

Master of Science in Informatics Engineering  
MSc Thesis 2014/2015  
Final Report

# Development of an OpenFlow controller application for enhanced path computation

Ricardo Ramalho dos Santos  
rrsantos@student.dei.uc.pt

Advisors:

Marília Curado

Andreas Kessler

Date: July 6th, 2015



**FCTUC** DEPARTAMENTO  
DE ENGENHARIA INFORMÁTICA  
FACULDADE DE CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE DE COIMBRA



UNIVERSITY OF COIMBRA

MASTER OF SCIENCE IN INFORMATICS  
ENGINEERING

FINAL REPORT

---

# Development of an Openflow application for enhanced path computation

---

*Author:*

Ricardo Ramalho dos Santos

*Advisors:*

Dr. Marília Curado

Dr. Andreas Kessler

July 6, 2015



## *Acknowledgements*

I would like to express my gratitude to Marília Curado which, besides being my advisor for this thesis, introduced me to the LCT and to the world of research. It has been quite a journey until now and, when looking back, I can see how much I have done and achieved. Thank you also for all the support, criticism and feedback, which were always crucial for improving all my outcomes.

Thank you Bruno Sousa, working with you during my first steps in LCT was a very important experience that surely contributed for the increase of my interest in computer networks.

I express my gratitude to my co-advisor Andreas Kassler, for all the mentoring before and during this thesis. Sweden has definitely been a more pleasant adventure by having your guidance. A special thanks goes also to the University of Karlstad for the excellent conditions that were offered to me while I was there.

I thank my parents for all the support in all my decisions, either inside and outside my academic career. You have always been by my side, even when I did not realize that.

To my sister, your conquests and your ability of overtaking all the obstacles in your life so far are a big motivation to me, and for that, I show you my appreciation.

Finally, last but not least, a special thanks to all my friends that, wherever I was, made me feel at home.



## *Resumo*

Redes definidas por software (SDN) é um paradigma das redes informáticas recente que permite uma gestão centralizada dos dispositivos de rede através de uma entidade (controlador), responsável pelo processamento dos dados recebidos e instalação de fluxos nos dispositivos geridos. Ao trabalhar num ambiente SDN, é possível monitorizar o estado atual de uma rede e calcular caminhos para fluxos novos, de acordo com esse estado.

Adicionalmente, quando caminhos diferentes estão disponíveis, e usando um algoritmo para o cálculo de caminhos com múltiplas restrições, vários caminhos podem ser calculados, seguindo as restrições impostas pelas métricas recolhidas a partir da rede. Combinando o uso de protocolos de transporte *multi-homed*, como o Multipath TCP (MPTCP), os múltiplos sub-fluxos criados podem ser atribuídos a diferentes caminhos, contribuindo para o aumento de aspectos como a taxa de transferência ou resiliência.

Este trabalho apresenta uma aplicação para um controlador SDN que calcula caminhos entre nós numa rede usando diferentes algoritmos. Ela contém um módulo de monitorização de métricas de Qualidade de Serviço (QoS) que adiciona dados sobre a qualidade da ligação à topologia existente, utilizados para o cálculo de caminhos através da utilização do algoritmo SAMCRA. Quando vários caminhos estão disponíveis, diferentes estratégias podem ser usadas para escolher o próximo caminho. A aplicação é avaliada juntamente com a utilização do MPTCP, permitindo uma interação entre as camadas de transporte e de rede.

Os principais resultados mostraram que o uso de um algoritmo para cálculo de múltiplos caminhos com restrições melhora os valores das métricas de QoS medidas nas ligações estabelecidas, enquanto que a utilização de um algoritmo que produz um conjunto maior de soluções aumenta, em geral, a taxa de transferência dos fluxos.

**Palavras-chave:** Redes definidas por software, OpenFlow, MPTCP, Cálculo de caminhos, Qualidade de Serviço, CORE, emuladores de redes.





## *Abstract*

Software-Defined Networking (SDN) is a recent computer networking trend that allows a centralized management of network devices through a controller entity, responsible for processing incoming data, and assigning matching flows in the managed devices. When working in a SDN environment, it is possible to monitor the current state of a network and perform path calculation for new flows according to that state.

In addition, when different paths are available, and using a constrained multiple path computation algorithm, numerous paths can be calculated, following up constraints imposed by the metrics collected from the network. Combined that with the usage of multi-homed network transport protocols, such as Multipath TCP (MPTCP), the multiple created sub-flows can be assigned to different paths, contributing to the increase of aspects such as resilience or throughput.

This work presents a SDN controller application that calculates paths between network nodes by using different path computation algorithms. It features a Quality of Service (QoS) metrics monitoring module that adds link quality information to the existing topology data, used for calculating constrained QoS metrics paths through the usage of the Self-adaptive Multiple Constraints Routing Algorithm (SAMCRA) algorithm. When multiple paths are available, different flow pinning strategies can be used to select the next path. This application is evaluated along with the usage of MPTCP, allowing a cross-layer interaction between the transport and network layers.

The main obtained results showed that the usage of a constrained multiple path algorithm improves the QoS metrics values measured in the established connections, while the usage of an algorithm that produces a larger set of available paths increases the overall flows throughput.

**Keywords:** Software-Defined Networking, OpenFlow, MPTCP, Path computation, Quality of Service, CORE, network emulators.



# Contents

Acknowledgements	<a href="#">i</a>
Resumo	<a href="#">iii</a>
Abstract	<a href="#">v</a>
Contents	<a href="#">vii</a>
List of Figures	<a href="#">xi</a>
List of Tables	<a href="#">xv</a>
List of Abbreviations	<a href="#">xvii</a>
<b>1 Introduction</b>	<b><a href="#">1</a></b>
1.1 Motivation . . . . .	<a href="#">1</a>
1.2 Objectives . . . . .	<a href="#">2</a>
1.3 Contributions . . . . .	<a href="#">2</a>
1.4 Structure . . . . .	<a href="#">4</a>
<b>2 Software-Defined Networking</b>	<b><a href="#">5</a></b>
2.1 Background in SDN . . . . .	<a href="#">7</a>
2.2 SDN Controllers . . . . .	<a href="#">9</a>
2.3 Summary . . . . .	<a href="#">12</a>
<b>3 Transport Protocols</b>	<b><a href="#">15</a></b>
3.1 Single-homed transport protocols . . . . .	<a href="#">15</a>

---

3.1.1	Transmission Control Protocol . . . . .	16
3.1.2	User Datagram Protocol . . . . .	16
3.1.3	Datagram Congestion Control Protocol . . . . .	17
3.2	Multi-homed transport protocols . . . . .	18
3.2.1	Stream Control Transmission Protocol . . . . .	18
3.2.2	Multi-path Transmission Control Protocol . . . . .	20
3.3	Summary . . . . .	22
<b>4</b>	<b>Path computation algorithms</b>	<b>25</b>
4.1	Single-path algorithms . . . . .	26
4.1.1	Dijkstra's Algorithm . . . . .	26
4.1.2	A* Algorithm . . . . .	27
4.1.3	Bellman-Ford Algorithm . . . . .	27
4.1.4	Johnson's Algorithm . . . . .	28
4.1.5	Single-path algorithms with constraints . . . . .	29
4.2	Multi-path algorithms . . . . .	31
4.2.1	Link-disjoint algorithms . . . . .	33
4.2.2	Node-disjoint algorithms . . . . .	35
4.2.3	Multiple-path constrained algorithms . . . . .	37
4.3	Path computation in Software-Defined Networking . . . . .	40
4.4	Summary . . . . .	43
<b>5</b>	<b>Proposed Architecture</b>	<b>45</b>
5.1	Introduction and goals . . . . .	45
5.2	Used tools . . . . .	47
5.2.1	Opendaylight . . . . .	47
5.2.2	Open vSwitch . . . . .	47
5.2.3	CORE . . . . .	48
5.3	Evaluation platform . . . . .	50
5.3.1	Scenario Creator . . . . .	50
5.3.2	Experimenter framework . . . . .	51
5.4	SDN controller application specification . . . . .	52
5.4.1	Components description . . . . .	53

5.4.2	Application workflow . . . . .	63
5.5	SDN controller application implementation details . . . . .	64
5.5.1	Packet Handler . . . . .	64
5.5.2	Address Tracker . . . . .	65
5.5.3	TCP/UDP Packet Handler . . . . .	67
5.5.4	Packet Dispatcher . . . . .	67
5.5.5	Flow Writer . . . . .	68
5.5.6	Host Manager . . . . .	69
5.5.7	Network Graph Service . . . . .	69
5.5.8	Topology Change Handler . . . . .	70
5.5.9	Metrics Collector . . . . .	71
5.5.10	Path Calculator . . . . .	75
5.5.11	Flow Scheduler . . . . .	78
5.5.12	Application Main Module . . . . .	81
5.6	Summary . . . . .	82
<b>6</b>	<b>Evaluation results</b>	<b>83</b>
6.1	Transport protocol and flow scheduler evaluation . . . . .	83
6.1.1	Testing scenario . . . . .	84
6.1.2	Results . . . . .	87
6.2	Path computation algorithms evaluation . . . . .	92
6.2.1	Testing scenario . . . . .	93
6.2.2	Results . . . . .	97
6.3	Summary . . . . .	105
<b>7</b>	<b>Additional Contributions</b>	<b>107</b>
7.1	Opendaylight OVSDB REST client . . . . .	107
7.2	Multiflow . . . . .	108
7.2.1	Used architecture . . . . .	108
7.2.2	Obtained results . . . . .	110
7.3	Opendaylight workshop . . . . .	112
7.4	Summary . . . . .	113

<b>8 Project Management</b>	<b>115</b>
8.1 First Semester . . . . .	115
8.2 Second Semester . . . . .	118
<b>9 Final Considerations</b>	<b>121</b>
<b>References</b>	<b>125</b>
<b>Appendix A</b>	<b>141</b>
<b>Appendix B</b>	<b>151</b>
<b>Appendix C</b>	<b>155</b>

# List of Figures

2.1	Software-Defined Networking architecture diagram . . . . .	6
4.1	Two link-disjoint paths between host A and B . . . . .	34
4.2	Two node-disjoint paths between host A and B . . . . .	36
5.1	OVSDB Schema [Pfaff and Davie, 2013] . . . . .	48
5.2	Example of the usage of CORE's GUI . . . . .	49
5.3	Diagram of the experimenter framework . . . . .	53
5.4	Architecture of the SDN controller application . . . . .	54
5.5	Hash-based load-balancing . . . . .	60
5.6	Minimum-flows based load-balancing . . . . .	60
5.7	Static path load-balancing . . . . .	61
5.8	Random load-balancing . . . . .	61
5.9	Round-robin load-balancing . . . . .	62
5.10	Link delay monitoring . . . . .	73
5.11	Example of the content of a delay monitoring packet . . . . .	74
6.1	Topology used in the flow-based load balancing experiments . . .	84
6.2	CDF of the used file sizes in the flow schedulers evaluation experiments . . . . .	85
6.3	CDF of the used waiting times in the flow schedulers evaluation experiments . . . . .	86
6.4	All scheduler evaluation results, without SDN controller . . . . .	87
6.5	All scheduler evaluation results (with delay), without SDN controller	89

6.6	Round-robin and minimum flows evaluation results (TCP and MPTCP), using the SDN controller . . . . .	90
6.7	Hash-based and random, scheduler evaluation results (TCP and MPTCP), using the SDN controller . . . . .	91
6.8	Round-robin and minimum flows evaluation results with delay (TCP and MPTCP), using the SDN controller . . . . .	92
6.9	Hash-based and random, scheduler evaluation results with delay (TCP and MPTCP), using the SDN controller . . . . .	93
6.10	Topology used in the evaluation of path computation algorithms .	94
6.11	CDF of the used file sizes in the flow schedulers evaluation experiments . . . . .	96
6.12	Average mean transfer times, normalized with SAMCRA (Round-robin) . . . . .	98
6.13	CDF for RTT delay with 1 parallel transfer . . . . .	100
6.14	CDF for RTT delay with 8 parallel transfers . . . . .	101
6.15	CDF for RTT delay with 15 parallel transfers . . . . .	102
6.16	CDF for throughput with 1 parallel transfer . . . . .	103
6.17	CDF for throughput with 8 parallel transfers . . . . .	104
6.18	CDF for throughput with 15 parallel transfers . . . . .	105
7.1	Example of the GUI of the Opendaylight OVSDB REST client . .	108
7.2	Multiflow architecture . . . . .	109
7.3	Multiflow results when not using any premium clients . . . . .	111
7.4	Multiflow results when using one premium client . . . . .	112
7.5	Multiflow results when using two premium clients . . . . .	112
8.1	First semester work plan . . . . .	117
8.2	Second semester work plan . . . . .	120
A.1	Round-robin evaluation results, no SDN controller . . . . .	141
A.2	Hashed-based evaluation results, no SDN controller . . . . .	141
A.3	Random scheduler evaluation results, no SDN controller . . . . .	142
A.4	Minimum flow evaluation results, no SDN controller . . . . .	142
A.5	RR scheduler evaluation results (with delay), no SDN controller .	142



A.6 Hash-based scheduler evaluation results (with delay), no SDN controller . . . . .	142
A.7 Random scheduler evaluation results (with delay), no SDN controller	143
A.8 Min flows scheduler evaluation results (with delay), no SDN controller . . . . .	143
A.9 RR scheduler evaluation results with TCP, with SDN controller .	143
A.10 RR scheduler evaluation results with MPTCP, with SDN controller	143
A.11 Min flows scheduler evaluation results with TCP, with SDN controller	144
A.12 Min flows scheduler evaluation results with MPTCP, with SDN controller . . . . .	144
A.13 Hash-based scheduler evaluation results with TCP, with SDN controller . . . . .	144
A.14 Hash-based scheduler evaluation results with MPTCP, with SDN controller . . . . .	144
A.15 Random scheduler evaluation results with TCP, with SDN controller	145
A.16 Random scheduler evaluation results with MPTCP, with SDN controller . . . . .	145
A.17 RR scheduler evaluation results with delay and TCP, with SDN controller . . . . .	145
A.18 RR scheduler evaluation results with delay and MPTCP, with SDN controller . . . . .	145
A.19 Min flows scheduler evaluation results with delay and TCP, with SDN controller . . . . .	146
A.20 Min flows scheduler evaluation results with delay and MPTCP, with SDN controller . . . . .	146
A.21 Hash-based scheduler evaluation results with delay and TCP, with SDN controller . . . . .	146
A.22 Hash-based scheduler evaluation results with delay and MPTCP, with SDN controller . . . . .	146
A.23 Random scheduler evaluation results with delay and TCP, with SDN controller . . . . .	147
A.24 Random scheduler evaluation results with delay and MPTCP, with SDN controller . . . . .	147

B.1	Transfer times with Dijkstra's (Hash-based) . . . . .	151
B.2	Transfer times with Dijkstra's (Round-robin) . . . . .	151
B.3	Transfer times with Disjoint (Hash-based) . . . . .	152
B.4	Transfer times with Disjoint (Round-robin) . . . . .	152
B.5	Transfer times with Yen's (Hash-based) . . . . .	152
B.6	Transfer times with Yen's (Round-robin) . . . . .	152
B.7	Transfer times with SAMCRA (Hash-based) . . . . .	153
B.8	Transfer times with SAMCRA (Round-robin) . . . . .	153

# List of Tables

2.1	Existing SDN controllers [ <a href="#">Kreutz et al., 2014</a> ; <a href="#">Xia et al., 2014</a> ; <a href="#">Lara et al., 2013</a> ; <a href="#">Nunes et al., 2014</a> ] . . . . .	12
3.1	Overview of transport protocols . . . . .	23
4.1	Evaluated single-path computation algorithms . . . . .	32
4.2	Evaluated multiple-path computation algorithms . . . . .	39
6.1	Characteristics of the link in the path computation algorithms evaluation topology . . . . .	95
6.2	Normalized average MPTCP transfer times using Dijkstra's algorithm . . . . .	99
A.1	Average TCP transfer times with static flow allocation and no path delay . . . . .	147
A.2	Average TCP transfer times with static flow allocation and path delay . . . . .	148
A.3	Average TCP transfer times using the controller application and no path delay . . . . .	148
A.4	Average MPTCP transfer times using the controller application and no path delay . . . . .	149
A.5	Average TCP transfer times using the controller application with path delay . . . . .	149
A.6	Average MPTCP transfer times using the controller application with path delay . . . . .	150

B.1	Average MPTCP transfer times (ms) for different path computation algorithms . . . . .	153
B.2	CDF throughput values (Mbps) with 1 parallel transfer . . . . .	154
B.3	CDF throughput values (Mbps) with 8 parallel transfers . . . . .	154
B.4	CDF throughput values (Mbps) with 15 parallel transfers . . . . .	154

# List of Abbreviations

**ACK** Acknowledgement

**ACROSS** Autonomous Control for a Reliable Internet of Services

**API** Application Programming Interface

**ARP** Address Resolution Protocol

**BFD** Bidirectional Forwarding Detection

**BGP** Border Gateway Protocol

**CCID** Congestion Control ID

**CDF** Cumulative Distribution Function

**CFM** Connectivity Fault Management

**CLI** Command-line interface

**CMT** Concurrent Multipath Transfer

**CORE** Common Open Research Emulator

**COST** European Cooperation in Science and Technology

**CSP** Constrained Shortest Path

**CWND** Congestion window size

**D-ITG** Distributed Internet Traffic Generator

<b>DCCP</b>	Datagram Congestion Control Protocol
<b>DCLC</b>	Delay constrained least cost
<b>DHCP</b>	Dynamic Host Configuration Protocol
<b>DIMCRA</b>	Link-disjoint Multiple Constraints Routing Algorithm
<b>DoS</b>	Denial of Service
<b>EBF</b>	Extended Bellman-Ford Algorithm
<b>ECMP</b>	Equal-cost multi-path routing
<b>ForCES</b>	Forwarding and Control Element Separation
<b>FQ CoDel</b>	Fair Queueing with Controller Delay
<b>FTP</b>	File Transfer Protocol
<b>GRE</b>	Generic Routing Encapsulation
<b>GUI</b>	Graphical User Interface
<b>HCMB</b>	Hop Constrained Max Bandwidth
<b>ICMP</b>	Internet Control Message Protocol
<b>IP</b>	Internet Protocol
<b>IPv4</b>	Internet Protocol version 4
<b>IPv6</b>	Internet Protocol version 6
<b>JUNG</b>	Java Universal Network/Graph
<b>KMCSP</b>	$k$ multiple-constrained-shortest path
<b>KSP</b>	$k$ -shortest paths
<b>LAN</b>	Local Area Network
<b>LARAC</b>	Lagrange Relaxation based Aggregated Cost

---

<b>LLDP</b>	Link Layer Discovery Protocol
<b>MAC</b>	Media Access Control
<b>MADSWIP</b>	Maximally Disjoint Shortest and Widest Paths
<b>MANET</b>	Mobile Ad Hoc Network
<b>MCLPP</b>	Multiple Constrained Link-disjoint Path Problem
<b>MCP</b>	Multi-constrained path problem
<b>MD-SAL</b>	Model-Driven Service Abstraction Layer
<b>MPTCP</b>	Multi-path Transmission Control Protocol
<b>MuLaViTo</b>	Multi-Layer Visualization Tool
<b>Netconf</b>	Network configuration
<b>OF</b>	OpenFlow
<b>OS</b>	Operating System
<b>OSGi</b>	Open Services Gateway Initiative
<b>OSPF</b>	Open Shortest Path First
<b>OVS</b>	Open vSwitch
<b>OVSDb</b>	Open vSwitch Database
<b>OWAMP</b>	One-Way Active Measurement Protocol
<b>P2P</b>	Peer-to-peer
<b>POF</b>	Protocol-Oblivious Forwarding
<b>QoE</b>	Quality of Experience
<b>QoS</b>	Quality of Service
<b>REST</b>	Representational state transfer

**RIP** Routing Information Protocol

**ROIA** Real-Time Online Interactive Applications

**ROR** Resource Optimization-based with Customized Link-Disjoint Degree Routing

**RSTP** Rapid Spanning Tree Protocol

**RTT** Round-trip time

**SACK** Selective Acknowledgement

**SAL** Service Abstraction Layer

**SAMCRA** Self-adaptive Multiple Constraints Routing Algorithm

**SCTP** Stream Transmission Control Protocol

**SDN** Software-Defined Networking

**SFQ** Stochastic Fairness Queueing

**SNMP** Simple Network Management Protocol

**SP** Shortest Path

**SPB** Shortest Path Bridging

**SS** Slow-start

**STP** Spanning Tree Protocol

**STSM** Short-term scientific mission

**Swe-CTW** Swedish Communication Technologies Workshop

**SYN** Synchronize

**TAMCRA** Tunable Accuracy Multiple Constraints Routing Algorithm

**TCP** Transmission Control Protocol



**UDP** User Datagram Protocol

**VLAN** Virtual local access network

**VoIP** Voice over Internet Protocol

**VXLAN** Virtual Extensible LAN

**WSN** Wireless Sensor Networks



# Chapter 1

## Introduction

This document contains the description of the work that followed the development and deployment of a Software-Defined Networking (SDN) controller application capable of performing multiple path calculation using different computation strategies, along with its respective evaluation, based in the utilization of a multiple path transport protocol.

