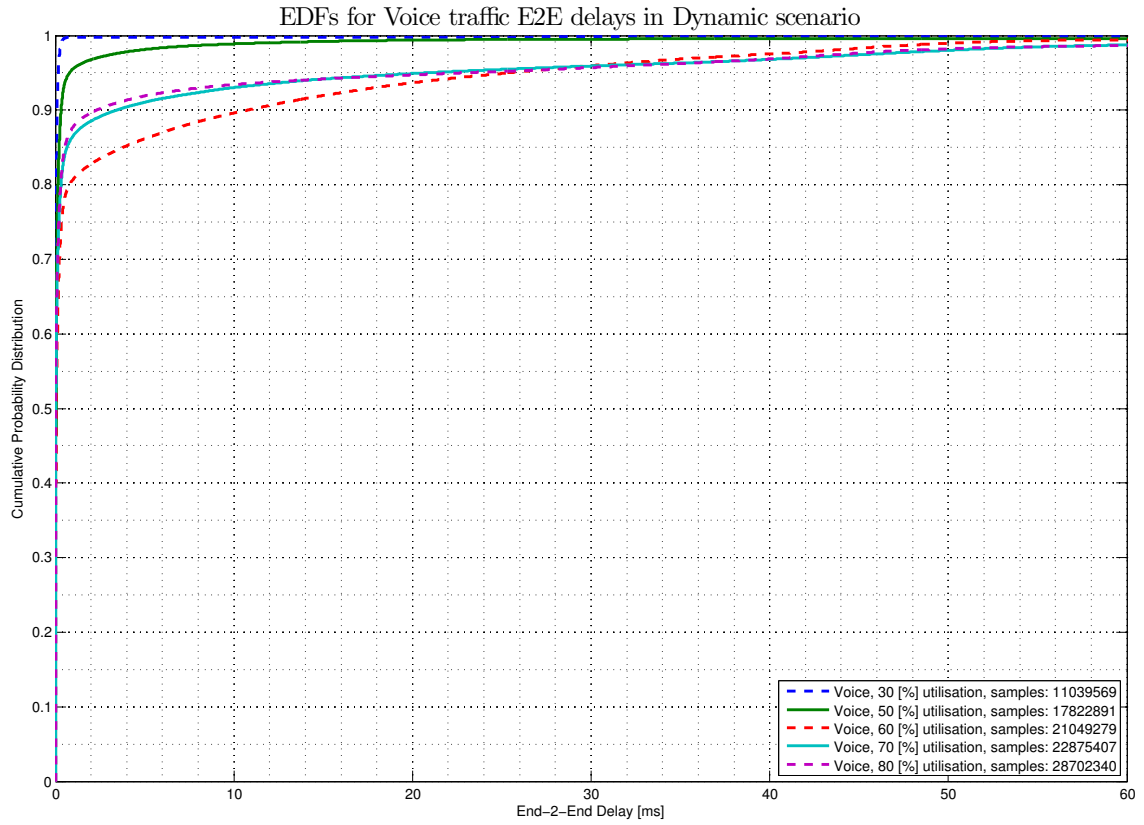


Figure A.1: Delay distributions for Dynamic at different utilisation levels from 30 runs.