The motivation of this work is presented in the first section, followed by its objectives. The contributions that were made are described next, concluding with this document's structure.

### 1.1 Motivation

With the increase of the size of computer network topologies in data centers, traffic management becomes a more complex task, as each device needs to be configured individually and most of the used routing protocols cannot have a complete view of the network without introducing a complex distributed message mechanism. Additionally, the inclusion of data regarding the current network state in terms of delay, packet-loss and other Quality of Service (QoS) metrics when performing path computation is a difficult task due to the complexity of obtaining those values, but also to include them on the currently used algorithms.

The number of available network interfaces in the endpoint devices that are connected to this type of infrastructures also tends to increase. Examples can be

found in smartphones with a typical configuration of a 4G modem and a Wi-Fi transmitter, laptops with an Ethernet port and a Wi-Fi transmitter and desktop computers with more than one network card interface. However, there are not many solutions that provide a simultaneous usage of more than one interface when establishing and using a single connection, scenarios where the increase of throughput and resilience are beneficial.

## 1.2 Objectives

The work presented in this thesis has as main goal the development of a SDN controller application that provides multiple path computation algorithms that follows different strategies, including link-disjoint and QoS-metrics aware paths.

The benefits of using a multipath transport protocol will be studied, along with the usage of different flow pinning schedulers for defining in which path the created flows are installed. The evaluation will compare the different path computation algorithms, by the measurement of data from applications running in endpoint nodes from the managed networks.

The final obtained results will enhance the existing documentation regarding experimental work with SDN and the rest of the research topics addressed in this thesis.

## 1.3 Contributions

This thesis introduces Software-Defined Networks as a new computer networking paradigm and presents related work in some of its main trending research areas such as its performance, scalability and flexibility. The coverage of this topic includes an additional analysis of existing SDN controllers.

Regarding the network transport layer, it is identified a list of commonly used single-homed transport protocols (e.g. User Datagram Protocol and Transmission Control Protocol), as well as more recent multi-homed capable transport protocols, such as the Stream Control Transmission Protocol (SCTP) and the Multiple Path Transmission Control Protocol (MPTCP).

A comparative study of existing algorithms is presented, providing a background on path computation. This study is divided in 2 groups: single-path and multi-path computation algorithms. The existence of constrained metrics in these algorithms is also analysed.

The specification of the existing and to be developed required application modules is described, followed up by the testing framework built for deploying and evaluate the application, along with the used tools.

The author presents an evaluation of the implemented application, based in experimenting work in SDN networks where the used topology allows the existence of multiple paths between its nodes. Results were achieved by using the different implemented path computation strategies and flow pinning schedulers in the application. All the experimenting work was conducted in an emulated testbed environment using the Common Open Research Emulator (CORE). In order to deploy and use the related SDN tools, new extensions to this application were developed and added to the original installation.

A Short-Term Scientific Mission (STSM) for the European Cooperation in Science and Technology (COST) action IC1304 Autonomous Control for a Reliable Internet of Services (ACROSS) was submitted and accepted by the respective COST committee. This allowed the author to visit the University of Karlstad, in Sweden, where all the experimenting progress took place. Besides the work directly related to the topic of this thesis, different tasks were held during this period, which included the organization and presentation of a workshop in the development of SDN controller applications and the development of a client application that enabled the configuration of QoS mechanisms in Openflow switches. Additionally, while in Sweden, the author participated in different project meetings, where the progress of the ongoing work related to this thesis was presented.

A scientific poster entitled "Multipathing in Software-Defined Networking: Interaction between SDN and MPTCP" was accepted and presented in the Swedish Communication Technologies Workshop (Swe-CTW 2015) [Santos et al., 2015].

## 1.4 Structure

This document is structured as follows. Chapter 2 presents a description of the main concepts of Software-Defined Networking, with an overview of the existing SDN controllers. In chapter 3 the main used single and multiple-path based transport protocols are listed.

Chapter 4 follows up describing existing path computation algorithms that can be used in the context of flow routing in computer networks. Chapter 5 describes the architecture of the developed controller application, used testing environments, tools and by the details of its implementation. The evaluation of the application appears in chapter 6 and the additional contributions to this work in chapter 7.

The planning and the management of the project are exposed in chapter 8, based on the work performed during the first semester and on the tasks held during the second semester.

Finally, the project final considerations are presented in chapter 9.

## Chapter 2

# Software-Defined Networking

In traditionally used computer networking devices, such as routers and switches, the typical configuration resides in having the forwarding plane coupled with the control plane, or even if decoupled, both exist inside the same device and they need to be configured. The forwarding plane is responsible for sending incoming packets through its output ports based on existing rules, while the control plane handles the decision process of which action is taken for each packet, by setting up the forwarding rules used by the forwarding plane.

Software-Defined Networking (SDN) is a recent and trending computer networking architecture [Feamster et al., 2014] that allows the physical separation of the network control plane from the forwarding plane. This concept allows the control of several forwarding planes by a control plane managed by a single entity, allowing the abstraction of the existing network infrastructure from the applications perspective [Fundation, 2012]. This abstraction allows the user to manage the network from a global point of view, rather than individually from each device.

Within SDN, the control plane can be managed with a software application, entitled the controller. It communicates with physical and virtual switches through a common protocol, independently from their vendors.

The mostly used protocol in SDN is OpenFlow (OF) [McKeown et al., 2008], despite the existence of other known device configuration protocols, such as SNMP (Simple Network Management Protocol) [Case et al., 1989] or NETCONF (Network Configuration) [Enns et al., 2011]. This protocol and the interfaces used

for establishing the communication between the controller and the OpenFlow capable devices are part of the controller Southbound interface, while the Northbound interface allows external applications to communicate with the controller, through specific Application Programming Interfaces (APIs) [Kim and Feamster, 2013]. While the Southbound interface only differs on the used protocol and on its version compatibility in both devices and controller, the Northbound interface is independent and specific for each software controller. Additionally, the concept of East and Westbound interfaces were introduced, allowing the communication between controllers. An example of this communication can be found in *HyperFlow* [Tootoonchian and Ganjali, 2010], a distributed event-based control plane for OpenFlow, where several controllers can exchange data and share a global network view among each other. Figure 2.1 illustrates the existing interfaces and components of the SDN architecture.

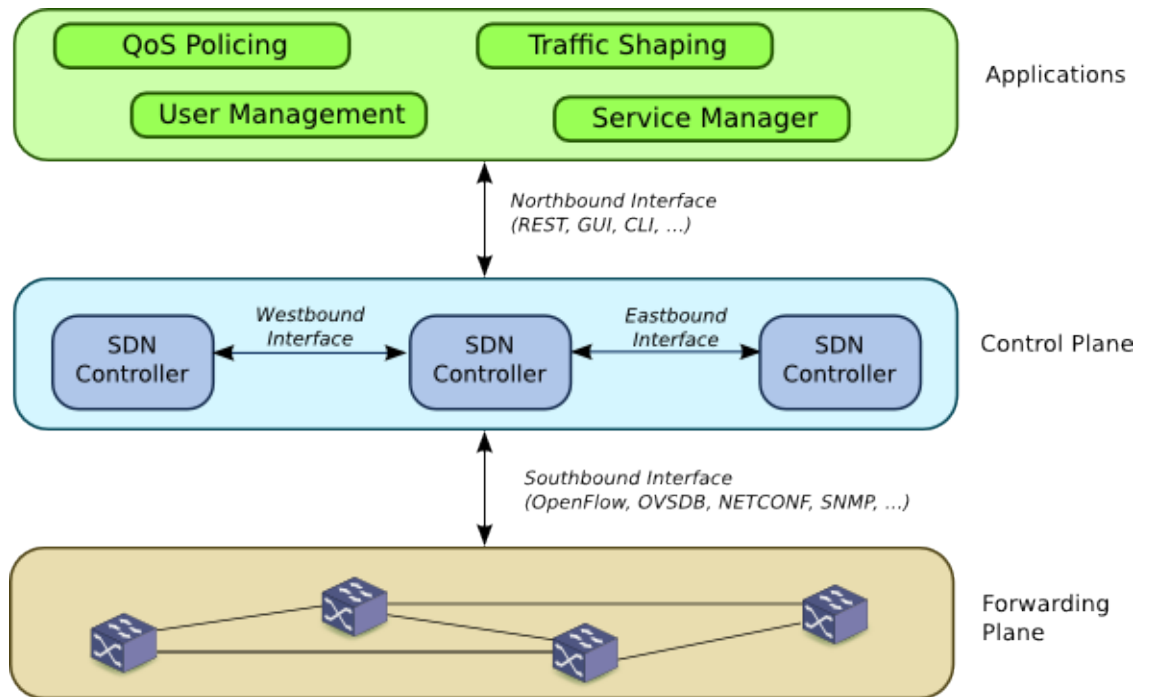


Figure 2.1: Software-Defined Networking architecture diagram



## 2.1 Background in SDN

Since it is an emerging computer networks trend, there is always the resistance to change from old network architectures perspectives to the usage and implementation of SDN capable technologies, having aspects as performance, flexibility, scalability, security and interoperability in some of the main key challenges to introduce further development in the industry [Sezer et al., 2013]. Besides those aspects, its implementation is already noticeable in big data center networks, like Google's [Jain et al., 2013].

While performance in SDN is directly related to the processing capabilities of the used devices (affecting latency and throughput), flexibility refers to being able to introduce new features in systems. Scalability is focused in the controller behaviour and its ability to communicate with other controllers and multiple network nodes (and corresponding latency).

Due to the future expansion of the usage of SDN, security became an important topic (an example can be a malicious user masquerading as a controller and perform attacks in network switches). FortNox [Porras et al., 2012] is a software extension to a SDN controller that implements role-based authorization in a SDN system. With role-based authorization, signed security OpenFlow applications are identified and the installation of conflicting flow rules by unknown OpenFlow applications can be detected and the corresponding flows can be rejected.

Integrating the implementation of SDN products in existing networks is directly related to the interoperability key challenge. The replacement of all the network devices for SDN-capable ones can be a complicated solution due to administrative and financial costs. On the other hand, the expansion of the complexity in the tasks performed by network operators is noticeable when developing and implementing SDN solutions in existing networks [Kanaumi et al., 2010], as there is increased work when configuring a SDN network and preserve the connectivity of the previous existing network devices at the same time. Therefore, putting together legacy and SDN devices requires specially driven solutions. An example of the usage of SDN with traditional network devices is QuagFlow [Nascimento et al., 2010], an application that integrates Quagga [Ishiguro et al., 2007], a popular software routing suite that combines implementations of the Bor-

der Gateway Protocol (BGP) [Rekhter and Li, 1995], Open Shortest Path First (OSPF) [Moy, 1989] and Routing Information Protocol (RIP) [Hedrick, 1988] routing algorithms, with an OpenFlow-compatible system. With QuagFlow it is possible to provide the routing service to legacy network devices through the usage of an OpenFlow controller application.

The current usage of Software-Defined Networking can be found in the literature through other different topics. An example can be found with the creation and implementation of Quality of Service (QoS) services that run on OpenFlow controllers or over SDN-capable networks. For video, voice, online gaming and other real-time application service providers, achieving QoS guarantees is a crucial goal. QoS can be offered by service providers by offering acceptable values for different network metrics, such as guaranteed bandwidth, end-to-end delay, jitter and packet loss rate [ITU-T, 2008]. When using SDN, the view of a global state of a network can be a great advantage when configuring the switches, as this network topology view can offer a significant input when calculating QoS-aware flows.

An extension to a SDN controller Northbound API that provides QoS management support [Humernbrum et al., 2014] presents additional support to Real-Time Online Interactive Applications (ROIA), allowing this type of applications to request and specify their QoS and network requirements. Given examples for ROIA presented by the authors are First-Person Shooter games, Real-Time Strategy games and e-learning applications.

By using a SDN controller's Northbound API, *MonSamp* [Raumer et al., 2014] is a monitoring application that collects QoS metrics from existing flows in SDN switches, taking advantage of OpenFlow capabilities. OpenSample [Suh et al., 2014] is another network monitoring platform built over an SDN controller that significantly lowers the sampling interval of port and flow statistic counters, by using sFlow [Phaal et al., 2001], a traffic monitoring tool supported on a vast number of switch models, alternatively to the legacy OpenFlow capabilities. Through the usage of an entropy-based method, another solution that uses sFlow can also be used to detect anomalies in a network [Giotis et al., 2014].

## 2.2 SDN Controllers

As the popularity of Software-Defined Networking raises, the number of available SDN controllers is also increasing. When choosing a SDN controller, several aspects must be considered and compared, such as the existing APIs (North, South, East and Westbound), base network services, available management interfaces, used architecture frameworks, used implementation programming language [Kreutz et al., 2014], its active development state, the size and reachability of the user and developer community and the supported OF versions. Besides SDN controllers, other topics of SDN can be found on previously made surveys, such as OpenFlow software and hardware switches, used emulation and simulation tools, and ongoing SDN-related research work [Xia et al., 2014; Lara et al., 2013; Nunes et al., 2014].

As it was previously mentioned, Southbound APIs are used to perform the communication between the control plane and forwarding plane of the existing devices. While OF is the main used protocol in this part of the SDN architecture, extensions and implementations to other known protocols can be found in SDN controllers. Examples of these protocols as alternatives to OpenFlow used to communicate and configure the device's forwarding plane through the control plane are OpFlex [Smith et al., 2014], POF (Protocol-oblivious forwarding) [Song, 2013] or ForCES (Forwarding and control element separation) [Yang et al., 2004].

Other protocols can be used as a complement to configure managed devices, such as OVSDB (Open vSwitch Database Protocol) [Pfaff and Davie, 2013], a protocol used to communicate with Open vSwitch (OVS) [Pfaff et al., 2009] and manage its configurable features. Open vSwitch is a software switch suite that besides its support for OF flow rules, it contains additional features (e.g. QoS configuration, Internet Protocol (IP) tunneling, connectivity fault management and VLAN support). SNMP and NETCONF are also used, mostly to configure and communicate with legacy network devices.

For the controllers that adopted OpenFlow as Southbound interface protocol, the used version is also a variable factor, as new fields and capabilities have been added through the past versions.

The Northbound APIs are used to establish the communication between the

SDN controller and existing applications. This communication can have different purposes, including fetching flow statistics, managing existing or adding new flows, connecting to new devices or editing administrative user permissions. The used protocol for this communication is open to the controller developers, as there is not a defined interface for this purpose, and even when the interface uses the same architecture, e.g. Representational State Transfer (REST), there is not an existing standardization of the provided functions and respective API nomenclature. The existing controller management interfaces' are directly related to the Northbound API, as it is common to manage and configure the controller through the provided northbound interface. Additionally, other common used mechanisms can be used (e.g. Command-line interface (CLI) or a web management page).

When it comes to the controller architecture, it can be distinguished between being distributed or centralized. The scalability issue seen in the previous section plays an important rule in this topic. While having a centralized controller makes the task of implementing a SDN network simpler, it can be the cause of a network's bottleneck, as it might not be able to process all the network requests [Yeganeh et al., 2013]. Hence, having a distributed solution among controllers can be a solution for more complex networks, as the workload can be distributed along different machines. On the other hand, by maintaining a typical SDN global network state among distributed controllers can introduce new trade-offs and additional issues, such as inconsistency when there is an increase of the application's logic complexity [Levin et al., 2012].

Currently there are several available controllers than can be installed in order to configure a SDN-based network, which can be distinguished and evaluated through the aspects previously listed in this section.

One of the first developed OF controllers is NOX [Nicira, 2008]. Written with C++ and with support to OF 1.0, later its development was followed by POX, a Python based controller. However, while these two applications had a active user and developer community until 2013, currently it is almost non-existent, when comparing to other current solutions. Additionally, an OF 1.3 capable version of NOX was also developed by CPqD [CPqD, 2012].

The same happened with Beacon [Erickson, 2013], a OF controller developed using Java, with a multithreaded architecture built over an Open Services Gate-

way Initiative (OSGi) framework, allowing the enhancement of its modularity and service-oriented capabilities. Despite the robustness of this controller it is also outdated, having its last version released in 2013. Similarly to Beacon, Maestro is a multithreaded Java-based OF controller developed by members of Rice University, with its last version released in 2011.

Ryu [[Ryu SDN Framework Community, 2012](#)] is a controller that supports OF 1.0, 1.2, 1.3 and 1.4 and offers a wide number of applications that run on top of it, including a switching hub, a link aggregation and a spanning tree application. In addition to its Northbound REST API methods used to perform flow management, it contains a basic set of methods for managing the QoS functionalities of Open vSwitch instances.

A C-based OF controller entitled OpenMul [[Kulcloud, 2012](#)] contains an architecture that allows applications to run and communicate with the controller through the same addressing space (due to its low-level programming language it is possible to take that aspect in consideration), contributing to the improve of its performance. Nevertheless, by using C and despite its performance benefits, the creation and modification of controller applications becomes a more complex task, in comparison to the other controllers that use high-level languages, such as Java or Python. It supports OF 1.0, 1.3 and 1.4.

The Floodlight controller [[Big Switch Networks, 2012](#)] is based on Java, and uses an OF Java library that supports OF 1.0 and 1.3 versions, with additional experimental support for OF 1.1, 1.2 and 1.4. Its most recent version was released by the end of December 2014 and includes built-in applications, such as an OpenStack integration plug-in, a virtual switch, a firewall and a circuit pusher, responsible for assigning flows to designated paths between hosts.

Opendaylight [[Medved et al., 2014](#)] is an OF controller managed by the Linux Foundation and backed by important computer networks-related companies, with Cisco, Brocade, HP, IBM and Intel being examples of those. It provides a structured layered architecture that provides several built-in functionalities, supported by their respective northbound and southbound APIs. It supports OF 1.0 and 1.3 and similarly to Beacon, it is structured by the OSGi framework, simplifying the management of the installed and active features. On the other hand, due to the same highly-assembled architecture, the development of new controller

Table 2.1: Existing SDN controllers [[Kreutz et al., 2014](#); [Xia et al., 2014](#); [Lara et al., 2013](#); [Nunes et al., 2014](#)]

Name	Platform	Developers	OpenFlow Versions
NOX	C++	Nicira	1.0
POX	Python	Nicira	1.0
Beacon	Java	Stanford	1.0
Floodlight	Java	Big Switch Networks	1.0, 1.1, 1.2, 1.3, 1.4
Maestro	Java	Rice University	1.0
Ryu	Python	Ryu SDN Community	1.0, 1.2, 1.3 and 1.4
nox13offib	C++	CPqD	1.3
OpenDayLight	Java	Linux Foundation	1.0 and 1.3
OpenMul	C	Kulcloud	1.0, 1.3 and 1.4

applications is more difficult due to the need of understanding all the required framework management proceedings. However, due to its community support, having mailing-lists, an Internet Relay Chat (IRC) channel, a dedicated Wiki page and a website for questions and answers, it is simple to get help by other users and developers.

Table 2.1 presents a list of the current existing SDN controllers and their supported OF versions, implementation programming language and the responsible company.

## 2.3 Summary

The definition and specification of SDN were described in this section. Along with its architecture that divides the control plane from the forwarding plane on compatible network devices, the SDN controller was also introduced.

The used protocols in SDN were also listed, being OpenFlow the most common among SDN-capable devices.

The different API that SDN controllers used were described, having the Northbound API for communicating with applications, the Southbound with forwarding plane devices and East and Westbound with other controllers. Besides these different APIs, existing SDN extensions regarding security, flexibility, perfor-

mance, scalability and interoperability were discussed.

A survey and analysis of the existing SDN controllers were presented, comparing the existing solutions. While there is a significant number of OF controllers, many are deprecated or did not have a new version for a considerable amount of time. From the analysed controllers, NOX, POX, Beacon, Maestro and nox13offlib can be included in that list.

Despite OpenMul being a computational efficient controller due to having C as its programming language, the expansion of its components gets more complex and prone to programming errors due to the low-level programming requirements needed, having other solutions based on higher-level programming languages preferable to use when implementing controller applications.

While Ryu, Floodlight and Opendaylight seem to be great candidates when choosing an OF controller, and despite the level of complexity required to develop controller applications due to its OGSi-based architecture, Opendaylight reveals being the most suitable controller to use at the moment, due to its community size, available resources (either documentation and supporting communication channels) and its development perspectives, with further releases already scheduled and a team composed by members of important companies.





# Chapter 3

## Transport Protocols

This section presents an overview of existing transport protocols, TCP (Transmission Control Protocol) [Cerf and Icahn, 2005], UDP (User Datagram Protocol) [Postel, 1980], DCCP (Datagram Congestion Control Protocol) [Kohler, 2006], SCTP (Stream Control Transmission Protocol) [R. Stewart, 2007] and MPTCP (Multi-path Transmission Control Protocol) [Ford et al., 2011], respectively. By the end of this section, Table 3.1 presents a comparative list of the features of these transport protocols.

### 3.1 Single-homed transport protocols

As transport protocols are used to successfully transmit and deliver data between applications, they use different mechanisms to distinguish and multiplex the sent traffic, such as labelling source and destination port numbers. Besides that type of configuration, transport protocols were created and are used according to the user and applications needs (e.g. data transfer, multimedia streaming, device signalling). The most typical and common configuration uses one source and destination IP address when communicating, as most protocols were designed for supporting only one network interface (single-homed). The next subsections will describe how some of this type of transport protocols behave.

### 3.1.1 Transmission Control Protocol

Transmission Control Protocol (TCP) is a transport protocol vastly used among the Internet and local networks, presented in 1974 and later defined [Postel, 1981]. It is connection-oriented and provides reliable data transfer between two endpoints. With TCP, data is transferred through streams while maintaining the order through which packets are sent. Reliability is reached through the exchange of Acknowledgement (ACK) messages upon packet receiving, and at the same time flow control occurs by adapting the transfer window size, specifying the maximum number of bytes that can be transferred to the receiver. Since it is also full-duplex, it allows simultaneous data transfers in both directions.

The congestion-control of TCP exists among different implementations. The first proposed algorithm was Tahoe TCP and its first modification was Reno TCP. These two introduced the usage of a congestion window that limits the number of sent packets until the previous ones are acknowledged Jacobson [1988]. The slow-start (SS), congestion avoidance and fast retransmit mechanisms were also included. Additionally, Reno TCP presents a the Fast Recovery state.

Vegas TCP [Brakmo and Peterson, 1995] is another variation of a TCP congestion control mechanism that monitors and uses the round-trip time (RTT) variation for setting up the congestion window, reacting to the packet delay, contrary to the previous mechanisms that are focused in the loss of packets.

Another implementation of the TCP congestion window algorithm is Westwood [Mascolo et al., 2001]. It uses bandwidth end-to-end calculations on the sender side by measuring the rate of incoming ACK packets. This rate is used when calculating the congestion window size after a congestion situation is detected (the sender receives three duplicated ACKs). This mechanism is particularly efficient when connected to wireless networks, due to the variable bandwidth caused by the variations on the received wireless signal.

### 3.1.2 User Datagram Protocol

Contradictory to TCP, User Datagram Protocol (UDP) is a transport protocol that is not connection-oriented, making it a stateless protocol. It allows data to be sent by packet datagrams, and as it does not provides reliability, it is not

possible to know if a sent packet was delivered to its destination. Due to its simplicity and lack of retransmission delays, it is suitable to be used in real-time applications, such as Voice over IP (VoIP), video streaming or online gaming. Additionally it allows multicast and broadcast communication.

Though it is used mostly for multimedia applications due to its unreliability, the lack of congestion-control mechanisms in the implementation of UDP can represent a problem when multiple flows are sharing the same link. A practical example can be a typical house Local Area Network (LAN) with a low bandwidth internet access, where several users are streaming videos. In this situation, either congestion-control is implemented on the used applications and the stream quality is adjusted in the server, or the network becomes easily congested, degrading the overall QoS for all the users. In order to present a solution to these kind of scenario, the Datagram Congestion Control Protocol (DCCP) was introduced.

### 3.1.3 Datagram Congestion Control Protocol

The Datagram Congestion Control Protocol is a recent (standardized in 2006) transport protocol. As its name indicates, and similarly to UDP, it communicates through the usage of datagrams. It provides unreliable congestion control, a feature not present in UDP, and consequently required to be implemented by the applications that used it as a transport protocol. The usage of a congestion-controlled unreliable communication is beneficial to multimedia streaming and communication applications, where the end-to-end delay needs to be taken in consideration in order to achieve quality in the transmitted sessions, while being aware of the existing link capacity in the used connections [Kohler et al., 2006].

DCCP was built considering mostly scenarios where traffic is transferred unidirectionally (e.g. a server streaming a video to a client). Based on that feature, and since it is connection-oriented, it supports half-connections (when only one endpoint sends data to the other). Since it also uses the exchange of ACK messages to aid in the congestion control mechanism (but in opposition to TCP, without retransmissions), a half-connection can be formed having a host sending data to another and receiving only the ACK packets.

The congestion support of DCCP allows the application to pick between dif-

ferent congestion control mechanisms, through the usage of a Congestion Control ID (CCID). This identifier can be configured for each existing half-connection.

Although DCCP is supported in the most recent Linux distributions as a module, its current usage is limited. Developers are required to define the protocol constant values (e.g. `IPPROTO_DCCP`, for identifying the protocol number), and need to use basic socket system calls with the defined options with programming languages that allow this customization, as DCCP specific libraries and wrappers are not publicly available for application development yet, keeping short the list of applications that use DCCP.

## 3.2 Multi-homed transport protocols

The increase of the number of network interfaces in existing devices enabled the specification, creation and utilization of transport protocols that take in consideration more than one interface (multi-homed), when establishing and using a network connection. The following subsections describe two of the most relevant multi-homed protocols, Stream Control Transmission Protocol (SCTP) and Multi-path Transmission Control Protocol (MPTCP) and their most important features and usage examples.

### 3.2.1 Stream Control Transmission Protocol

Stream Control Transmission Protocol is a transport protocol that offers multi-homing support, as opposite to the previous seen transport protocols, TCP, UDP. Multi-homing allows the usage of all the existing network interfaces in a device, contributing to the existence of multiple paths on a single connection (SCTP association) and therefore providing resilience support.

Two communication models can be distinguished when using SCTP, the primary backup Model and the Concurrent Multipath Transfer (CMT) [Iyengar et al., 2006]. In the first one, a primary path is used to perform data transfer. All the remaining available paths are backup paths that are used to replace the primary path upon link failure, detected through the increase of packet retransmission time-outs. With CMT, all the available paths are used to transfer data

on a SCTP association between two endpoints, contributing to the increasing of the used applications' throughput.

SCTP can handle multiple and independent streams per association, allowing data to be partitioned across multiple streams when transferring, hence benefiting from an independently sequenced delivery. This feature allows packets from other streams to continue to be sent when packet loss occurs in one stream. Although it supports streams, it organizes data transfer through datagrams.

In order to treat packet delivery, SCTP also offers a Selective ACK (SACK) mechanism, allowing the transmission of multiple Acknowledgement messages for different sent packets on a single message. Other key aspect of SCTP is the usage of a 4-way handshake when establishing an association, adding protection against SYN (Synchronize) flooding Denial of Service (DoS) attacks. For congestion control, SCTP uses the same algorithms as TCP. While its native implementation offers reliable communication, existing extensions allows it to implement full or partial unreliability [Stewart et al., 2004; Huang and Lin, 2013].

SCTP support is becoming a native feature in Linux distributions and there are available libraries for other operating systems (OS). However, similarly to DCCP, it is not an application transparent protocol, as it is required to invoke specific SCTP system functions in order to use it. This implies that existing applications cannot use SCTP, unless they were specially coded with SCTP support features, negatively affecting its widespread through users and developers.

Similar support issues are also found when using SCTP across different networks, as middle box devices (e.g. firewalls and network address translators), due to the lack of native SCTP support or the non-existence of specific SCTP-aware policies [Chen et al., 2013; Hayes et al., 2008].

When using the legacy path switching strategy with the primary backup model, SCTP lacks decision mechanisms for when there are faulty conditions in the network (e.g. transmission delay or in advanced scenarios, hardware resources saturation). Hence, further proposals and extensions to the original SCTP implementation were presented, with the main goal as keeping this selection aware of these variables [Sousa et al., 2013; Funasaka et al., 2005].

### 3.2.2 Multi-path Transmission Control Protocol

The Multi-path Transmission Control Protocol is a protocol based on the original implementation of TCP with multi-homing support. Similarly to SCTP, it allows the usage of multiple network interfaces simultaneously on a single connection, increasing the overall throughput. Each individual connection made through a MPTCP connection is called a subflow and is seen as a regular TCP connection with a byte-stream oriented data transfer. On the other hand, in opposition to SCTP, where only one connection can be established between each address for both endpoints, MPTCP allows the creation of cross-path subflows.

Concerning the usage of MPTCP, specific kernel modifications need to be performed, as the existing implementation is still under development, and are not included in standard Linux distributions. However, contrary to SCTP, MPTCP is transparent from the applications' point of view, as the standard TCP system calls are used in a MPTCP connection. Therefore, after enabling MPTCP in a system, it is not required to perform any modifications on the used applications in order to use it.

When starting a MPTCP connection, the endpoint sends a regular TCP SYN packet, as if it was starting a regular TCP connection. The difference resides in the packet options field, where it is added a `MP_CAPABLE` option, including a 64 bit key, used to identify the sender endpoint when establishing additional subflows. Upon receiving this packet, if the receiver endpoint also supports MPTCP, it sends a SYN/ACK packet with a `MP_CAPABLE` option and its 64 bit key, used for identification as well. If the receiving device does not support MPTCP, it will send a SYN/ACK packet without the `MP_CAPABLE` option, and the rest of the communication between the two devices will behave like a regular TCP connection. This makes MPTCP backwards compatible with the original TCP version.

After a MPTCP connection is established and the first subflow is created, additional subflows can be created using the remaining existing network addresses from the hosts. These subflows are formed through the process of initiating a normal TCP connection, but the SYN, SYN/ACK and ACK are sent containing a `MP_JOIN` option that includes the address to be added from the sending device, along additional options. When the other device received this message,

it will reply with a SYN/ACK packet containing also a MP\_JOIN option with its corresponding address.

Another possible situation can occur when a host wants to inform the other host about additional potential addresses to be used in a MPTCP connection, either because an address changed on one of the network interfaces, or a new connection is established (e.g. the device connected to a WiFi network). For these situations, the endpoint wanting to announce its address sends a TCP packet with an ADD\_ADDR option, containing the address to be announced.

Regarding congestion control, additionally to the inherited congestion mechanisms from TCP, MPTCP introduces a new multi-path aware congestion algorithm [Wischik et al., 2011]. This algorithm allows load-balancing along the used paths and respective subflows, contributing to improve the fairness when using bottleneck links. For reliability, MPTCP also inherits it from TCP, as well as its packet retransmission mechanisms, with an additional feature that allows data to be retransmitted through a different subflow instead of the original one.

Several work regarding the usage of multi-homed protocols can be found in the literature. An intercontinental experiment comparing the performance of CMT-SCTP with MPTCP was performed [Becke et al., 2013], achieving higher throughput with MPTCP due to its capability to create a full mesh of paths with all the available interfaces. The performance of MPTCP in modern data center network topologies was also examined [Raiciu et al., 2011], showing that the load balancing mechanisms of MPTCP provides an important advantage when dealing with existing bottlenecked traffic patterns in the network.

The combination of SDN and MPTCP is also a trending topic in computer networks research work. Multiple experiments were performed when testing the capabilities of MPTCP in SDN network environments [van der Pol et al., 2012, 2013], where an OpenFlow-based network was connected across different countries and the overall throughput behaviour was monitored when transferring data, observing that due to MPTCP's congestion control, it would stabilize using only two disjoint paths. MPTCP was also deployed in a SDN capable network [Chawanat et al., 2014], measuring the raise of the throughput, but in a smaller-sized network. Previous work also includes a set of tools that allow researchers to do experimenting work over a multi-path network through an OpenFlow configured

network [Németh et al., 2013].

A QoS-oriented mechanism built to increase the support and performance of multimedia applications was introduced as an extension for MPTCP that adds partial reliability [Diop et al., 2012]. This is achieved by implementing selective discarding when sending packets, a technique that removes packets with less importance when the network is saturated, such as B frames when streaming encoded video, decreasing the loss rate of I and P frames, the most important video frames, and consequently improving the resulting QoS. Another proposed solution is the introduction of time-constrained partial reliability, a technique that drops queued packets that exceed a waiting time limit. However, using these two introduced mechanisms it could not be achieved significant QoS improving, showing that further work in QoS-aware service providing needs to be enforced.

QoS and Quality-of-Experience (QoE) when using MPTCP is a subject also studied [Kongsen et al., 2012], where the variation of these parameters in disaster situations is explored, showing that MPTCP's resilience support and congestion control provides an advantage in these extreme conditions. Yet, the authors also pointed that there is still further work to be performed regarding multipath transport that would affect QoE/QoS metrics, using as examples packet scheduling and resource allocation.

MPTCP can also already be found in commercial products, as Apple implemented it in iOS7 and it is used to provide connectivity support to Siri, its virtual assistant software package. When the phone is connected to more than one networks (Wi-Fi and cellular data), it will use Wi-Fi as a primary TCP connection, and the second as a backup connection. This implementation of MPTCP is different from the previously described behaviour, since its main goal is the installation of resilient paths (instead of increasing the connection's throughput).

### 3.3 Summary

This section presented a description of relevant single and multi-homed transport protocols, which included TCP, UDP and DCCP as single-homed protocols and SCTP and MPTCP in the multi-homed protocols section. Finally a comparative table with its most important features was presented.



Table 3.1: Overview of transport protocols

Feature	TCP	UDP	DCCP	SCTP	MPTCP
Connection-oriented	Yes	No	Yes	Yes	Yes
Transfer mode	Byte-Stream	Datagram	Datagram	Datagram	Byte-Stream
Stateful	Yes	No	Yes	Yes	Yes
Reliability	Yes	No	No	Yes	Yes
Congestion-control	Yes	No	Yes	Yes	Yes
Multi-homing	No	No	No	Yes	Yes
Native OS integration	Yes	Yes	Module	No	No
Application compatibility	Yes	Yes	No	Specific API	Yes

The usage of a multi-homed capable transport protocol takes advantage of the existence of multiple connected network interfaces, aiding in the improvement of resilience and throughput. Yet, it is difficult to change existing network infrastructures and applications in order to support new protocols. While SCTP can offer multiple backup paths or use them at the same time, it requires specific system calls to establish and maintain an association, as well a distinct protocol number on IP packets. These two disadvantages make MPTCP a more suitable option, since it is transparent to applications, as they keep using the regular TCP system calls, and each MPTCP sub-flow is a TCP connection, having just additional header options on each packet, contributing to the increase of compatibility with middleboxes.

While the increase of throughput over a multi-homed connection might be beneficial to data transfers, e.g. File Transfer Protocol (FTP) traffic, the usage of different network paths might cause problems with delay-sensitive traffic, as the presence of heterogeneity on the observed QoS metrics values for different paths can degrade the perceived QoE. Hence, single-path transfers remains the most trustful choice for unreliable multimedia traffic.

Although DCCP, being an unreliable datagram transport protocol, provides congestion control that can be benign to multimedia applications, similarly to SCTP it requires specific application system calls and its traffic can be blocked in middleboxes since it is a recent protocol with a different IP protocol number identifier (i.e. comparing to the vastly used TCP and UDP). Despite UDP being a simple transport protocol it is massively used for transferring unreliable traffic.

Though it does not implement any congestion control or other different optimization features, the combination of its usage with efficient QoS routing policies can contribute to the improvement of the perceived user experience.

# Chapter 4

## Path computation algorithms

In everyday science, path computation is a relevant field of study. One of the most common applications is the calculation of an existing shortest path (SP), which can involve the calculation of a SP either between two points of interest in a city, between two vertexes in a graph, or between two nodes inside a network.

When performing path calculation, some parameters can be used to identify and distinguish the used methods and algorithms. These can be the weight of each link (the length of a road, for example), the existence of negative weighted paths, having single or multiple constraints applied to each link, or the number of considered paths (single or multiple path-aware algorithms).

Regarding the performance of algorithms, there are common metrics that are used when evaluating one. These can include its temporal and spatial complexity. The temporal complexity is used to quantify the running time of an algorithm, while spatial complexity indicates the amount of data that is needed to store in memory while running an algorithm. In order to measure these two metrics, it is often used the *Big-O* notation (e.g.  $O(n^2)$ , with  $n$  as an argument of the algorithm input), a method used to asymptotically measure complexity [Sipser, 2006].

This section will present some of the most significant solutions for path computation, either through the calculation of single or multiple paths.

## 4.1 Single-path algorithms

There is a significant number of existing algorithms that perform single-path path calculation. However, each of the known and most used solutions are more suitable when the used input matches a given criteria. The following subsections will explore some of the most used single-path algorithms and their variations.

### 4.1.1 Dijkstra's Algorithm

Dijkstra's Algorithm [Dijkstra, 1959] was designed to solve efficiently problems where it was required to find the SP between a given source node and the remaining nodes in a graph with non-negative weights. The algorithm works by iteratively traversing the existing nodes, starting from the source node and finishing when it reaches the destination node. When it passes through a node it calculates the distance from the starting point until where it is, and if its smaller than the previously stored value (in the beginning all the distances are set to  $+\infty$ ), it saves the new distance.

Its original implementation examines all the existing nodes paths, running in  $O(n^2)$  time [Cherkassky et al., 1996], having  $n$  as the number of nodes in the graph. However, additional versions were implemented with better overall performances. One example of those implementations uses Fibonnaci heaps [Fredman and Tarjan, 1987], maintaining additional queue data structures for keeping the cumulative lengths sorted between each node and runs in  $O(m + n \log n)$ , faster than the original version, with  $m$  being the number of edges/links.

A restriction in this algorithm, as previously said, is that it won't compute when having negative weights on its input, as it behaves like a greedy algorithm, presuming that for each iteration it is found the best solution. The appearance of negative weight links allows a previous calculated choice (meant to be the best), stop being the optimal one, making the algorithm fail to compute the SP in these situations. Other solutions have been developed in order to solve the shortest-path problem when having negative weights, as seen in 4.1.3 and 4.1.4.

In path computation in computer networks the most known application of Dijkstra's algorithm is included in the OSPF routing protocol.

### 4.1.2 A\* Algorithm

The A\* Algorithm [Hart et al., 1968] calculates the SP between a source and a destination node with the usage of a heuristic function,  $h(n)$ . This function computes an estimation of the distance of the node that is being visited to the destination goal. Through each iteration, the algorithm calculates the travelled distance from the source node to the current visited node, known as  $g(n)$ , and adds the value of the calculated heuristic, forming the distance function  $f(n) = g(n) + h(n)$ . It then picks the shortest calculated  $f(n)$  value for all the surrounded unvisited nodes, iteratively repeating the same steps until the destination node is reached.

It is mostly used for the path computation among artificial intelligence in video games, as it is possible to achieve admissible heuristic functions when performing path calculation with grid-based input representations, such as the Euclidean distance (the absolute distance between the two considered points). If the A\* algorithm uses a non-admissible heuristic (if it overestimates the distance to the end node), it may produce non-optimal solutions. Similarly to Dijkstra's Algorithm (as it behaves identically if it uses a heuristic function  $h(n) = 0$ ), the A\* algorithm does not work correctly when there are negative edge weights. Its complexity depends on the heuristic function; in a worst case scenario its complexity runs in  $O(b^d)$ , having  $b$  as the branching factor (average number of nodes to expand per iteration) and  $d$  as the length of the existing SP, as it travels all the nodes exponentially. However, the big drawback in the usage of this algorithm is its spatial complexity, as it keeps in memory all the data used to compute the node distances while it is running [Russell et al., 1995].

Nevertheless, when calculating the SP in computer networks, where its representation is graph based and the prediction of the remaining distance to a node is hard to calculate, it becomes a difficult task to maintain a good heuristic function, making Dijkstra's algorithm a preferable solution against the A\* algorithm.

### 4.1.3 Bellman-Ford Algorithm

A possible solution for solving the shortest-path problem between a source node and all the existing ones in a graph in situations where there are negative weight

paths is the usage of the Bellman-Ford algorithm [Bellman, 1956]. Its functioning is similar to Dijkstra's algorithm, as it updates the distance between the source node and ones that it crosses for each iteration.