Delay data for Dynamic scenario at different utilisation levels				
		Video	Best effort	Voice
Util: 30 [%]	P₉₅ [ms]	0.15	0.22	0.16
	P₅₀ [ms]	0.04	0.05	0.04
	P₂₀ [ms]	0.03	0.03	0.03
	Avg. [ms]	0.68	0.22	0.29
	Min. [ms]	0.00	0.00	0.00
	Max. [ms]	940.97	333.13	414.28
	Flows	95	100	371
	Packets	3574226	1519778	11039569
Util: 50 [%]	P₉₅ [ms]	2.76	1.39	0.79
	P₅₀ [ms]	0.04	0.05	0.04
	P₂₀ [ms]	0.02	0.03	0.03
	Avg. [ms]	1.41	1.17	1.25
	Min. [ms]	0.00	0.00	0.00
	Max. [ms]	1072.66	582.80	669.17
	Flows	154	149	596
	Packets	5795406	2270949	17822891
Util: 60 [%]	P₉₅ [ms]	37.27	23.06	25.37
	P₅₀ [ms]	0.07	0.06	0.06
	P₂₀ [ms]	0.03	0.03	0.03
	Avg. [ms]	7.03	3.83	4.38
	Min. [ms]	0.00	0.00	0.00
	Max. [ms]	1263.25	764.42	802.38
	Flows	190	192	703
	Packets	7163659	2909450	21049279
Util: 70 [%]	P₉₅ [ms]	214.29	82.18	20.75
	P₅₀ [ms]	0.33	0.07	0.04
	P₂₀ [ms]	0.03	0.02	0.02
	Avg. [ms]	46.01	12.97	4.31
	Min. [ms]	0.00	0.00	0.00
	Max. [ms]	1323.39	885.72	853.28
	Flows	218	218	765
	Packets	7960230	3239855	22875407
Util: 80 [%]	P₉₅ [ms]	238.84	140.81	22.76
	P₅₀ [ms]	1.07	0.08	0.03
	P₂₀ [ms]	0.03	0.02	0.02
	Avg. [ms]	52.36	18.64	4.59
	Min. [ms]	0.00	0.01	0.00
	Max. [ms]	1032.58	913.98	916.25
	Flows	224	238	960
	Packets	8094772	3436081	28702340

Table A.2: Delay statistics for Dynamic scenarios at different utilisation levels from 30 runs.

Packet loss data for Dynamic scenario at different utilisation levels				
		Video	Best effort	Voice
Util: 30 [%]	P₉₅ [%]	0.26	0.00	0.00
	P₅₀ [%]	0.00	0.00	0.00
	P₂₀ [%]	0.00	0.00	0.00
	Avg. [%]	0.04	0.00	0.00
	Min. [%]	0.00	0.00	0.00
	Max. [%]	0.50	0.00	0.00
	Flows	95	100	371
	Packets	3574226	1519778	11039569
Util: 50 [%]	P₉₅ [%]	0.89	0.23	0.02
	P₅₀ [%]	0.04	0.00	0.00
	P₂₀ [%]	0.00	0.00	0.00
	Avg. [%]	0.21	0.04	0.00
	Min. [%]	0.00	0.00	0.00
	Max. [%]	1.11	0.43	0.21
	Flows	154	149	596
	Packets	5795406	2270949	17822891
Util: 60 [%]	P₉₅ [%]	1.50	4.29	0.51
	P₅₀ [%]	0.22	0.00	0.00
	P₂₀ [%]	0.00	0.00	0.00
	Avg. [%]	0.48	0.68	0.13
	Min. [%]	0.00	0.00	0.00
	Max. [%]	3.88	11.08	2.19
	Flows	190	192	703
	Packets	7163659	2909450	21049279
Util: 70 [%]	P₉₅ [%]	15.56	11.21	2.36
	P₅₀ [%]	0.53	0.11	0.02
	P₂₀ [%]	0.00	0.00	0.00
	Avg. [%]	3.22	2.78	0.33
	Min. [%]	0.00	0.00	0.00
	Max. [%]	26.55	20.18	4.93
	Flows	218	218	765
	Packets	7960230	3239855	22875407
Util: 80 [%]	P₉₅ [%]	22.24	30.67	1.98
	P₅₀ [%]	1.08	0.49	0.13
	P₂₀ [%]	0.04	0.00	0.00
	Avg. [%]	4.39	5.54	0.48
	Min. [%]	0.00	0.00	0.00
	Max. [%]	26.34	35.82	20.45
	Flows	224	238	960
	Packets	8094772	3436081	28702340

Table A.3: Packet loss statistics for Dynamic scenarios at different utilisation levels from 30 runs.

B Resource allocation techniques

This appendix chapter contains a perspective on current deployments of QoS and resource allocation techniques, which include IntServ, DiffServ and MPLS. The chapter ends with a conclusive summary of their efficiency characteristics.

B.1 Integrated services (IntServ)

IntServ was one of the earlier approaches, designed to guarantee requirements according to SLAs [Evans and Filsfils, 2007, pp. 303–304]. IntServ delivers an E2E reservation of bandwidth resources in the network on a per flow basis. This requires flow state information on each hop on the IntServ path, in order to accommodate the specified SLA requirements and schedule the packets in each flow, accordingly.

To control whether resources can be reserved for a flow or not, a signalling protocol is needed to distribute the flow state information. This admission control is mostly implemented using the Resource reSerVation Protocol (RSVP). It uses a *Path* message, which is sent from the sending host, to the receiving host [Allied Telesis, 2005, pp.40-3–40-5]. Every IntServ compatible router on the way, stores the previous hop address from the message and writes its own address instead. Thus, storing flow state information on every hop. Finally, the router forwards the message according to the underlying routing protocol. When the message arrives at the receiver, it sends back a *Resv* message, which contain the actual reservation. All hops along the path, then reserves the requested resources, according to the content in the *Resv* message.

One of the main issues of IntServ is the internal storage of flow states. It does not scale well for larger networks and the reservations cannot survive router outages. Therefore, the actual number of IntServ deployments remain rather limited [Tanenbaum and Wetherall, 2011, p.439].

B.2 Differentiated services (DiffServ)

The most popular technique for applying QoS in IP networks is DiffServ [Evans and Filsfils, 2007, p. 150], which takes on a slightly simpler approach. Instead of flows, it is based on classes. Traffic is administratively defined into a limited set of service classes, which are identified using the 6-bit DSCP field in the IP-header [Evans and Filsfils, 2007,

p.149]. Either the traffic is marked directly from the source, or by a router when entering the DiffServ domain. This pushes complexity to the edges of the network and is thereby deemed more scalable.

Each hop in the DiffServ domain is independently configured with a set of forwarding rules for the predefined traffic classes. That is, the classes follow a so-called Per Hop Behaviour (PHB), which means that the resource allocation is controlled individually by each of the hop instances, by using queuing and scheduling disciplines.

The simplest PHB scheme is Expedited Forwarding (EF), where the traffic is characterised as either regular or expedited [Tanenbaum and Wetherall, 2011, pp.440–442]. If there is any traffic in the expedited queue, then it is forwarded before servicing the queue with regular traffic.

A more fine-grained PHB scheme, called Assured Forwarding (AF), defines 4 priority classes, where traffic in each class is further prescribed a prioritisation into 3 different discard classes. The routers can then allocate resources for the each of the priority classes and if necessary, start dropping packets within each of the classes, based on the assigned discard class.

Compared to IntServ, resources cannot be guaranteed E2E, but this is outweighed by the simpler set-up DiffServ provides, without the need for a reservation protocol like RSVP.

B.3 Multi-Protocol Label Switching - Traffic Engineering

In larger IP-networks, e.g. those owned by ISPs, MPLS is a widely deployed technology [Tanenbaum and Wetherall, 2011, pp.488–492]. When an IP-packet enters the MPLS domain, a Label Edge Router (LER) inspects the IP-packet to determine the destination, and then adds a label to the packet, by encapsulating it in a separate header. The label is then used at the next hop within the MPLS domain, i.e. the Label Switched Router (LSR), to determine the further path of the packet. When the packet reaches the other edge of the MPLS domain, the label header is popped of and the packet is forwarded to the next network.

Using labels to determine packet routes, simplifies the switching process to a table lookup with the label as index, hence it is relatively simple and thereby also fast. Furthermore, this pushes the complexity of traffic classification to the edges of the network, making the core simple and effective.

The packet routes are identified using labels and therefore also referred to as LSPs, which are set up using upper layer control protocols. Commonly, MPLS routers aggregate flows into different groups, each called a Forwarding Equivalence Class (FEC). These map to a specific label, thus the FECs can be compared to the service classes in DiffServ.

With MPLS as the basic network technology, several network protocols, have been extended to also provide mechanisms for Traffic Engineering (TE). One of them is RSVP-TE, where the traffic engineering functionalities allows LSPs to be set up, according to available bandwidth on the links in the MPLS domain. This allocates resources through-

out the network, for the particular path, which in this context is also referred to as traffic engineering tunnels. Effectively, this means that instead of always using the shortest path, traffic can be switched using a longer, but less congested path in the MPLS domain. That is, if the shortest path does not have enough available bandwidth, then the second shortest path is utilised, given that it has the available bandwidth free to use. This functionality can help load balancing the traffic in the network.