The main difference between the Bellman-Ford and Dijkstra's algorithms is that while Dijkstra's works by iteratively visiting the node with the shortest distance to the previously visited node, the Bellman-Ford algorithm visits every node's edge and calculates the respective distance, crossing all the existing input nodes with a complexity of  $O(nm)$ .

Since it visits every node and computes all the respective distances, it is possible to apply this algorithm in inputs with negative weights, contrary to Dijkstra's algorithm that uses a greedy approach, crossing only once each node and failing to present a correct result when there are paths with negative weights that can present a solution with a shorter distance. Due to the possibility of existing negative cycles, by the end of the execution of the algorithm, it cycles through all the edges again, and if it detects any changes in the stored values it detects the existence of a negative cycle.

Similarly to Dijkstra's, The Bellman-Ford algorithm is also applied in routing protocols, being the used algorithm in RIP.

#### 4.1.4 Johnson's Algorithm

While the explored algorithms in the previous sections were designed to solve the SP problem from a single source through a destination node ( $A^*$ ) or through all the remaining nodes (Dijkstra's and Bellman-Ford), sometimes it is required to solve this problem for all the source nodes, in order to get the SP between all the nodes in the input graph. A possible resolution can be Johnson's algorithm [Johnson, 1977], an algorithm built to solve the all-pairs SP problem.

This algorithm combines the usage of Bellman-Ford and Dijkstra's algorithms and runs in  $O(n^2 \log n + nm \log n)$  (having  $n$  as the number of nodes and  $m$  the number of edges), calculating the distance between all the existing nodes, iteratively.

It starts by calculating the SP from a single node to all the others using the Bellman-Ford algorithm. After that distance is calculated, it updates the weight

of all the edges by using the calculated distance between the previously source node and the nodes that form the edge to be updated. This update allows the replacement of all the distances with non-negative equivalent ones. Following this first iteration, it is applied Dijkstra's algorithm for all the remaining nodes (since there are not any remaining negative distances).

The algorithm ends its execution when the distance between all the nodes is calculated or when it is found a negative cycle while running the Bellman-Ford algorithm.

#### 4.1.5 Single-path algorithms with constraints

The typical SP problem solution involves the calculation of a path between one or more nodes through by minimizing the sum of the existing weights in the travelled nodes, from the source to the destination nodes. Mathematically this represents an optimization problem where the objective function is to minimize one variable. However, there are situations where this method of path calculation does not comply with the requirements of specific applications, e.g. QoS-aware applications and services, where multiple metrics and constraints need to fulfil the required demands. This problem is often referred as the *Constrained Shortest Path* (CSP) or *Multi-constrained path problem* (MCP), when the number of constraints is bigger than one. When the problem is to minimize the cost of a path following a delay constraint it is called *Delay constrained least cost* (DCLC) path problem.

Those multiple metrics must be calculated considering the measured values in all the links that form a path between the source and destination node. Since all these metrics have a different impact on the quality of a link (for example, it might be preferable to have a link with low end-to-end delay but high bandwidth availability), there are known three types of metrics that can be distinguished [Wang and Crowcroft, 1996]:

- **Additive metrics:** When the measured value of a path is given by the sum of the same metric values for each link;
- **Multiplicative metrics:** When the measured value of a path is given by the multiplication of the same metric values for each link;

- **Concave metrics:** When the measured value of a path is given by the minimum achieved value from all the links.

Delay, jitter, link cost and hop count are additive metrics, as they need to be minimized in order to improve QoS. Bandwidth is a concave metric (the link with the lowest bandwidth affects the remaining link in the path) and packet loss is included in the multiplicative metrics. Additive and multiplicative constrained metrics can also be identified as path constraints, while concave constrained metrics can also be referred as link constraints [Lee et al., 1995].

When solving SP problems with multiple constraints, the complexity of the existing problem also needs to be considered. Algorithms with constraints on two or more metrics are said to NP-complete [Garey et al., 1980; Chen and Nahrstedt, 1998], making huge input scenarios almost impossible to solve in real-time, a feature required when dealing with reactive flow path calculations.

Another concern about running algorithms that involve the input from different metrics among several network devices is the inaccuracy of the collected data. The sources of this inaccuracy can be related to not being possible to collect the metrics timely, or due to the overhead generated by the propagation of advertisements regarding state changes in the network nodes [Guérin and Orda, 1999]. For these situations, the usage of Software-Defined Networking solutions can be beneficial, due to the existence of a centralized controller with a global view of the network that can play an important role when fetching metrics data across the network.

A common approach to build a solution for the MCP is through the modification of existing path calculation algorithms, adding the remaining constraint requirement verifications. However, due to its complexity, additional adaptations that reduce the algorithm's complexity need to be implemented, in order to solve this problem in polynomial time.

One example of a presented solution for solving the MCP in polynomial time, when having two constrained additive metrics, is through the creation of a new weight function based on one of the constrained variables [Chen and Nahrstedt, 1998], simplifying the original problem by replacing one constraint and respective weight values with equivalent finite integer ones (being the constraint an



adjustable part of the algorithm, independently of the used input values). With this method, when using a modified Dijkstra's algorithm that performs the double constraint verification, the MCP was proved to be solved with a  $O(x^2n^2)$  time complexity, with  $x$  being a integer bound to the algorithm implementation. The equivalent solution using an extended Bellman-Ford algorithm (EBF) has a time complexity of  $O(xnm)$ . Both implementations with these two algorithms have a polynomial time complexity, simplifying the original NP-Complete problem. This integer bounding approach is also used in other similar related work with the Bellman-Ford algorithm, and is named *Limited granularity heuristic* [Yuan, 1999].

The Lagrangian Relaxation Method [Fisher, 2004] is used in LARAC (Lagrange Relaxation based Aggregated Cost) [Juttner et al., 2001], an algorithm built to solve the DCLC problem in polynomial time. Through the usage of a multiplier variable ( $\lambda$ ), the original cost function is modified in order to include the path delay in the equation to be solved. This way, by varying  $\lambda$ , it can found paths that respect the delay constraint bounds and have the cost value minimized, taking the path's delay in consideration. The example of an optimal solution can be achieved when  $\lambda = 0$  and the sum of the path delay respects the constraint, since the cost function is not affected by the path delay. This algorithm runs in  $O(m^2 \log^4 m)$  [Upadhyaya and Dhingra, 2010].

Additionally, more extensive and comprehensive lists of the existing path computation problems and solution proposals can be found in existing surveys [Chen and Nahrstedt, 1998; Kuipers et al., 2002; Upadhyaya and Dhingra, 2010].

A summary of the evaluated single path computation algorithms can be seen in table 4.1.

## 4.2 Multi-path algorithms

When computing different paths in computer networks, there are situations when it is beneficial to have more than one path calculated from a source to a destination node. These can include the increase of resilience by having backup paths available to provide a fail-over mechanism in a faulty network, or increasing the throughput of the network connection, when using a multi-path protocol (such

Table 4.1: Evaluated single-path computation algorithms

Name	Problem	Constraints	Complexity	Observations
[Dijkstra, 1959]	SP	No	$O(m+n \log n)$	Does not work with negative weight paths
[Hart et al., 1968]	SP	No	$O(b^d)$	Uses a heuristic function to calculate shortest paths
[Bellman, 1956]	SP	No	$O(nm)$	Can compute with negative weight paths
[Johnson, 1977]	SP	No	$O(n^2 \log n + nm \log n)$	Uses Dijkstra's and the Bellman-Ford algorithm
[Juttner et al., 2001]	DCLC	Delay and Cost	$O(m^2 \log^4 m)$	Computes using the Lagrange Relaxation method
[Chen and Nahrstedt, 1998]	MCP	Any 2	$O(xnm)$	Simplifies the constraints using a limited granularity heuristic

as MPTCP), that can create multiple sub-flows for each connection.

Existing routing protocols (e.g. OSPF) already offer multi-path routing support. Equal-cost multi-path routing (ECMP) can be used in routing devices, allowing the existence of multiple possible paths for a single destination address. Each path is evaluated according to routing (QoS) metrics and if multiple paths are tied with the same results, the routing device can use its defined scheduling methods for picking one. Additionally, in order to replace traditional loop preventing protocols, such as Spanning Tree Protocol (STP) or Rapid Spanning Tree Protocol (RSTP) [IEEE Standards Association, 2004], a new protocol specified in the *IEEE 802.1aq* standard entitled Shortest Path Bridging (SPB) [Allan et al., 2010] was created, allowing more than one path to be active in a network.

The problem for finding multiple shortest paths is often known as the  $k$ -shortest paths (KSP) problem. Several characteristics can be distinguished between multiple path computation algorithms, that can be more suitable to specific situations. One of them is the existence of loops in existing paths, something that frequently exists in complex computer networks topologies, but cannot be included in the calculated paths. This variation of the KSP problem is called the  $k$ -shortest simple (loopless) paths problem. The existence of paths with negative weights should also not be considered in computer networks graph representations.

Yen's algorithm [Yen, 1971] is an approach for obtaining the  $k$ -shortest simple paths from a source to a destination can be made through the calculation of de-

viations from the previous calculated paths. This can be achieved by temporarily redefining the distance between some of the edges contained in the last shortest paths to  $+\infty$ , then re-applying a shortest-path algorithm to the modified network graph. When using Dijkstra's algorithm for calculating the shortest paths (see 4.1.1), the algorithm runs in  $O(kn(m+n \log n))$ , as it computes  $kn$  shortest paths while running. An improvement for this algorithm was also presented, reducing its complexity through discarding the paths that would be calculated more than once [Lawler, 1972].

Other technique for computing the KSP is by using Eppstein's algorithm [Eppstein, 1998], which includes computing an auxiliary tree that represents the sidetrack distance difference of each incoming node's edge to the shortest-path found from a source to that node. This tree can be calculated from an original shortest-path tree (generated by a single-shortest-path algorithm, e.g. Dijkstra's) and allows to sort the remaining shortest paths by their length, providing a subset of shortest paths between a pair of nodes. While this algorithm runs in  $O(m + n \log n + k)$ , it allows the existence of loops, making it not suitable for computer networks path calculation.

Furthermore, other solutions for the KSP can be found in a previously done comparative study [Brander and Sinclair, 1996].

The avoidance of network bottlenecks is another factor that motivates the calculation of different paths. There are two specific scenarios where these kind of algorithms can differ from its classification, in order to attempt to remove this type of obstacles, respectively by being link-disjointed or node-disjointed solutions. The following two sections will describe how each of these two operate, additionally identifying existent work.

### 4.2.1 Link-disjoint algorithms

The existence of routing solutions through the usage of link-disjoint paths can offer alternative paths to redirect existing traffic upon link failure, or providing different paths for multiple traffic flows. This approach can aid in avoiding reaching a congestion state, often perceived when sharing several flows through the same path. A simple example can occur when two clients are connected to

the Internet through a domestic network and one of them is downloading a huge amount of data through a peer-to-peer (P2P) client application. In this scenario the other user often perceives a congested connection if the used network does not have enough bandwidth for supporting both clients at the same time. Figure 4.1 illustrates an example of two disjoint paths between two nodes.

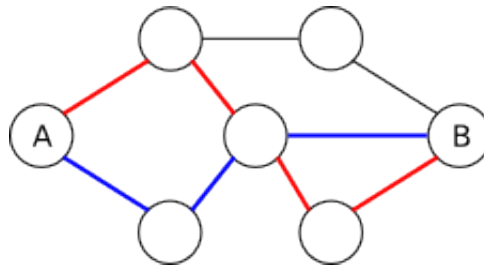


Figure 4.1: Two link-disjoint paths between host A and B

Between link-disjoint path calculation problems, there are different versions concerning the source and destination nodes of the paths:

- Finding  $k$  disjoint paths between  $k$  pairs of vertices [Shiloach and Perl, 1978];
- Finding 2 disjoint paths between a source node and every possible destination node [Suurballe and Tarjan, 1984];
- Finding  $k$  disjoint paths between a source and destination node [Li et al., 1992].

The first version can be used to solve printed circuits design or routing problems in computer networks, whether there are  $k$  pairs of components that need to be independently connected or  $k$  pairs of nodes that are required to be connected without sharing the used links for providing load-balancing fairness.

Finding a pair of disjoint paths between a source node and all the remaining nodes can be applied when building fail-over solutions in a network, e.g. a server needs to build a primary and a backup path between each other node in a mobile ad hoc network (MANET) in order to be able to provide reliability when transferring data.

As previously said when describing MPTCP, different sub-flows are created when establishing a connection between two hosts. A common configuration is to create a sub-flow for each existing available network interface, so when 2 hosts have  $k$  connected network interfaces,  $k$  MPTCP sub-flows are created. For this scenario it is helpful to have  $k$  link-disjoint paths (one for each sub-flow), in order to maximize the obtained connection throughput. However, due to the complexity of this problem, it is difficult to find algorithms that run in efficient time and are suitable to implement in computer networks path computation. For example, the solution proposed by Li et al. [Li et al., 1992] calculates  $k$  disjoint paths, but it only works with acyclic and directed input graphs.

An alternative to the computation of fully link-disjoint paths is the calculation of partially disjoint paths through the introduction of a link-disjoint degree, corresponding to the percentage of link-disjoint paths between an existing path and a second one, when it is not possible to find link-disjoint paths and at the same time fill existing network requirements. This approach is used with the Resource Optimization-based with Customized Link-Disjoint Degree Routing (ROR) algorithm [Chap et al., 2011].

### 4.2.2 Node-disjoint algorithms

In the node-disjoint path problem, visited nodes are not repeated among the calculated paths (besides the source and destination ones). This problem is also called the vertex-disjoint multiple path problem and it is an example of it is represented in Figure 4.2. The calculation of node-disjoint paths is, identically to link-disjoint path computation algorithms, used mainly for reliability and load-balancing in computer networks. An important example of its applications can be with Wireless Sensor Networks (WSN), where the power consumption of the used devices needs to be minimized. The usage of an efficient load-balancing mechanism through the usage of node-disjoint paths is a solution in aid to conserve energy resources [Cardei and Wu, 2006].

As all the other path computation algorithms, several variants of this problem can be formed, such as the existence of negative weights or the edges being directed or undirected. Since a node cannot be visited more than once, there are

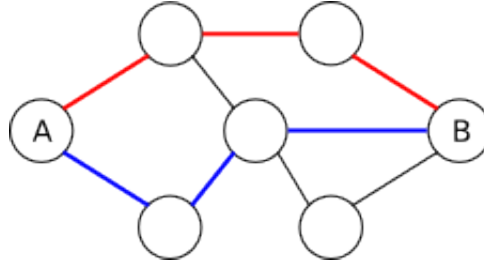


Figure 4.2: Two node-disjoint paths between host A and B

no loops in the generated solutions for this problem, making all the node-disjoint paths simple.

Many distributed algorithms were proposed for finding multiple node-disjoint paths through a network, through a mechanism of message exchanging between nodes for finding paths among them [Sidhu et al., 1991; Lee and Gerla, 2001]. Likewise distributed routing algorithms, with SDN this type of approach becomes obsoleted, since the SDN controller has full knowledge of the network topology, sparing the overhead of the generated messages and simplifying the algorithm's behaviour.

In order to solve this problem through existing generalized algorithms, a node-disjoint problem with directed edges can be easily turned into a link-disjoint problem, by creating an adjacent auxiliary node for each existing node, connected by a weightless edge and with all the outbound arcs of the original node linked to this second. An undirected graph can also be turned into a directed one by turning each undirected edge to two edges with opposite directions [Suurballe, 1974]. By making these modifications, one can turn a node-disjoint into a link-disjoint problem and apply one of the solutions previously proposed.

A solution presented for calculating either node or link-disjoint paths by Meghanathan [Meghanathan, 2007] implements an algorithm that runs Dijkstra's algorithm iteratively multiple times for finding different paths. Yet, after running Dijkstra's algorithm during each iteration, the algorithm firstly removes either every link or node (if calculating link or node-disjoint paths respectively) that were in the previous calculated path from the used graph. Given the fact that the first shortest-path found might contain edges or nodes that can be used to build different disjoint paths, by preliminarily removing these edges/nodes the algorithm

does not allow finding other paths, even when they were present in the original graph, making it not suitable for situations when more paths are required to be found.

### 4.2.3 Multiple-path constrained algorithms

Similarly to the single path computation algorithms, we can also impose constraints to be fulfilled or parameters to be minimized. Again, this type of problem ends up having a huge complexity, aggravated by the increased number of calculations to perform, a factor directly proportional to the number of paths that are required to calculate, in most of the cases.

The A\* Prune algorithm [Liu and Ramakrishnan, 2001] solves the KMCSP ( $k$  multiple-constrained-shortest path) problem by providing an adaptation to the original A\* algorithm (see 4.1.2). It builds paths from a source to a destination node, expanding the visited nodes and discarding the paths that violate the imposed constraints. Like the A\* algorithm, this algorithm uses a heuristic function for projecting an assumption of the cost of a path, based on all the constrained variables. This heuristic calculates admissible paths by using an additional algorithm for computing the lower bounds of each constrained weight (examples the authors referred are Dijkstra's algorithm or the Lagrangian relaxation method). It runs in  $O(khd^2(h + r + \log(khd)))$ , having  $k$  as the number of paths to find,  $d$  the degree of the input graph (maximum number of edges connected to a node in the graph),  $r$  the number of constrained metrics and  $h$  as the maximum hops of the computed  $k$  paths.

A *Limited Path Heuristic* [Yuan, 1999] can be used along with an EBF implementation that originally computes all the optimal paths between the source node and all the remaining nodes [Widyono et al., 1994]. In the original algorithm extension, the number of existing optimal paths can grow exponentially, depending on the number of nodes and links in the input data. Hence, with the *Limited Path Heuristic* this number of paths is limited by a variable ( $x$ ), reducing the time complexity of the algorithm to  $O(x^2nm)$ , making it solvable in polynomial time. Nevertheless, when using this heuristic the best optimal path can not always be found, since at least  $x$  optimal paths can be discovered before finding

the best optimal one.

Li et al. [Li et al., 1990, 1992] present algorithms for finding both node and link-disjoint paths with multiple restrictions. However, they are restricted to acyclic networks, a restriction hard to find in complex network topologies, making the algorithms not suitable for being used in those scenarios.

The self-adaptive multiple constraints routing algorithm (SAMCRA) [Van Mieghem et al., 2001] and the tunable accuracy multiple constraints routing algorithm (TAMCRA) [De Neve and Van Mieghem, 2000] are two algorithms that use a non-linear representation of a path length by combining all the path weights and constraints, obtaining the final path length through the calculation of the maximum value of the quotient of the division between the path weight and respective constant, for each metric. Both algorithms allow the calculation of  $k$ -shortest paths, with  $k$  being specified on the input of the algorithm in TAMCRA and optimally found with SAMCRA. TAMCRA runs in  $O(kn \log(kn) + k^3 Cm)$  and SAMCRA in  $O(kn \log(kn) + k^2 Cm)$ , having  $n$  as the number of nodes,  $k$  the number of calculated paths,  $m$  the number of edges and  $C$  the number of constraints.

DIMCRA (link-disjoint multiple constraints routing algorithm) [Guo et al., 2003] is an adapted version of SAMCRA that computes disjoint paths with multiple constraints, but it only provides solutions for the multiple constrained link-disjoint pair problem (MCLPP), therefore only producing a maximum number of 2 disjoint paths. As seen before, a pair of disjoint paths can be used as a fail over or load balancing solution in a computer network. Nonetheless, in order to maximize the throughput of each subflow created in a multipath transport protocol, more than 2 paths can be required, if the number of created subflows is also greater than 2, which makes this algorithm unsuitable for generic situations where multipath protocols create multiple subflows per connection. Similarly to DIMCRA, the Maximally Disjoint Shortest and Widest Paths (MADSWIP) algorithm [Taft-Plotkin et al., 1999] calculates a pair of maximally disjoint paths, but focusing in maximizing the path bandwidth and minimizing the delay (shortest-widest paths [Wang and Crowcroft, 1996]).

An overview of the analysed multiple path computation algorithms can be found in table 4.2.