The calculations of these LSP-wise bandwidth allocations, can either be done off- or on-line. Network operators can perform the offline calculation by using some 3rd party planning tool and deploy new label configurations afterwards, where online calculations are internally performed by the routers.

This online solution, is also known as the *automatic bandwidth* feature and provides some degree of adaptiveness to the resource allocation [Steenbergen, 2013]. In general, it works by using a specified adjust time interval to periodically check the bandwidth usage for the LSP in the previous adjust interval, and then allocate bandwidth based on the maximum observed utilisation value [Nanduri et al., 2012]. This means that if more bandwidth is needed and the current path has this available, the LSP gets more resources assigned on its existing path. But if there is no room for further bandwidth allocation, the LSP is rerouted via another set of links.

Dependent on the specific vendor implementation, the feature is further controlled by different parameters for update thresholds, minimum and maximum bandwidth allocation etc.

B.4 Efficiency characteristics

The above-mentioned approaches have characteristics that are worth noting, when analysing efficient resource allocation in IP-networks. Regarding scalability, IntServ suffers in larger networks, due to the flow-based state information base. That is part of the reason why it is practically not in use. DiffServ supersedes this by aggregating traffic into a limited set of classes, hence limiting the need for local storage of information about managing separate traffic flows. MPLS networks are also typically aggregating the traffic into service classes, namely the FECs, thereby also limiting the needed state storage for the LSP tunnels.

Both MPLS-TE and IntServ offer bandwidth allocation guarantees End-To-End (E2E), where DiffServ only ensures resources one hop at a time. In order to guarantee SLA requirements E2E in a DiffServ set up, the network operator needs to perform thorough capacity planning, before the set up can be deployed. Unless the operator has a centralised provisioning platform, tuning of the QoS parameters afterwards, would require a considerable amount of work, in order to walk through the devices and re-configure each of them. Such a scenario will most likely be sub-optimal with respect to changing traffic conditions, due to its somewhat static configuration.

This exemplifies that simplicity often comes at a cost. However, DiffServ can be combined with MPLS-TE to gain benefit from both techniques [Evans and Filsfils, 2007, p.404]. TE is used for setting up the E2E tunnels for an aggregated set of traffic flows, while DiffServ can prioritise the service classes within the TE tunnel. This concept is also

referred to as DiffServ-aware MPLS-TE.

Maybe the most important aspect regarding efficient resource allocation is adaptability. A static allocation may be good on average. However, if the traffic conditions are varying, then the static configuration can end up being unfit. MPLS-TE seeks to accommodate this via the automatic bandwidth feature.

As depicted in figure B.1, a pitfall of the feature is that if a short peak of traffic appear, it can induce a high allocation at the next bandwidth adjustment. This may result in an over-booking of the bandwidth and thereby blocking for other LSPs to settle on the link, hence the result is inefficient use of the available bandwidth.

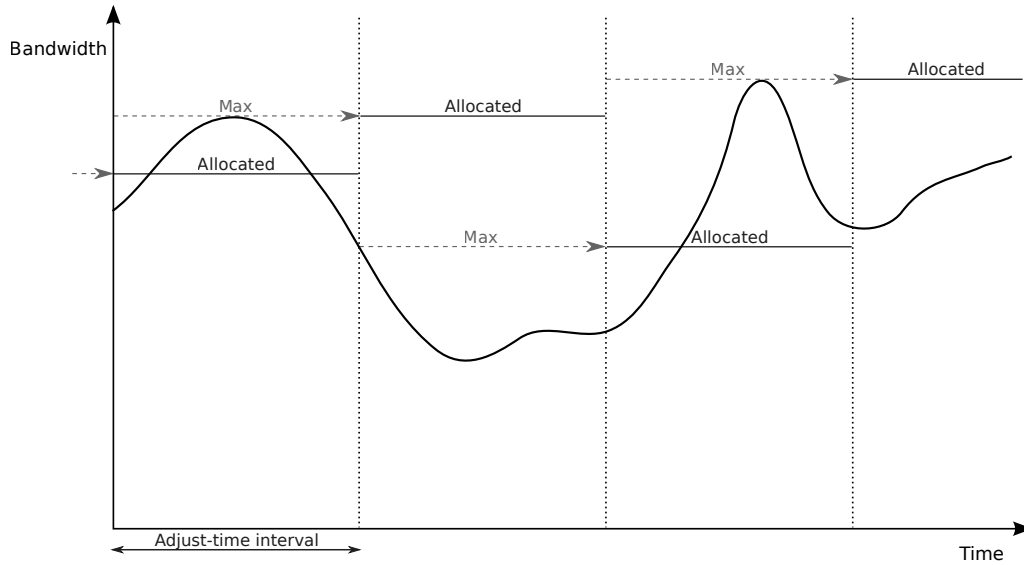


Figure B.1: Automatic bandwidth feature, allocating bandwidth every adjust-time interval, based on maximum of measured bandwidth.