Table 4.2: Evaluated multiple-path computation algorithms

Name	Problem	Constraints	Complexity	Observations
[Yen, 1971]	$k$ -shortest simple paths	No	$O(kn(m + n \log n))$	Uses Dijkstra's algorithm multiple times
[Eppstein, 1998]	KSP	No	$O(m + n \log n + k)$	Computes solutions with cycles
[Li et al., 1992]	Disjoint KSP	No	$O(n^{k-1}m)$	Only works with acyclic directed graphs
[Suurballe and Tarjan, 1984]	Pairs of link disjoint paths	No	$O(m \log_{(1+m/n)} n)$	Finds a pair of disjoint paths between a source and all the other graph nodes
[Chap et al., 2011]	Disjoint KSP	No	Not mentioned	Uses a link-disjoint degree to calculate partially disjoint paths
[Yuan, 1999]	KMCSP	Any 2	$O(k^2nm)$	Positive / negative weight graph
[Liu and Ramakrishnan, 2001]	KMCSP	Yes	$O(khd^2(h + r + \log(khd)))$	Extends the A* algorithm
[De Neve and Van Mieghem, 2000]	KMCSP	Yes	$O(kn \log(kn) + k^3Cm)$	Combines all the path metrics into a single one
[Van Mieghem et al., 2001]	KMCSP	Yes	$O(kn \log(kn) + k^2Cm)$	Similar to TAMCRA, but optimally finding the existing $k$ paths
[Guo et al., 2003]	Link-disjoint KMCSP	Yes	Not mentioned	Calculates a pair of disjoint paths using SAMCRA

### 4.3 Path computation in Software-Defined Networking

Due to the centralized management architecture present in SDN, the used controller application can have full knowledge of the existing network topology, making path computation a simpler task (despite the need of mechanisms that allow the controller to know the managed network state), rather than use distributed algorithms among the existing nodes. Along with path calculation, a number of different specific tasks can be performed; installing alternative paths upon path failure, enhancing the QoS and QoE of the existing traffic demand, providing load balancing for dynamic installed flows and enhance the scalability of network nodes in data centers are some of the applications of path calculation inside SDN.

Regarding the installation of alternative paths as a fast failover solution when using OpenFlow, auxiliary protocols can be used to monitor the state of the present links, such as Bidirectional Forwarding Detection (BFD) [Katz, 2010] or Connectivity Fault Management (CFM) [van der Pol et al., 2011]. Additionally, different link reconfiguration schemes can be used; Lee et al. [Lee et al., 2014] created a pre-planned path protection solution that covers three different schemes; Link Protection (existing spare capacity in links can be used by different links), Destination Only (alternative paths are calculated based on the flow destination address) and Interface Specific Forwarding (packet forwarding rules take in consideration the source node id and destination address). By using preplanned paths, when link failure occurs in the network, it is only required to modify the flow entries in the source and destination nodes, rather than reconfiguring each node of the new path.

Through the monitoring capabilities of OpenFlow, it is possible to collect network traffic metrics from big topologies (e.g. fat-tree networks), a function sometimes hard to perform with traditional load-balance-aware routing mechanisms, like the equal-cost multipath routing protocol (ECMP) [Hopps, 2000], due to the high overhead caused by the huge density of the network. A dynamic load-balancer with multi-path support was built as an OpenFlow controller application, supporting traffic adaptations in the network [Li and Pan, 2013]. This load-balancer runs an algorithm that selects links with more avail-

able bandwidth when performing path computation (by monitoring the respective OpenFlow switch port statistics). This is a simple recursive algorithm that calculates paths running either from a lower to a higher topology layer level or in the opposite direction, depending in where the destination node is located. Besides its better results comparing to other existing load-balancing techniques, the algorithm is specific only to be used with fat-tree topology networks, not being an actual solution to other existing network topology problems.

QoS routing, as said before, is another example of an usage of path computation within SDN. OpenQoS [Egilmez et al., 2012] distinguishes incoming flows in a network as multimedia or data flows, distributing them among two types of paths; multimedia flows are assigned to QoS guaranteed paths and data flows are placed among existing shortest-paths. QoS-reserved routes are calculated through the resolution of the DCLC problem with the application of the LARAC algorithm, using constrained path delay values and through the minimization of a cost function built from the path congestion and path delay.

The CSP is also solved as a mean to provide QoS routing in an optimization framework used for streaming video in an OpenFlow network [Egilmez et al., 2013]. In this framework, video transmission can be distinguished by two levels of QoS, when the used codec allows video encoding through different layers (a feature not always present in commonly used video codecs). This way, the base layer of a video (with the most significant information) is assigned to the highest QoS level, the enhancement video layer to the lower QoS level and all the remaining (data) flows are routed through the shortest path (with an best-effort approach). This routing problem is solved again recurring to the LARAC algorithm, with a delay constraint and a cost function based on the weighed sum of packet loss and on the delay variation.

The optimization of path computation in order to maximize the aggregated QoE of a network domain is demonstrated in Q-POINT [Dobrijević et al., 2014], a framework that uses a mixed integer linear programming model that calculates the most QoE-aware suitable paths for each existing flow, distinguishing between audio, video and data flows, setting up the originated flow rules in each OpenFlow switch through a SDN controller.

Regarding the usage of the algorithms compared in the previous sections for

path computation in SDN, different choices can be made according to the type of problem. For the calculation of a single SP between a source and a destination node, both Dijkstra's or Bellman-Ford algorithms are suitable, but since Dijkstra's time complexity is lower than Bellman-Ford and in this specific scenario the weight of a path cannot be negative, Dijkstra's algorithm can be considered a more efficient choice. The A\* problem requires a good heuristic function, that can be hard to implement, having an exponential complexity in a worst case scenario, making it not affordable for the calculation of SP in a SDN-based network.

Yen's algorithm is a simple solution for solving the KSP problem, as it can be easily implemented as a SDN controller module, having as input a computer network graph. The remaining analysed algorithms for this specific problem would not suit this type of scenario, as either their input cannot contain cycles or the generated solutions can include a node more than once in a single path.

The problem of computing multiple disjoint paths is more complex and it is considered NP-Complete. The studied solutions are built for specific input scenarios or they do only generate a pair of disjoint paths. While the calculation of a pair of link or edge disjoint paths is meant principally to provide a backup path solution, more than 2 paths might be required to find when using MPTCP (if the number of existing network interfaces is greater than 2). Hence, while it might not be possible to find  $k$  fully disjoint paths between 2 nodes, an approach that computes partially disjoint paths needs to be chosen.

In order to apply the knowledge of collected QoS metrics in a SDN network, a controller can also implement a path computation module using one of the described algorithms, based on different heuristic approximation functions. Though the listed MCP single path algorithms were designed to contain 2 QoS metrics (delay and cost with LARAC or any 2 metrics with EBF with the limited granularity heuristic), in more complex scenarios it might be required to take more than 2 network performance variables in consideration. Nonetheless, easily any KMCSP algorithm can be turned to a MCP one when  $k = 1$ . In a similar way, when removing the number of constraints from the algorithm input, one can turn a KMCSP algorithm into one that computes KSP.

Either the A\* Prune or the SAMCRA algorithms can be suitable for solving the KMCSP, MCP and KSP problems, as they have similar complexity. The A\*

Prune efficiency depends in the quality of the used heuristic function, but as it occurs in the original implementation of  $A^*$ , it consumes a big amount of memory when expanding the existing nodes, which can be problematic with huge SDN network topologies. Therefore, since SAMCRA relies in the linear approximation weight function, it is a more proper solution.

## 4.4 Summary

An overview of existing path computation algorithms was presented in this section. The simpler single-path algorithms studied (Dijkstra's,  $A^*$ , Bellman-Ford) are used to calculate solve the SP problem and are used as a reference or as subroutine calls in other algorithms. More constraints can be added to the SP problem, originating the MCP problem, which is proved to be NP-Complete. In order to reduce this complexity, approximation and heuristic functions can be used and still be able to solve this problem in polynomial time (although sometimes the best solution found might not be the best existing one).

Additionally, existing solutions to compute multiple paths (KSP problem) and 3 variants of this problem were introduced: Link-disjoint (each link that forms a path cannot be used again by other calculated paths), node-disjoint (the same as link-disjoint but a node cannot be included in more than one path) and KMCSP. While the KSP problem can be solved with polynomial complexity, these 3 versions are more complex, and similarly to the MCP problem, the used algorithms require approximation and heuristic methods in order to compute with admissible times.

The usage of path computation with SDN were discussed and the usage of the previously analysed algorithms in a SDN context were reviewed. After analysing the current state of the art in SDN controllers and related work, it is noticeable that currently there are not any publicly available applications that perform constrained multiple path computation. This makes the development of an application with support for multiple path transport protocols and constrained path computation based in the existing network QoS metrics a significant contribution for the existing work in SDN and related areas.



# Chapter 5

## Proposed Architecture

In order to build and evaluate the proposed controller application that will allow enhanced path computation in a SDN network, it is needed to define the testing environment where the experiments will be held, along with the specification of the application to be developed. These initial steps will support the future work to be done, making it more structured, organized and with reference documentation to follow.

This chapter will firstly introduce the goals to be achieved with the evaluation of the developed application, followed by a description of the tools that are required to build its testing environments.

An elaborated definition of the framework that will allow the creation and modification of experimenting scenarios is presented, combining the usage of the tools previously described. The specification of the developed SDN controller application is exposed, listing and detailing each of its modules. Finally, details regarding the implementation of the application are shown.

### 5.1 Introduction and goals

When performing experiments with computer networks where it is required more than one device to build the desired testing scenarios, there are usually three possible approaches to achieve that goal: either the network scenario can be build using physical devices, using an emulated environment or using a hybrid

approach, combining the usage of physical devices with emulated ones.

While the usage of physical devices represents typically existing network environments in a more accurate level, such as the components physical delay or existing radio interference (when considering a wireless environment), its setup, configuration and device synchronization becomes more complex as the number of used devices increases, or when it is required to perform a topology modification.

Since the emulation of network environments is made through software, it becomes easy to create and modify new network topologies. However, there are some limitations when choosing this approach, such as the processing power of the device used to run the emulator (which can restrict the number of emulated nodes or even its running performance), running specific software or different operating systems versions inside the emulated nodes or the difference in the transmission/propagation times from the ones obtained in physical testbeds. Therefore, by taking in consideration the previously aspects, it is crucial to have a machine with a huge amount of processing power and a network emulator that maximises the independence of shared resources between the host machine and the emulated nodes.

A combination of emulated and physical network devices can be specially useful when testing a scenario where there are fixed components in a network (e.g. a SDN controller) and components that change among scenarios (endpoints or OpenFlow switches, for example).

Through the configuration of different computer networks through the usage of an network emulator, it will be possible to perform tests in SDN-environments, while exploring its behaviour when using MPTCP, along with different path computation and flow scheduling approaches that take in consideration the actual state of a network, obtained through the monitoring of statistics regarding the consumption of existing network resources.

These experiments will have as a main goal the proposal of an application that will run inside an OpenFlow controller and will allow the computation of paths inside a managed network using different strategies, for existing and new, unmatched flows.



## 5.2 Used tools

This section will describe the tools that will be used for performing experiments in the context of SDN.

As referred in chapter 2, a fundamental component in SDN experiments is the controller, responsible for managing the SDN-capable devices. Therefore, Opendaylight will be presented, along with Common Open Research Emulator (CORE), a network emulator capable of creating multiple nodes that will support OF. This will be possible by using Open vSwitch, a software-based switch vastly used in SDN environments, as an alternative to hardware switches.

### 5.2.1 Opendaylight

As it was previously mentioned in section 2.2, Opendaylight was chosen as the preferable SDN controller to use, mostly due to its well structured architecture and its huge and active user and developer community. From its large set of features, some of the most important that will be used include the OVSDB and Flow Manager Northbound APIs, and the Topology Manager and OpenFlow Plugin internal module.

A more detailed explanation regarding how Opendaylight will be explored in order to create a dynamic path calculation application can be found in section 5.4.

### 5.2.2 Open vSwitch

Open vSwitch (OVS) is a software-based switch, implemented at kernel level. Besides its native switching feature, it supports a huge number of networking features, targeted mostly to a production level usage.

Some of the more important features include the configuration of QoS policies, VLAN tagging, STP, tunnelling protocols (e.g. Generic Routing Encapsulation (GRE) [Hanks et al., 2000] and Virtual Extensible LAN (VXLAN) [Mahalingam et al., 2014]) implementation and usage of OpenFlow for setting up flow rules and the implementation of monitoring protocols, such as BFD, CFM, NetFlow [Claire, 2004], sFlow and others.

Since it is software-based, it is vastly used on network virtualization environments, bridging physical network interfaces (connected, for instance, to a hypervisor) with multiple virtual network interfaces connected to locally deployed virtual machines.

Open vSwitch supports remote management connections through the usage of the OVSDB protocol, which is often implemented in SDN controllers, such as OpenDaylight, as explained in section 5.2.1. Since the configuration of OVSDB is based on a database, a manager can easily configure an instance of OVS through the manipulation of table row entries (e.g. from the Port, Bridge, QoS, Queue tables). The current design of OVSDB schema can be seen in figure 5.1.

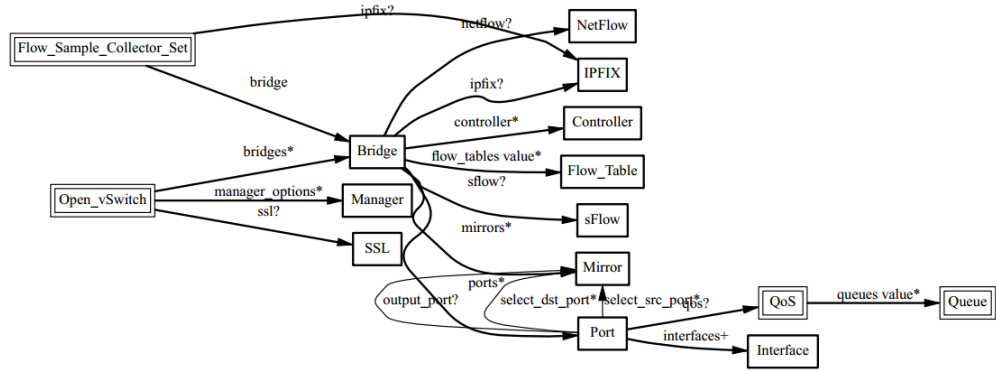


Figure 5.1: OVSDB Schema [Pfaff and Davie, 2013]

### 5.2.3 CORE

The Common Open Research Emulator (CORE) [Ahrenholz, 2010] is a network emulation framework that uses Linux containers to create network nodes and existing operating system bridging tools to establish links among them. Additionally to the emulated networks, it can be used to connect to existing networks connected through the device's network interfaces.

In order to create and manage network nodes and links there exist two possibilities. The first one is through its graphical user interface (GUI), as it can be seen in figure 5.2, which facilitates the design and drawing of network topologies on a visual canvas. The second method is through scripts written using an

existing Python API, making it possible to specify and define network programmatically. This was the method used to build the framework that allowed the creation of the testing scenarios, referred in section 5.3.

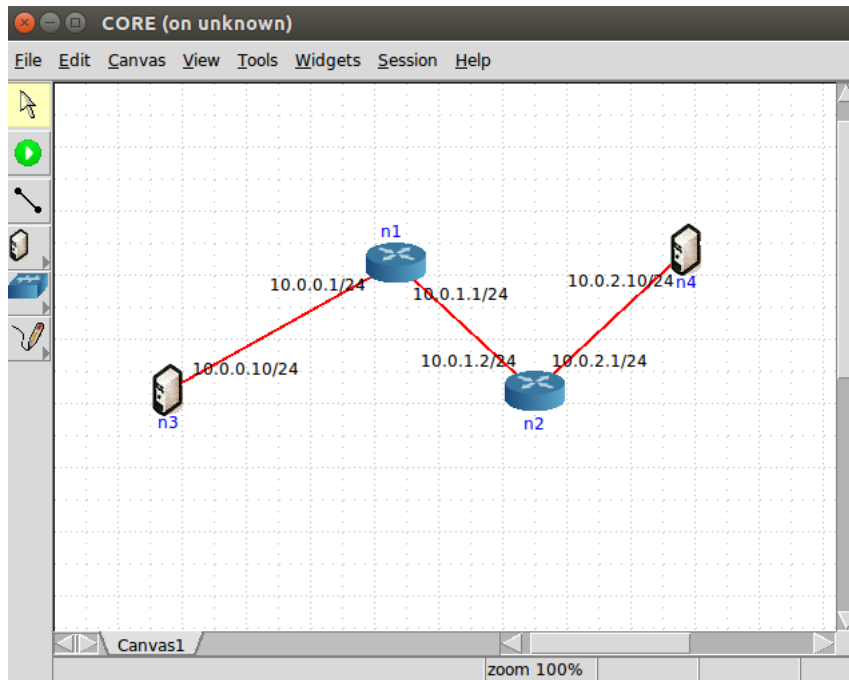


Figure 5.2: Example of the usage of CORE’s GUI

Besides the creation of network nodes, it is possible to run services inside the created nodes. These can be either previously shipped with CORE’s software package (e.g. Secure Shell (SSH), Quagga, Dynamic Host Configuration Protocol (DHCP) server), or customized by the user, through the specification of the desired start-up and shutdown commands, along with the directories used by the service. This allows SDN-related applications to be run inside a CORE node, such as OVS or OpenDaylight.

When analysing with other network emulators that are suitable for SDN-based network prototyping, Mininet [Lantz et al., 2010] is vastly used among the researcher community. It provides an easy CLI that enables users to deploy already defined network topologies, or a Python API that allows them to manually define the topology to be created.

Mininet also uses OVS and automatically configures it on each of the switch

nodes of the topology (it creates the required, bridges and ports, and connects it to the specified controller IP address, if specified). However, despite its support to OVS, it is not as versatile as CORE when it comes to customizing the topology's endpoints. While CORE allows the users to easily specify predefined or customized services to run inside the node, startup and shutdown scripts and node folders, Mininet does not offer similar support to node customization, only allowing to run command-line commands from its management CLI, making it more difficult to perform that same operations that are possible with CORE. Consequently, this makes CORE a more suitable option in terms of customize experimenting scenarios, which is the main reason for its choice as the used network emulator.

### 5.3 Evaluation platform

In this section there will be presented the components that allowed the construction of a testing framework environment, supporting the tools listed in the previous section.

This framework is based on a Scenario Creator, responsible for creating scenario files that describe the used topology and events that occur during the experiment, and on an Experimenter framework, that parses scenario files, builds the respective network topology and runs all the events specified in the scenario file.

#### 5.3.1 Scenario Creator

In order to explore the capabilities of the Python API provided by CORE and facilitate the creation of new topologies to be used on testing scenarios, a Scenario Creator auxiliary application was built. Besides the scenario topology, it also allows the specification of different types of nodes to be run in CORE (Endpoint, Router, OpenDaylight and OVS), QoS configurations in OVS nodes and events to be held during the experiments.

Through the definition of the topology nodes, existing links (along with their parameters, such as bandwidth, delay or packet loss rate) and how they are

connected to each other, the Scenario Creator instantiates the proper CORE objects and configures them according to the provided specifications.

The following events can be included in a scenario, each following a timestamp that determines when they occur:

- Start and stop sending network traffic
- Start and stop receiving network traffic
- Setting up a node's interface up or down
- Add a network route to a node
- Modify an existing link in real time (bandwidth, delay, jitter, packet loss and duplicate rate)

After running the Scenario Creator, it is produced a JSON output file containing all the data regarding the specified scenario. This output format allows other applications to easily read and parse its contents with existing libraries, independently of their programming languages.

### 5.3.2 Experimenter framework

Concerning the requirements of performing experiments in different scenarios (previously specified by the Scenario Creator), this Experimenter framework allows an user to run experiments with additional configurations, adjusted to the context where they are run. Examples of these configurations could be setting up MPTCP in the host machine, or starting traffic sniffers before starting the experiment.

The following enumeration explains each of the steps that this framework takes when running an experiment, as described in figure 5.3.

1. A scenario file is created by the Scenario Creator, specifying the scenario topology (nodes and links), existing OVS QoS configurations and network events;
2. The experimenter framework reads the scenario file, provided as input;

3. The Opendaylight controller is started, along with the developed application;
4. Using CORE's Python API, the framework builds the network topology, as described in the scenario file. This includes the creation of network nodes, links and its parametrization. Any other experiment configuration, if specified in the executable file, does also run in this phase;
5. The OVS nodes connect to the controller and start to be managed by the path calculator application;
6. According to the traffic demand data present in the events section scenario file, the traffic generators senders and receivers are instantiated in the respective nodes and the senders start to generate traffic. During this phase, all the other events are taken in consideration and run according to their start time;
7. By the end of the experiment, log files are produced, based on the results perceived in the endpoint nodes. These log files contain information regarding the average, minimum and maximum delay, packet loss percentage, average bitrate, number of transmitted packets and number of received bytes, for each flow.

## 5.4 SDN controller application specification

In this section it is presented the specification of an application to be run within the Opendaylight SDN-controller.

This application will allow different path routing strategies; besides a traditional single-path routing strategy, it will also including multiple-path routing and low-latency aware path computation. Additionally, when multiple paths are available, different flow allocation mechanisms will also be evaluated.