In general, automatic bandwidth is based on simple counters and does not know about the surrounding conditions in the network [Steenbergen, 2013, 16min25sec]. For example, if a link gets congested somewhere in the network and packets are dropped, this will cause TCP traffic to throttle down and thereby also the bandwidth utilisation on the link. Effectively, leading to lower numbers in the counters, which will make automatic bandwidth lower its resource allocation, on a false basis [Steenbergen, 2013, p.22].

Another dependency for the auto bandwidth feature is that it relies on the possibility of using an alternative path for re-routing the LSPs, which might not always be possible.

To sum up, what can be learned from the current resource allocation techniques is, that in order to be successful, a more efficient solution should be able to scale well. Furthermore, using static Per Hop Behaviour (PHB) independently on each of the FEs, will not be a feasible solution in environments with varying traffic conditions. Last but not least, a system needs to be aware of the global network state, in order to perform proper decisions and adaptations.

C Generating traffic

There are several different ways to generate traffic in a test network. One way is to generate “real” traffic, meaning that videos will be streamed from Netflix, actual Skype calls will be made, etc. This approach has the disadvantage that it is not very flexible and it becomes difficult to change the setup between tests. With the scenario presented in section 2.3, where a 100 [Mb/s] link is used, it would require a lot of 7 [Mb/s] video streams and 300 [Kb/s] voice calls to reach the capacity of the links. This approach is therefore not feasible for the use case scenario.

Another approach is to record actual streams of data for the different categories, and then playback the streams during the tests. This approach is more flexible and scales better than the first mentioned. However some work will have to be put into creating a setup for recording traces and generating representative traces for the different traffic categories. If the results of the tests are to have any statistical significance, this approach becomes limited because, it requires a lot of different streams to be recorded, in order for all streams to not behave exactly the same.

A final approach is to generate synthetic traffic, based on stochastic models for the different traffic categories. This requires some insight into how the different traffic categories act, and what characterises them in terms of stochastic models. When generating synthetic traffic, the generated traffic will never be exactly like traffic that could be encountered in real life, because all models have their limitations. Some models focus on the users request of data, others focus on payload sizes or inter-arrival time distributions, while other models focus on the correlation between packets over time. There are a lot of traffic generators out there. The traffic generated in this work is based on using the traffic generator developed by Botta et al. [2012] called Distributed-Internet Traffic Generator (D-ITG), as it is one of the few capable of matching the requirements for traffic generation that will be explained in the following.

C.1 Video

The network characteristics of Netflix, amongst others, have been investigated by Rao et al. [2011]. They show that the characteristics of a Netflix stream matches the general video streaming model shown in figure C.1.

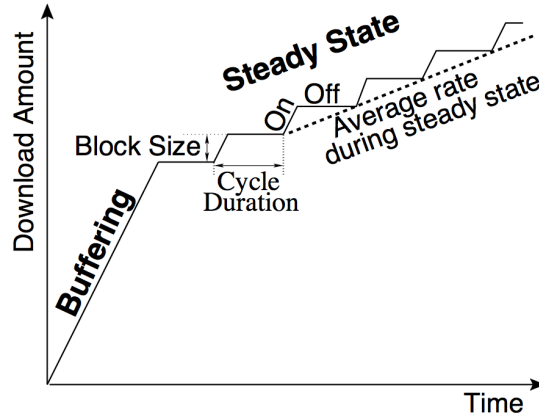


Figure C.1: General model of video streaming behaviour [Rao et al., 2011]

The model shows that a stream is characterised by an initial burst followed by periodic ON-OFF cycles. The burst period is used to buffer up the video content in different resolutions at the user, such that the video player can determine the appropriate resolution for the video given the user's amount of available bandwidth. Once the buffering period is over, the stream is characterised by an ON-OFF model which maintains a certain amount of data in the player buffer. Parameters of the ON-OFF cycles are given by the average bit rate required to stream the remaining video content. The periodic ON-OFF cycles are defined such that no traffic is transmitted during an OFF period. During an ON period a certain amount of data will be transferred, before entering an OFF period again.