The main goal of using these different strategies is to evaluate and compare their behaviours among each other.

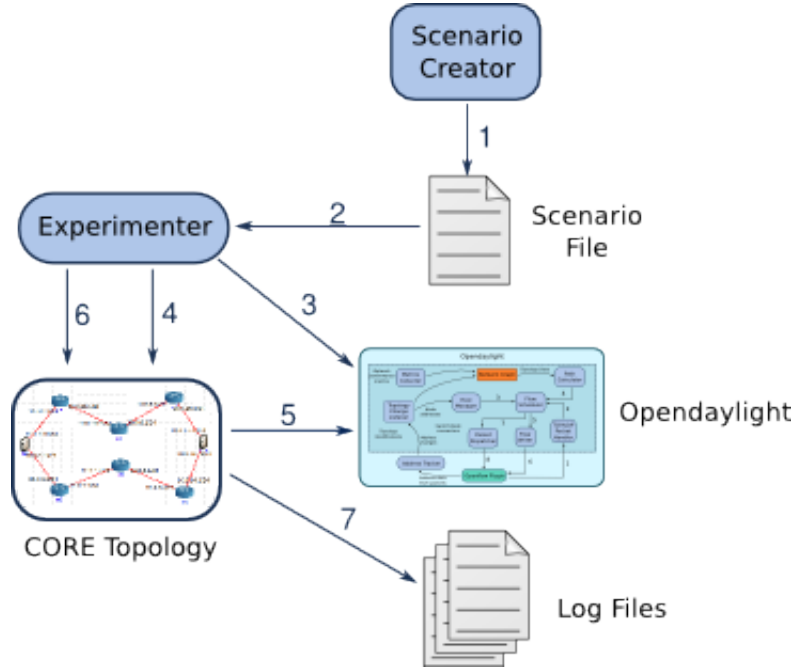


Figure 5.3: Diagram of the experimenter framework

Following up this section, it will be made a detailed description of the components that integrate the application (figure 5.4), having this description complemented with the workflow of the application when the controller receives an incoming packet from a managed OF switch.

#### 5.4.1 Components description

Opendaylight is built over an elaborated modular and service-oriented architecture. Its last stable version uses Karaf, an OSGi-based framework that manages the installation and deployment of services inside the controller.

This controller uses a model-driven approach to create a service abstraction layer (MD-SAL), contributing to the standardization of the existing north and southbound APIs, and specifying how services use the application internal data storage and communicate with each other internally.

With MD-SAL, a service is described using the YANG data modelling language. With it, it is possible to specify its data structures (and respective relationships), methods, dependencies and notifications. Based on this specification,

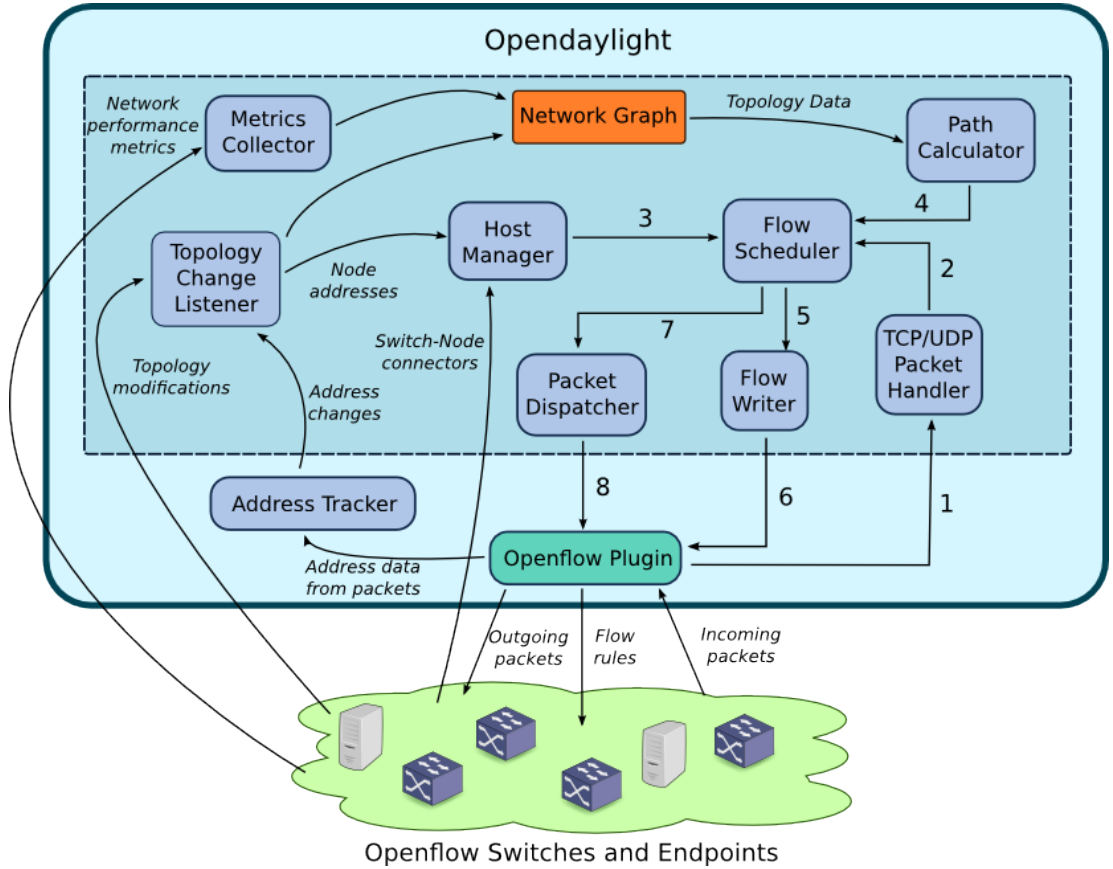


Figure 5.4: Architecture of the SDN controller application

it is possible to generate the Java source code of the service through the usage of code generators that parse YANG models.

The current distribution of the Opendaylight controller already includes modules that will be used for supporting the application to be develop. These are the OpenFlow Plugin, Address Tracker, Packet Handler and Topology Manager, respectively. In addition, new components will be also created, in order to support the previously described application features, which include the Metrics Collector, the Path Calculator, the Flow Scheduler, the Host Manager, the Packet Dispatcher, the Topology Change Listener and the Flow Writer.

The following subsections will describe each one of the components that the application will use.



#### 5.4.1.1 OpenFlow Plugin

The OpenFlow Plugin is an integrated component of Opendaylight that provides the required means of communication between Opendaylight and the managed network where it is connected. The usage of the OpenFlow Plugin can be distinguished between two different types of interaction with other Opendaylight components.

The first involves the communication between high-level components of Opendaylight and it is specially based on the MD-SAL architecture. One of the most important features of MD-SAL that the OpenFlow Plugin uses is the publication of notifications upon receiving a packet from an OF switch. The registered services in Opendaylight can subscribe to these notifications and access the packet data. It also provides an API that other services can use for managing flows on the OF switches, converting MD-SAL stored objects into OF API calls.

The second part of the usage of the OpenFlow Plugin is related with the communication with the network devices that support OF. This is made through the usage of an OF protocol library that is implemented according to the protocol specification (currently the used OF version is 1.3). With the required OF libraries it communicates with the OF devices through that are compatible with the same version of the protocol.

#### 5.4.1.2 Topology Manager

The Topology Manager is responsible for maintaining coherent the information stored in the topology database provided by Opendaylight. This includes the information about active endpoints in the network and OF-capable devices, along with the respective links between the components.

The topology data is obtained by parsing Link Layer Discovery Protocol (LLDP) messages exchanged by the network devices that support the protocol and through packets sent and received by the endpoints that and forwarded through the OF switches.

#### 5.4.1.3 Address Tracker

The Address Tracker is a built-in component of Opendaylight, originally implemented on the sample L2 Switch application.

When the controller receives a packet forwarded from an OF switch, the Address Tracker keeps track of the node that originally sent it and associates it to the OF switch where it is connected. These informations include the attachment points (identifier of the node and port of the respective OF switch), IP and MAC addresses, a timestamp of when it was firstly seen, a timestamp of when it was lastly seen and an identifier field for the node.

With this type of information, it is possible for other components to know where each endpoint is located in the network (e.g. when performing path computation to establish new flow rules between two endpoints).

#### 5.4.1.4 Topology Change Handler

As the name suggests, the Topology Change Listener handles modifications in the managed topology. These modifications includes the appearance and removal of new network links, nodes and addresses (IP and MAC).

Link modification notifications are generated by the Topology Manager, and upon receiving a notification of a change, the Topology Change Handler updates the existing Network Graph data structure with the respective modifications.

Node changes are tracked Opendaylight Inventory Listener, a built-in component in ODL that sends notifications whenever an OF-capable device joins/leaves the managed network. The Topology Change Handler reacts to notifications regarding new nodes in the network, and for each one it instructs the Flow Writer to install the base OF flow rules (IP and LLDP packets sent to controller. Remaining packets are dropped).

When new addresses are added to the topology information by the Address Tracker, the Topology Change Handler maps the new addresses to the respective node connectors (typically switch ports) from the OF nodes where the hosts with the addresses are attached in the network. This mapping is made through the Host Manager, whose functionalities are described in the next section.

#### 5.4.1.5 Host Manager

The Host Manager is responsible for providing data related to the existing endpoint and OF nodes connected to the network managed by OpenDaylight. It keeps maps of the L2 and L3 addresses and the respective switch ports where they are attached, along with each OF node controller port connector (the switch port that is used to establish communication with OpenDaylight).

The tracked data is stored locally, avoiding making a new database transaction for each request from other components. The Host Manager will only read data from MD-SAL regarding controller-switch connectors, but only in the case where the requested data is not locally available.

#### 5.4.1.6 Packet Handler

Incoming packets received by the controller are processed by the Packet Handler. Similarly to the Address Tracker, it is a component originally present in ODL's L2 Switch application, but modified to also handle TCP and UDP packets, additionally to Address Resolution Protocol (ARP), Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) packets.

When a TCP/UDP packet is received, the source and destination port numbers are extracted from the packet header, along with the fields from the lower layers (source and destination IP and Media Access Control (MAC) addresses) and the original packet payload. Then, the Packet Handler passes the collected data to the used Flow Scheduler.

#### 5.4.1.7 Metrics Collector

The Metrics Collector module is responsible for adding network performance metrics to the existing topology information. Currently the supported metrics are the link throughput, percentage of available and used bandwidth and one-way delay.

While some network metrics can be obtained passively without any additional configuration on the existing nodes by accessing existing statistics through the existing configurations (link throughput and bandwidth), in order to obtain the link delay it is required to periodically send packets through each link and keep

track of the time difference between when the packet was sent and when the packet was received. This requires an additional monitoring of the link-delay between the controller and the OF switches where the link is formed, which is subtracted from the final interval between when the packet was sent and when it was received back in the controller.

This module polls the monitored devices following an adjustable interval of time. For all the collected network performance metrics, the existing topology link data is augmented, making it possible to be independently accessed by other components in OpenDaylight.

#### 5.4.1.8 Flow Scheduler

The Flow Scheduler has as main task choosing an available calculated path between two hosts by following a flow pinning strategy. It is invoked by the Packet Handler, receiving as input the incoming packet header fields (TCP/UDP header data, IP and MAC source and destination addresses) and the original packet payload.

Based in the source and destination addresses, it gets the information about the switch ports where the endpoints are connected from the Host Manager (they need to previously have sent at least a packet to the network, so the Address Tracker can add the endpoint information). Then, if there are not calculated paths between the OF switch where the source and destination nodes are connected, the Flow Scheduler queries the Path Calculator for new paths.

The path where the new flow rules will be installed is then calculated following the configured flow pinning strategies. Currently, there are 5 flow scheduling strategies implemented:

**Static path flow scheduling** All the flows are assigned to the same path;

**Random path selection** As the name suggests, a flow is assigned to a random existing path between the available ones. It does not require to maintain any state variables, regarding the next path to calculate. However, it is a totally unpredictable strategy that does not take in consideration any input from the existing network/flow data;

**Hash-based path selection** A path number is calculated based on the result of a hash function that uses the incoming packet header fields as input, having its the output result restricted by the number of existing paths (similarly to ECMP). This strategy guarantees a deterministic behaviour, as flows with the same characteristics are always mapped to the same path, while keeping different flows generally mapped to different paths. By using a hash function, it is not required to keep stored information about the last flows assigned. Nonetheless, all the flow mapping is done without taking in consideration the existing network state;

**Round-Robin path selection** A new flow is assigned to the next path in the list of existing paths, in a circular manner (after the last path is assigned, the next one is the first). This behaviour makes Round-robin a predictive and easy to implement algorithm. However, when considering network paths, this approach can result in a worse behaviour when having paths with different characteristics, as the flows are not pinned according to the available resources. Additionally, it is required to keep in memory the current state of the algorithm, in order to be able to select the next path to assign to a new flow;

**Flow-based path selection** This algorithm assigns a flow to one of the paths that has the minimum number of assigned flows. It requires the monitoring of the existing flow counters for the managed links. Hence, the collected data must be accurate and precise, or the collected values can be incorrect, making this approach ineffective.

The Flow Scheduler then sends the calculated path information to the Flow Writer (for installing the new flow rules) and the original packet payload to the Packet Dispatcher, that will send it to the destination endpoint node.

#### 5.4.1.9 Path Calculator

The Path Calculator is responsible for the calculation of network paths between existing nodes. The calculated paths are used for installing new flow rules that

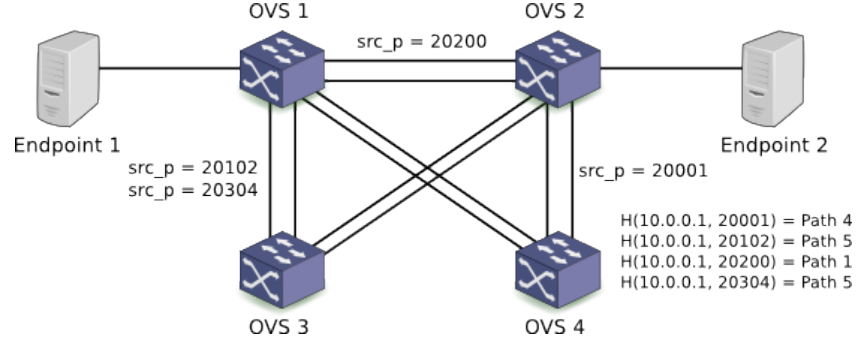


Figure 5.5: Hash-based load-balancing

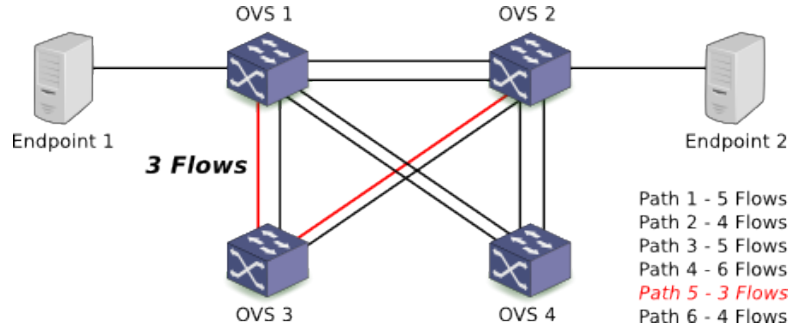


Figure 5.6: Minimum-flows based load-balancing

will allow network traffic to be routed between endpoint nodes connected to managed OF switches.

When it is required to establish one or multiple paths between two nodes, the Path Calculator uses the existing Network Graph and the currently configured path computation strategy and input to calculate the new paths. Three different path computation strategies were implemented. However, it is possible to easily implement new ones, without having extensive knowledge about the rest of the application architecture. The implemented strategies are the following:

**Single path routing** In this simple approach a single path is calculated by using Dijkstra's algorithm and all the traffic between two endpoints is routed through this path. When using MPTCP this strategy also routes all the sub-flows to the same path;

**Multiple disjoint path selection** Taking in consideration a shortest-path metric, multiple paths are calculated by using Yen's algorithm. This approach

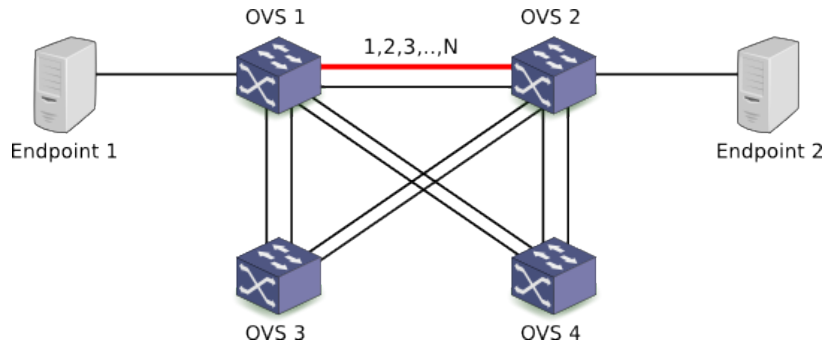


Figure 5.7: Static path load-balancing

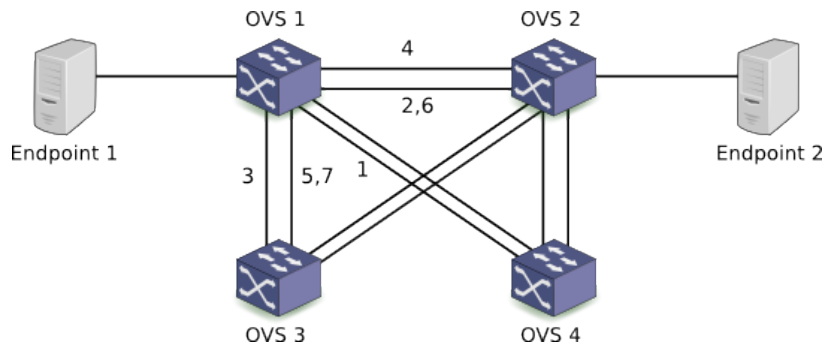


Figure 5.8: Random load-balancing

is made without taking in consideration node or link disjointness, which can result in having multiple shortest paths with repeated links. While the obtained paths are still valid solutions for connecting the source node to the destination, by having the same links shared in different paths, there is an increase of the risk of having bottlenecks in the used topology;

**Link disjoint path selection** Multiple paths are calculated by using sequentially Dijkstra's algorithm between the source and destination. For each time that the algorithm runs, the links present in the previously calculated path are ignored, guaranteeing link disjointness. However, this solution is not always effective, as it does not work when there must exist partial disjointness in the calculated paths (at least one link is shared between the existing paths);

**Constrained multiple path selection** Besides the length of the computed paths, this approach will also have as input the quality of the links, based on the

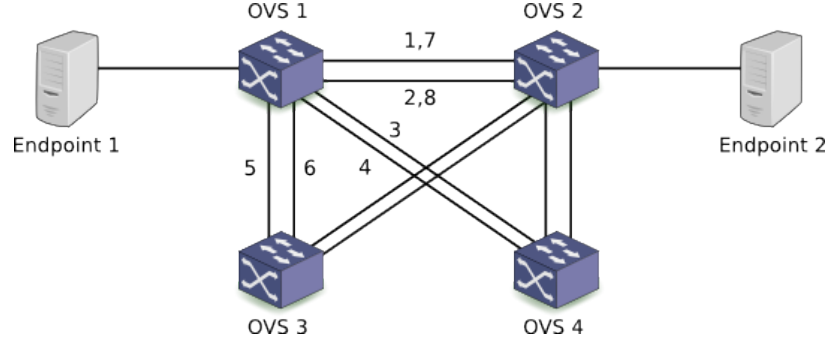


Figure 5.9: Round-robin load-balancing

metrics gathered by the Metrics Collector (delay and bandwidth usage), through the usage of the SAMCRA KMCSP algorithm.

Independently of the used strategy, after calculating the resulting paths, the Path Calculated sends the generated solution to the Flow Scheduler. The path computation strategy can be configured by reading a configuration file when the application starts.

#### 5.4.1.10 Flow Writer

This module receives as input calculated path solutions from the Flow Scheduler and the original packet header data that the controller previously received.

According to each node and port that is included in the generated solution, the Flow Writer creates flows with the header data of the packet being used in the matches part of the flow (i.e. source and destination IP address and source and destination port numbers). For each flow that is created, the OpenFlow plugin is used, handling the encoding of the flow to a standard OF call.

Additionally, in order to make communication between the source and destination nodes bidirectional, the Flow Writer also creates flows with the source and destination fields reversed (original source addresses and port numbers become the destination fields in the created flows' matches and vice versa).

#### 5.4.1.11 Packet Dispatcher

The Packet Dispatcher is responsible for sending OF packet-out messages from the controller to the managed OF nodes through the Packet Processing Service,



a built-in ODL component. These messages contain a packet payload and the respective ingress and egress OF switch ports that are used to identify the source and destination port of the packet.

The Flow Scheduler uses the Packet Dispatcher for delivering the original packet that arrived to controller to the destination endpoint node.

### 5.4.2 Application workflow

This section describes the workflow of the developed controller application, presenting the chain of events and the interaction between components that is triggered upon the reception of a new packet in the SDN controller. The same workflow is represented in figure 5.4.

1. The OpenFlow Plugin triggers a Packet Received notification that is handled by the Packet Handler. The packet payload is decoded and header fields are extracted from the obtained information;
2. If the received packet is a TCP or UDP packet, the Packet Handler sends it to the Flow Scheduler for further processing;
3. Based in the source and destination MAC and IP addresses of the received packet, the Flow Scheduler obtains from the Host Manager the information about the OF switch ports where the endpoints are connected;
4. If there were not any previous calculated paths between the source and destination OF switch, the Flow Scheduler queries the Path Calculator for new paths. The Path Calculator uses the configured path computation strategy to calculate one or multiple paths, with or without constrained network metrics. The topology information is obtained from the Network Graph storage data structure;
5. With the set of links that form the path between the source and destination nodes, the Flow Writer module produces a set of OF rules that are matched in the incoming packet header fields;
6. Each created flow from the calculated path is sent to the OpenFlow Plugin and then installed in the respective OF switch nodes;

7. The original received packet is sent to the Packet Dispatcher, along with the information about the ingress and egress destination switch ports;
8. The Packet Dispatcher pushes a packet-out message to the destination OF switch through the Openflow plugin, so it can reach the endpoint with the corresponding address.

## 5.5 SDN controller application implementation details

Following up the specification presented in the last section, this section describes in detail the implementation of each component from the application.

### 5.5.1 Packet Handler

The Packet Handler module is responsible for subscribing to incoming OpenFlow `PacketIn` messages received by the controller. These messages contain the data from packets who were not matched by any installed flow rule in the OF switch from where they were sent.

The notifications are subscribed using MD-SAL's `NotificationProviderService` and each packet type is processed accordingly using a dedicated packet decoder. The packet decoders are started when the Packet Handle module is loaded in Opendaylight. The following section details the implementation of the decoders.

#### 5.5.1.1 Decoders

Each packet decoder is responsible for listening to one type of notification generated by MD-SAL, corresponding to one type of packet from the ones supported (Ethernet, Ipv4, TCP and UDP).

After receiving a notification, firstly the decoder verifies if the packet fields correspond to the ones from the decoder packet type (`canDecode` method). This is made mainly through the inspection of the protocol fields present in the packet header.

Upon this confirmation, the `decode` is invoked, and accordingly to the packet type, its header fields are extracted from the received packet payload and stored in the new packet object. In order to store the packet fields in adequate data structures, a YANG model of the new packet types must exist in the Packet Handler model module, which specifies the existing packet header fields and respective data types.

Besides the new packet fields, the YANG model also specifies a new notification type, used to signal when a packet is decoded from the decoder, which can be received by other components in Opendaylight. The Ethernet and Ipv4 YANG packet models were already present in the L2 Switch application and were used without any additional modifications. The TCP and UDP models were implemented as a requirement for the transport layer OF packet matching, being one of the contributions presented in this work.

### 5.5.2 Address Tracker

The Address Tracker is divided in two main components: the Address Observers and the Address Observation Writer. The first one keeps track of new network addresses from incoming packets in the controller and the second one writes the changes made in the MD-SAL database storage.

When the Address Module starts, firstly it starts the Address Observation Writer, passing it the used instance of MD-SAL data broker service and setting its `timestampUpdateInterval` variable value (read from the module's configuration file).

The Address Observers are started after, and each of them its registered as a MD-SAL notification listener. This is made through calls to the `registerNotificationListener` method of MD-SAL's notification provider service.

#### 5.5.2.1 Address Observers

The Address Observers listen to incoming packet notifications received by the controller and collects address information about the existing endpoint nodes.

It can subscribe to IPv4, IPv6 and ARP packets which are received through

the `Ipv4PacketReceived`, the `Ipv6PacketReceived` and `ArpPacketReceived` notifications, respectively. The type of packet notification subscriptions is defined by the `observeAddressesFrom` variable, set in the module's configuration file.

For each received notification, the Address Observers extract each packet from the packet chain built by the Packet Handler (`RawPacket`, `EthernetPacket` and `Ipv6Packet/Ipv4Packet/ArpPacket`, depending on the type of packet the Address Observer is responsible to handle). The ingress port where the packet came is obtained from the `RawPacket`, the source MAC address from the `EthernetPacket` and the packet source IP address from the respective IPv4, IPv6 or ARP packet.

With the collected fields, the Address Observer calls the `addAddress` from the Address Observation Writer to add the new address to the MD-SAL database storage.

#### 5.5.2.2 Address Observation Writer

As it was previously mentioned, the Address Observation Writer stores new IP and MAC addresses from endpoint hosts and the respective OF switch port where it is connected in the MD-SAL data tree.

For each address that is added, it is also added a timestamp to set the first and last time the address was seen in the network. If an address was previously associated to a switch port and the interval since the last address apparition is greater than the value set for the `timestampUpdateInterval` variable, the last seen value is updated. This verification is made through a read-only transaction that gets the existing seen addresses in the respective switch port.

All the changes are updated in the MD-SAL database through a write-only transaction that inserts an `ArrayList` of instances of the `Addresses` class into the `AddressCapableNodeConnector` augmentation of the switch port object instance reference (`NodeConnectorRef`) where the address was seen. Additionally it is used a lock-safe mechanism that allows only one address observation to be made at the same time on a switch port.

### 5.5.3 TCP/UDP Packet Handler

The TCP/UDP Packet Handler module is implemented in the `IncomingPacketHandler` class, and as the name suggests, it processes incoming TCP and UDP packets received by the controller. These packets originally are processed by a TCP or UDP decoder from the Packet Handler module, as referred in section 5.5.1, and are obtained by subscribing to the `TcpPacketReceived` and `UdpPacketReceived` notifications, respectively.

When one of these packets enter the `onTcpPacketReceived` or `onUdpPacketReceived` methods, all the previous layers packet data is extracted by accessing the built packet chain, which contains all the previous accessed packets. This procedure involves the extraction of the source and destination MAC and IP addresses, as well as the original packet payload (later used to send it to the packet destination node).

Following this step, the TCP/UDP packet data and the other collected values are sent to the used Path Scheduler, through the `processTcpPacket` and `processUdpPacket` methods.

### 5.5.4 Packet Dispatcher

By using Opendaylight's Packet Processing Service, the Packet Dispatcher module is able to send `PacketOut` messages to OF switches managed by the controller. These messages contain the payload from packets received previously by the controller, and are sent after the path calculation, flow scheduling and flow installation is completed.

The `dispatchPacket` method receives as input the byte array corresponding to the packet payload, the identifier of the node where to send the packet and the destination IP address. Using the Host Manager, the Packet Dispatcher obtains the identifier of the switch port that is connected to the controller (to be set as the ingress port) and the switch port where the host with the destination IP address is connected (to be set as the egress port). It then uses the `sendPacketOut` method to build a `TransmitPacketInput` object with all the obtained fields, which is used as input in the `transmitPacket` method of the Packet Processing Service.

### 5.5.5 Flow Writer

The Flow Writer module creates **Flow** objects that are transmitted to OF switches by using Openflowplugin's SAL Flow Service.

Its **addPathFlows** method is called by a Flow Scheduler every time a new path need to be established, using the incoming packet header fields as the flows' match. It receives as input the source and destination Nodes, IP and MAC addresses and TCP/UDP port numbers, as well as the protocol number (for identifying UDP or TCP) and the switch port identifiers where the source and destination hosts are connected. Firstly it installs the flows that will send the matching packets to the endpoint nodes, by calling the **addSrcToDestFlow** (twice, according to the source and destination node, respectively), which creates a **Flow** object and installs it in the respective switch. After that, flows with the same match are installed for each link presented in the built path. Since it is common to have bidirectional communication between two nodes, two flows are installed per link, having all the match fields reversed (source and destination addresses and port numbers) in the second one.

The flows installed in the path are temporary, as they are configured with an idle time out, making them expire and consequently being deleted from the switch after this timer is reached. The idle time out (in seconds) can be configured by setting the **pathFlowsIdleTimeout** in the application main module configuration file.

Besides the creation and installation of the flows required to maintain a path for incoming TCP and UDP packets between a source and destination node, the Flow Writer is also responsible for the installation of the initial flow rules that are installed when a new node establishes a connection with the controller. These flows use as match the IP and LLDP packet types and are used to send the respective incoming packets to the controller, being installed statically (hard and idle time out are not set).

A unique identifier is used for each flow, being stored in the **flowIdInc** variable, which is synchronously incremented every time a new flow is created. This procedure guarantees that new flows will not override existing configurations from old existing flow rules.

For each flow that is installed, the Flow Writer uses the `writeFlowToController` method. This method builds the `Node` and `Table` (set to 0, the one used by default when looking for a flow match in OF) instance identifiers that are required to configure the path where the flows will be installed in MD-SAL's database. Along with the created `Flow` class object, SAL Flow Service's `addFlow` procedure is called, installing the new flow.

### 5.5.6 Host Manager

The mapping between existing network addresses (IP and MAC) and the switch ports where they are connected, as well as the switch ports that are used for connecting each OF switch to the controller are stored in the Host Manager. This data is available to other components whenever it is required to compute a new path, install new flow rules or send packets to OF switches.

The IP and MAC address map is updated externally by the Topology Change Handler, whenever new addresses are observed in the network and stored in the corresponding `HashMap` instances entitled `ipv4AddressMapping` and `macAddressMapping`, respectively. The information about the switch port can be obtained by providing one of these address types and calling either the `getIpNodeConnector` or `getMacNodeConnector` methods.

When it is required to get the switch port used by an OF node to connect to the controller (through the `getControllerSwitchConnector` method), firstly the Host Manager tries to obtain this information locally by accessing the `controllerSwitchConnectors` map. If it is not available, all the existing node information is pulled from MD-SAL for further inspection by using the `readInventory` method. For each port present on each node, the one used to connect to the switch is extracted and added to the local map instance. This identification is possible by analysing the port name, since it is always named `LOCAL`.

### 5.5.7 Network Graph Service

The Network Graph Service is a service that allows the construction of a network graph using data from the links present in the managed network topology. It

is declared as an interface, declaring methods for managing the graph links and signalling when there are changes in the network graph. By using an interface class, it allows the creation of different implementations of the service.

The current implementation of this service was adapted from the original L2 switch Network Graph Service implementation. It uses the Java Universal Network/Graph (JUNG) framework, which provides libraries for creating and manipulating the network graph.

Additionally to the graph data structure, a map of the existing links in the graph is maintained by storing a key computed with the source and destination port of the link. This allows the verification if a link is already present in the graph, when adding new links. The verification is made by using the `linkAlreadyAdded` method.

### 5.5.8 Topology Change Handler

The Topology Change Handler listens to data change events on topology links, nodes and new addresses.

The link and address data monitoring is made by listening to data change events created by MD-SAL's Data Service (`DataBroker` class). When there are changes in MD-SAL's internal data, the created data is obtained from the generated notification, and each object is analysed.

If it corresponds to an instance of the `Addresses` class, it means that a new endpoint node was observed in the managed network and the corresponding network addresses were extracted from exchanged packet data. Therefore, the Topology Change Handler updates the Host Manager with the address data and corresponding switch port where the host is connected.

If the changed data is an instance of the `Link` class, there were changes in the network topology (a new link was observed by the Topology Manager, or an existing one was removed from the network), and the module needs to run the `TopologyDataChangeEventProcessor` thread after, in order to update the Network Graph.

The `TopologyDataChangeEventProcessor` thread is programmed to run when scheduled by the Topology Change Handler. It adds the new links firstly by clear-



ing the previous existing links from the graph, and adding the ones currently present in the network topology, by performing a MD-SAL database query to the corresponding information (which returns all the network links). By scheduling this thread it is possible to add multiple links to the Network Graph at the same time, instead of running a thread individually for each link that was created, since the retrieved information always contains all the topology links. The thread scheduling interval can be adjusted by configuring the `graphRefreshDelay` option in the application configuration file.

New nodes are monitored by receiving `NodeUpdated` notifications from the Opendaylight Inventory Listener. When a new node connects to the network, the Topology Change Handler starts a `InventoryChangeProcessor` thread. This thread is responsible for using the Flow Writer to install the initial flows to the new node, by calling its `addIpToControllerFlow` and `addLldpToControllerFlow` methods, setting up incoming IP and LLDP packets to be sent to the controller, respectively.

This component was adapted from the L2 Switch original implementation to be used in the current Path Calculator application project.

### 5.5.9 Metrics Collector

The Metrics Collector is part of the developed controller application but, similarly to the Packet Handler and the Address tracker, it runs independently as a standalone module that can be used by other applications.

Some of the components from the main application were also required in this module, such as a Flow Writer, a Topology Change Handler and a Host Manager. Therefore, the same components were introduced in the Metrics Collector, with modifications adapted to the requirements of the module.

The current implementation of the Metrics Collector supports delay and bandwidth monitoring, which can be enabled/disabled by setting the `collectDelay` and `collectBandwidth` options from the module configuration file to `true` or `false`, respectively. The next sections describe in detail how these two were implemented.

### 5.5.9.1 Delay Monitoring

OpenDaylight does not provide any delay monitoring tools by default and, consequently, the existing topology statistics lack data regarding link and path delay.

In order to add delay information about the existing topology, two possibilities were considered: the first one involved using external software tools in the topology nodes for extracting latency information, e.g. One-Way Active Measurement Protocol (OWAMP) or the native Internet Control Message Protocol (ICMP) ping tool. The second consisted in implementing a delay monitoring component in the Metrics Collector module.

While using an existing tool would simplify the implementation of the application, since it would only be required to implement a scraper for parsing the output interface of the tool, it would bring other issues such as the requirement of the monitoring application to be installed on each managed OF switch, which can not always be possible, or having to deal with clock synchronization issues.

Including a new monitoring tool in the controller application would add additional complexity to its implementation, and consequently increase the risk of having bugs in the produced code. However, by implementing a delay monitoring tool in the controller it would be guaranteed that it would work on every managed SDN network. Hence, the delay monitoring was implemented by using an active monitoring technique that measured the interval between the moment packets were inserted in the network, and the one when they were received. This approach was found previously used in existing work [Phemius and Bouet, 2013; Van Adrichem et al., 2014], but implemented in different SDN controllers (Floodlight and POX, respectively).

The delay monitoring implementation is divided in two different different link delay metrics: the controller-switch delay and the delay between inter-switch links (figure 5.10). The final delay value that is used for the link metric consists in the one-way delay between the inter-switch ports and is obtained by subtracting the timestamp of when the delay monitoring packet was sent and half of both controller-switch observed round-trip times, from the timestamp when the inter-switch delay monitoring packet was received. Considering the following values:

- $T_{linkpktrreceived}$  the timestamp when the packet arrived from  $s2$

- $T_{linkpktsent}$  the timestamp when the packet was sent to  $s1$
- $\frac{s1}{2}$  the one-way delay between the controller and  $s1$
- $\frac{s2}{2}$  the one-way delay between the controller and  $s2$

The link delay (in  $ms$ ) between two switches,  $Delay(s1, s2, link)$ , can be expressed by the following equation:

$$Delay(s1, s2, link) = T_{linkpktreceived} - T_{linkpktsent} - \frac{s1}{2} - \frac{s2}{2} \quad (5.1)$$

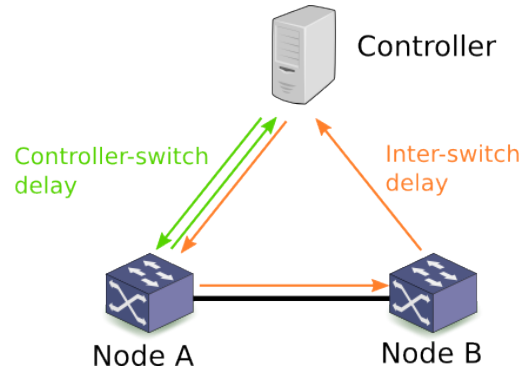


Figure 5.10: Link delay monitoring

For being possible to perform delay monitoring through the injection of packets in the network without interfering with existing network traffic and installed flow rules, a new type of packet was created using the EtherType value `0xbeef`. While the corresponding Ethernet frame contains a source and destination MAC address calculated based in the used OF switch datapath identifiers and port numbers, the `0xbeef` packet payload only contains an 8-byte value corresponding to a timestamp, mapped from a `long` value. The content of the Ethernet frame uses only 22 bytes, making it minimally bandwidth intrusive in the monitored network links. An example of one of this packet payload can be found in figure 5.11.

Since each delay monitoring packet needs to be sent back to the controller after being sent, a Flow Writer module is responsible for installing new flows that match the used `0xbeef` packet type, every time a Node Change Handler detects

Destination MAC	Source MAC	EtherType	Timestamp
0000f10000f3	0000e50000ca	beef	0000000aa88c9e78

Figure 5.11: Example of the content of a delay monitoring packet

a new node through a `NodeUpdated` notification, similarly the behaviour of the Topology Change Handler, described in section 5.5.8.

The interval used between each time the module sends delay monitoring packets to the controller-switch and inter-switch links can be adjusted by modifying the `controllerSwitchDelayInterval` and `linkDelayInterval` options in the Metrics Collector configuration file.

In the first version of the implementation of this module, all the links were monitored at the same time, which resulted in frequently having a large number of packets arriving back to the controller approximately in the same instances, contributing to the increase of the observed delay values due to the processing overhead (the Openflow plugin only generates the next packet received notification after the previous one is processed). In order to avoid this, an interval (in ms) between each sent packet was introduced in the module implementation. This value can be configured by setting the `delayPacketSendMaxInterval` and `delayPacketSendMinInterval` options. The interval generation strategy can also be set to be static (using only the maximum value) or randomized (having a value between the maximum and minimum option values) by setting the `delayPacketSendIntervalStrategy` option to `static` or `random`, respectively.

### 5.5.9.2 Bandwidth Collector

The Bandwidth Collector extracts bandwidth usage information from the existing OF switch ports. Contrary to delay monitoring, it obtains this data passively, by reading existing statistics from MD-SAL. The collected statistics include the number of transmitted bytes and current link speed.

By obtaining these values repeatedly over an interval of time, configured by the `bandwidthInterval` option (in ms), it is possible to estimate the current used bandwidth (in bits per second), as well as the available bandwidth, by subtracting the used bandwidth to the link speed.

After calculating the bandwidth estimation values, they are merged into the correspondent link `MetricsCapableLink` augmentation and stored in MD-SAL for further use by other components.

When monitoring the bandwidth utilization from the managed network nodes, several issues were found, regarding the update rate of these values. The component in OpenDaylight responsible for updating the data required for calculating the used bandwidth (byte counters) only updates these values every 3 seconds, which turned into a restriction when configuring the update interval of the Bandwidth Collector.

#### 5.5.10 Path Calculator

As it was previously described in section 5.4.1.9, the Path Calculator is responsible for computing paths between the existing network nodes. The application allows the usage of different path computation strategies, configurable by modifying the `pathStrategy` option in the application configuration file.

The `AbstractPathCalculator` class is used as an abstract base class required to be extended by every implemented path calculator strategies. It receives as its constructor input the used Network Graph Service and contains the declaration of the 2 `getPaths` methods that are used to calculate paths. Both these methods receive as input the identifier of the source and destination node of that paths to calculate, having the maximum number of paths to calculate as an additional parameter in the second one.

The following subsections will describe the currently implemented Path Calculation strategies.

##### 5.5.10.1 Single Path Calculator

The Single Path Calculator is implemented in the `ShortestPathCalculator` class. It uses Dijkstra's shortest path algorithm, present in JUNG's algorithms libraries, which returns a list with the corresponding path links, sorted by the visiting order between the source and destination.

Since this strategy only computes one shortest path between a source and destination, both path calculation methods are forced to generate only one path

in their output, having the `n paths` parameter ignored in the second `getPath` method.

The Single Path Calculator strategy can be used by setting up the `pathStrategy` option to `singlepath` in the application configuration file.

#### 5.5.10.2 Multiple Path Calculator

The Multiple Path Calculator is used by the `MultiplePathCalculator` class and it calculates multiple paths by solving the KSP problem. Paths are calculated using Yen's algorithm, which computes the  $k$ -shortest paths in the network graph, implemented in the Multi-Layer Visualization Tool (MuLaViTo) framework. Similarly as in the Single Path Calculator, this implementation uses JUNG's libraries.