The investigation of Netflix by Rao et al. [2011] presents the CDF shown in figure C.2.

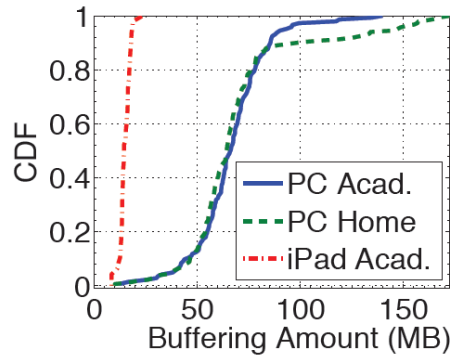


Figure C.2: CDF over the amount of buffering done by different Netflix players and on different networks [Rao et al., 2011]. PC Acad. is streaming to a PC on a university network, while PC Home is streaming to a PC at home. iPad Acad. represents streaming to an iPad using a university network.

The CDF shown that the amount of buffering for a Netflix stream is somewhere between

50 and 100 [MB]. The CDF is based on 200 randomly selected Netflix videos.

Figure C.3 shows a CDF of the block size (amount of data) transferred during the ON period for 200 randomly selected Netflix videos.

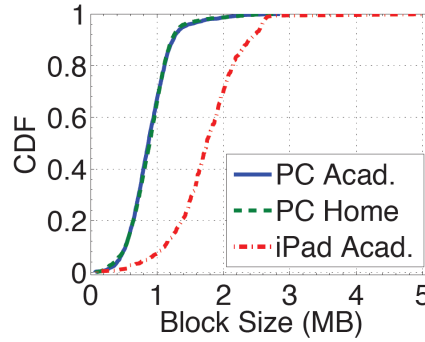


Figure C.3: CDF over the block size of the data transferred during the ON period by different Netflix players and on different networks [Rao et al., 2011].

The CDF of the block size shows that the majority of the used block sizes are between 0.5 to 1.5 [MB].

For the purpose of this work, this definition of video is not considered to be bursty enough. A slightly different approach to generating video is therefore proposed, which also relies on the traffic being generated as an ON-OFF process, but where the ON and OFF times are exponentially distributed, which results in a Markovian ON-OFF process, with constant payload size equal to the MTU of the network, e.g. 1500 bytes. The traffic analysis in section 2.2 determined that the average throughput of a video flow should be around 7[Mb/s] . The exponential parameters of the Markovian ON-OFF process must therefore be designed such the generated traffic matches the target. This is done by allowing the flow to burst more than 7[Mb/s] during an ON period, because no traffic will be transmitted during an OFF period. The traffic generated used for this work is limited to being able to burst 10[Mb/s] during an ON period. The exponentially distributed ON-OFF times are then designed based on these constraints. This will give the video traffic a more bursty behaviour.

C.2 Voice

Voice traffic can be generated as a flow with constant bit rate or as a flow with a variable bit rate. For tele networks voice is often encoded using ITU-T standards G.7xx, which specify different bit rates and encoding schemes.

For the sake of simplicity it has been chosen to generate voice with a constant bit rate. The target bit rate is set to 300[Kb/s] , which was the rate that was determined in section 2.2. The constant voice flow is then generated using a constant payload size of 75[B] , and a constant IDT of 524[packets/s] such that the target throughput is reached.

C.3 Best effort

The best-effort category is included to use bandwidth and as cross traffic in the network. It is not possible to create one definitive definition of how this traffic type should be generated, because of larger number of protocols contained in it. It has therefore been decided to generate best effort traffic with exponentially distributed payload sizes and exponentially distributed IDTs. The target throughput of best effort is set to $2[Mb/s]$, the payload size is generated with a mean of 1000 [Byte] and the IDT is generated with a mean of 262 [packets/s]. This will also make best effort traffic bursty, but not quite as bursty as video traffic.