Since the used Yen's algorithm implementation operates with directed edges and the used Network Graph implementation uses undirected edges for representing the existing network links, an additional procedure creates an auxiliary graph with directed edges with unitary weight from the existing topology before running the algorithm. It can be called by using the `getShortestPaths` from the `Yen` class object.

The `n` parameter from the `getPaths` is used for specifying the number of paths to calculate. If it is not specified, a predefined value of 5 paths is used as input for the algorithm.

The Multiple Path Calculator strategy can be used by setting up the `pathStrategy` option to `multipath` in the application configuration file.

#### 5.5.10.3 Link-disjoint Path Calculator

The Link-disjoint Path Calculator is implemented in the `DisjointPathCalculator` class and calculates multiple disjoint paths through multiple calls to the `DijkstraShortestPath` class (JUNG's libraries).

For each call to the `getPaths` method, a temporary network graph is created from the previous existing one, removing the edges that were used in the last calculated path. If there are not as many disjoint paths as requested, the algorithm stops and returns the computed paths.

The Link-disjoint Path Calculator strategy can be used by setting up the `pathStrategy` option to `disjoint` in the application configuration file.

#### 5.5.10.4 SAMCRA Path Calculator

The SAMCRA Path Calculator can perform multiple path computation with multiple constraints through the usage of the `SAMCRAPathCalculator` class, which uses instances of the `SAMCRA` class for obtaining new paths.

It receives as input a length function parameter, set by the `samcraLengthFunction` option, defined in the application configuration file, which sets the objective function used to calculate the link and path lengths in the algorithm (`getPathLength`). Additionally, the length function is also used to specify the criteria that defines the path domination criteria (`isDominated`). The MCP length function is configured by setting the length function variable as `mcp`, the DCLC with `dclc` and the HCMB as `hcmb`.

The `samcraMetrics` option can be used to specify the link metrics that will be used for path computation with the MCP, separated by commas. Currently the supported metrics are delay (`delay`), used bandwidth (`usedBw`), available bandwidth (`availableBw`) and hop count (`hops`). Besides hop count, which is basically the sum of all the path links, all the metrics are extracted from the corresponding link by obtaining the values stored in their `MetricsCapableLink` augmentation.

The respective constraints are provided as input to the `SAMCRA` class as a `double` array and obtained through the `samcraConstraints` configuration option, in the same order as they were declared in the `samcraMetrics` one.

Extending the behaviour of the original algorithm, if a path is required to be found with given constraints but the algorithm fails to find it, the application proceeds to relax the algorithm constraints and run it again. This relaxation can occur up to three times, by multiplying the original constraints by 2, 4 and 8 respectively. If by then, SAMCRA did not find a path, a final iteration runs by setting up the algorithm with an unique minimum-hops metric with an infinity constraint, in order to guarantee that a path is found between the source and destination node, if it exists. An alternative to this policy was blocking the

incoming flow, not allowing the respective rules to be installed in a path and consequently, forbidding the communication between the two nodes. The impact of the used policy by the application when a path is not found by SAMCRA can be studied as future work.

If the application requires multiple paths to be found, but SAMCRA fails to compute the desired number (e.g. due to the path domination criteria used to remove redundant paths in the algorithm), it creates an auxiliary copy of the existing network graph, where it excludes the links that are present in the previous calculated paths, and runs the algorithm again, until either when the number of required paths is filled or the algorithm fails to find a path. This extension to the original application was done, since in scenarios where multiple flows between the same source and destination were established in a short amount of time (e.g. when creating a new MPTCP connection with multiple sub-flows), the link characteristics would not be different when running SAMCRA, forcing the algorithm to always return the same path.

### 5.5.11 Flow Scheduler

The Flow Scheduler is implemented on a high level by the `AbstractFlowScheduler` class. This class contains two abstract methods, `processUdpPacket` and

`processTcpPacket`, which are responsible for handling incoming UDP and TCP packets and need to be implemented in each Flow Scheduler that will be configured. The used scheduler is defined by the `schedulingStrategy` variable, specified in the main module configuration file.

Currently, all the implemented schedulers have the same initial behaviour, when a new packet is received. Firstly, through by using the source and destination IP and MAC address from the packet header, the scheduler queries the Host Manager, and if the addresses were previously seen in the network, it gets the respective OF switch and port where they are connected.

With the source and destination OF nodes, it checks if they are different from the ones existing in memory (`changedNodes` function), and if they are reversed (incoming packet source address corresponds to the stored destination one and



vice versa), through the `reversedPath` function. If the nodes are different, then it requests new paths to be calculated by the existing Path Calculator.

The scheduler then gets the next path number, according to the used flow pinning strategy. If the obtained path is valid, it calls the Flow Writer to install the required path flows and finally, it uses the Packet Dispatcher to send the original packet to the destination node.

The next subsections will detail how each of the implemented flow schedulers chooses the next path.

#### 5.5.11.1 Hashed Flow Scheduler

The Hashed Flow Scheduler uses a hash function for computing the next path number (`getHashBasedPath`). This function gets as input the received packet source and destination IP addresses and TCP/UDP port numbers and computes the hash code of a string containing those fields together (e.g. `10.0.0.110.0.0.215000150001`).

With the obtained value it is applied a mask to turn any negative value into a positive, finishing the procedure by returning the modulo of the existing number of paths.

This scheduler can be used by configuring the `schedulingStrategy` variable option of the application main module to `hash`.

#### 5.5.11.2 Minimum Flows Flow Scheduler

This scheduler uses information from existing OF flow counter statistics in order to obtain the number of flows assigned to each path. It can be configured by setting the `schedulingStrategy` variable option of the application main module to `flows`.

The `FlowManager` class is used for obtaining these values from the MD-SAL data storage. Its `getTableFlowOutputCounters` returns the present flow counters, for each one of the OF switch ports, by parsing the `OutputAction` value of each flow (if applicable) installed in the node default table (0) and incrementing the respective port flow counter.

The `MinFlowsFlowScheduler` maps the obtained switch port flow counters to

the existing path numbers, sorting them by ascending order and extracting the path assigned to the minimum value.

When getting the flow counters from MD-SAL, some issues were found with its refresh rate: when adding multiple flows in a short interval of time (e.g. 10 flows installed in less than 1 second), it was observed that the flow counters were not updated immediately after each flow was configured, consequently making the flow counters stay with the previous observed values (0 if they were the first flows to be installed, for example).

In order to tackle this problem and keep an approximation of the current installed flow counters, local counters were implemented in the **FlowManager** class. By saving the timestamp from the last MD-SAL flow counter query and configuring a variable that specifies the minimum interval of time between MD-SAL queries (**localFlowCounterDuration**, configured in the application main module configuration file), it was possible to have an approximation of the current flow counters. These counters were incremented when a new flow was installed and replaced by the values presented in the MD-SAL database every time a new query was made.

This solution did not solve entirely this flow counting problem, since when new flows are installed shortly before the validity of the local flow counters expire, those values will not be registered in the new flow counter query. Also, when using the local counters with this approach, it is not possible to keep track of flows that expired (and decremented the flow counters). Nonetheless, this solution allowed this flow scheduler to obtain better results, comparing to the original strategy.

#### 5.5.11.3 Random Flow Scheduler

The Random Flow Scheduler uses an instance of the **Random** class to generate random numbers. The **randInt** function limits the obtained values between 0 and the maximum number of available paths, returning a randomized next path number, used to establish the next path to be used.

This scheduler can be used by configuring the **schedulingStrategy** variable option of the application main module to **random**.

#### 5.5.11.4 Round-robin Flow Scheduler

The Round-robin Flow Scheduler is configured by setting the `schedulingStrategy` variable option of the application main module to `rr`.

The `nextRoundRobinPath` variable keeps track of the next path to be used by the scheduler. When a new packet arrives with the same source and destination address, the current value of this variable is used to specify the path where the flows will be assigned, following by an increment of 1 and a modulo operation with the number of existing paths. This last operation assures that the used path number never exceeds the number of existing ones, by setting its value to 0 after the last path is used, as it can be seen in figure 5.9.

#### 5.5.11.5 Static Flow Scheduler

The Static Flow Scheduler can be used by setting the `schedulingStrategy` variable option of the application main module to `static`. For every received packet and respective existing paths between the source and destination nodes, even if there is more than one available path, the first path from the path list is chosen, as it can be observed in figure 5.7.

This configuration can be changed by changing the `STATIC_PATH_NUMBER` variable value to a different number (predefined as 0).

### 5.5.12 Application Main Module

When Opendaylight is initialized, the Path Calculator application main module is also loaded. This module is responsible to retrieve all the application configuration options and obtain the required MD-SAL dependencies (Notification Provider Service, Data Broker, RPC Provider Registry, SAL Flow Service and Packet Processing Service), used by the other application components in order to communicate with MD-SAL and their required functionalities.

After all the configurations and dependencies are loaded, all the application components are initialized, which include the Network Graph Service, Host Manager, Flow Writer, Packet Dispatcher, Topology Change Handler, TCP/UDP Packet Handler, Path Calculator and Flow Scheduler. The only exceptions are

the Metrics Collector, the Address Tracker and the Packet Handler, which run independently from the main application module.

Concerning the Path Calculator and the Flow Scheduler, since a different class needs to be instantiated according to the chosen strategies, the `getPathCalculator` and `getFlowScheduler` methods are called, instantiating the adequate classes, by reading the `pathStrategy` and `schedulingStrategy` options from the application configuration file, respectively.

## 5.6 Summary

In this chapter it was described the specification of the SDN controller application, according to this thesis proposal. The tools that will be used in the used experimenting environment were listed, which included Opendaylight, the chosen SDN controller where the application will be developed, Open vSwitch, an OpenFlow capable software switch that can be instantiated in the nodes meant to support OpenFlow, following by CORE as the network emulator to be used.

A framework built for performing experiments in computer network scenarios was presented, based on the previously described tools. This framework allows the specification of network topologies and respective configurations through a Scenario Creator application that generates scenario files, used as input for the Experimenter application, responsible for creating the described topology in CORE and running all the configuration and network events additionally specified in the input scenario file.

The application specification was detailed, focusing in presenting the modules that will run in the application and respective roles, along with their work flow during the application's life cycle. This specification was then concluded by the contents of the application implementation, where its internal behaviour was outlined, as well as its most important configuration options.

# Chapter 6

## Evaluation results

In order to analyse the behaviour of the developed application and the impact of the implemented flow pinning strategies and path computation algorithms, it is necessary to conduct experiments that evaluate the performance of network connectivity by different hosts in a SDN based network.

This chapter is divided in two distinct sections. The first is oriented to the evaluation of the implemented flow schedulers, either when using a single-path or a multiple-path transport protocol. The second section aims to test the implemented path calculators when using MPTCP for performing multiple data transfers between endpoints in a network where multiple paths with different characteristics are available.

### 6.1 Transport protocol and flow scheduler evaluation

In the article by Bredel et al. [[Bredel et al., 2014](#)], the authors performed experimental work in a OpenFlow environment by testing different flow allocation strategies on a SDN controller, along with the usage of MPTCP.

The experiment consisted in performing multiple parallel file transfers between two endpoints, connected through a network formed by interlinked OpenFlow switches. These experiments showed that a strategy that takes in consideration the current status of the network (in this context, the number of flows assigned

to each path between the endpoints) achieves better results than one that does not use this type of input. Additionally, Bredel et al. accomplished improved outcomes when using MPTCP (comparing to TCP experiment results).

The setup of this environment was recreated in CORE and the same experiments were performed with the same flow pinning strategies (Minimum-flows, Round-robin, Hash-based and Random), following testing conditions identical to the original ones. These experiments had the goal of validating the implemented flow scheduling algorithms and comparing the obtained results when using MPTCP and TCP. As a reference, the original results were used as a baseline for comparing the obtained ones. The next sections will describe in detail the testing scenario and experiment results, respectively.

### 6.1.1 Testing scenario

The scenario where the experiments are held is based on the OpenFlow testbed presented in the article and its topology is illustrated in figure 6.1.

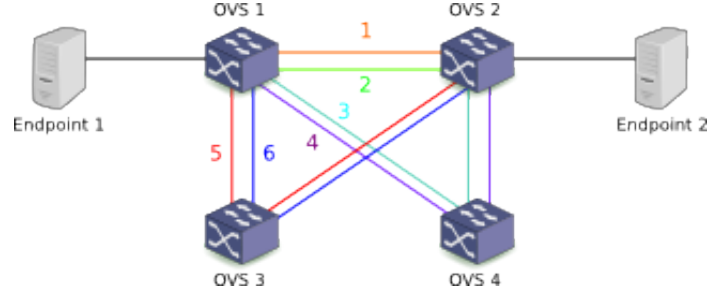


Figure 6.1: Topology used in the flow-based load balancing experiments

The topology contains two endpoints and four nodes running OVS. The endpoints are linked to the respective OVS nodes through a 100 Mbps link and the OVS nodes are connected through 10 Mbps links. There are a total of 6 link-disjoint possible paths between the endpoints (without considering the link that connects them to the network).

Considering the topology link delay, two variants of this scenario were considered. In the first one, every link had a fixed one-way delay of 2ms, making all paths having equal characteristics (besides hop count). In the second variant, a 200ms one-way delay was added to the links that formed paths 1 and 2, in order

to purposely increase the end-to-end latency in the topology shortest paths. Having these two versions of the testing scenario, it was possible to evaluate the flow scheduling strategies and transport protocols when having both homogeneous and heterogeneous available network paths.

The experiments measured the mean transfer time of multiple parallel file transfers (from 1 to 15) using the path selection strategies previously described in section 5.4.1.8 and compared the obtained results when using TCP and MPTCP. Each transfer consisted in sending 50 MB of data split between files with the respective size varying between 1 and 20 MB, generated by using a Zipf distribution with an exponent factor  $\alpha = 1.0$  (for achieving an even distribution for the values [Brakman et al., 1999]), having waiting times between each file sent defined by an exponential distribution with a rate parameter  $\lambda = 1.2$ . The cumulative distribution function (CDF) for the used values can be observed in figures 6.2 and 6.3, respectively. When using MPTCP, it is used a path-manager configuration that creates 3 sub-flows per connection, through the configuration of the `ndiffports` option.

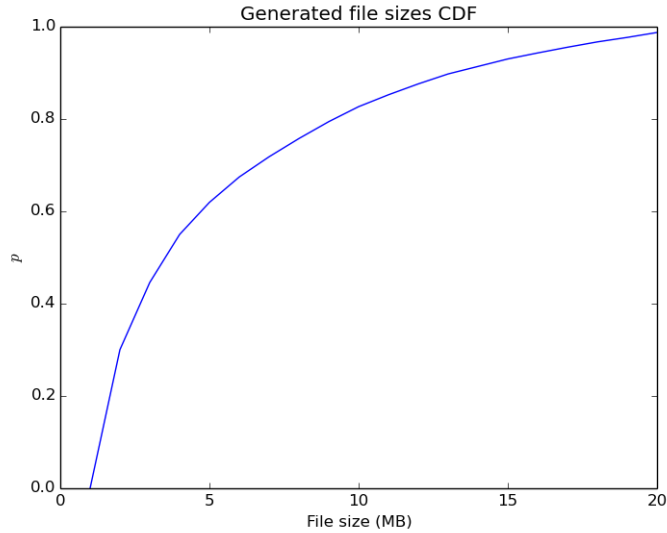


Figure 6.2: CDF of the used file sizes in the flow schedulers evaluation experiments

The evaluation of this scenario was primarily made without using an SDN controller. Since the topology was managed by the used testing framework, it was

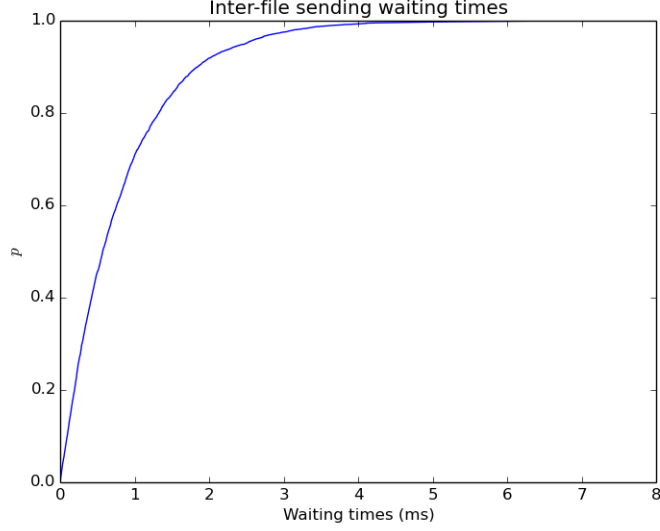


Figure 6.3: CDF of the used waiting times in the flow schedulers evaluation experiments

possible to know in advance the traffic that was going to be generated and create the matching flow rules in the OF switches locally (using management commands in OVS), according to the implemented flow schedulers and by having a static configuration of the existing disjoint paths. The obtained results were used as a baseline to compare the implementations to the ones in the original article with the chosen configurations (described in section 5.5.11), and later used to compare the same results when using the developed controller application.

However, this primary evaluation was made by using only TCP. Since MPTCP creates additional sub-flows with randomized source port numbers, it was not possible to predict the new created sub-flows without a controller that would parse the incoming packet data. In order to perform path computation in this scenario, the Link-disjoint path calculator algorithm was used, due to the fully-disjointness of the available paths in the tested topology.

Each test was run with 5 iterations that were used to generate the final average results.



### 6.1.2 Results

The results of the analysis of the different flow scheduling strategies are divided in two categories. The first presents the obtained results of the different schedulers when using TCP by statically installing the flows, without using the SDN controller. Finally, the same evaluation is presented by using the SDN controller application along with TCP and MPTCP. Complementary results can be found in Appendix A.

#### 6.1.2.1 Static flow allocation

##### Without additional path delay

Figure 6.4 presents the average transfer time for the evaluated scheduling strategies, when using TCP and by installing the flows prior to the traffic generation. The x-axis represents the number of parallel file transfers that was done and the y-axis the mean transfer time, in seconds. More detailed results, with the respective confidence intervals were included in Appendix A.

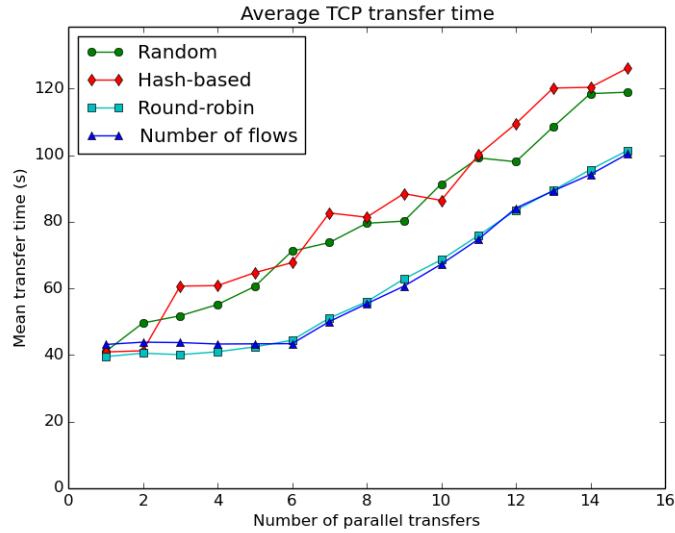


Figure 6.4: All scheduler evaluation results, without SDN controller

Comparing all the flow pinning strategies, it is visible that the Round-robin and the Minimum-flows approaches present the lowest mean transfer completion

times, with the second one reaching slightly lower results, comparing to Round Robin. It is also visible that the average transfer completion time begins to significantly increase after 6 parallel transfers, which is explained by the existence of only 6 link-disjoint paths. After having more parallel transfers than available paths, each path starts to become congested, explaining the growth of the transfer times.

### **With 200ms path delay**

Figure 6.5 presents the mean transfer completion time over the number of parallel transfers, with the same conditions as in the previously described test, but with a fixed delay of 200ms configured in the first 2 available paths (the shortest, considering a hop count metric). Overall, the general mean transfer time did not increase significantly, when comparing to the previous test without the increased path delay.

It is still possible to observe that the Hash-based and Random strategies achieve higher transfer times, comparing to Round-robin and the Minimum-flows strategies. However, it is visible that the Round-robin scheduler has higher transfer times than the Minimum flows. This is caused because the flows that are pinned to the paths with increased delay take more time to finish transferring the respective data, and Round-robin keeps assigning new flows to these paths, while the Minimum-flows scheduler is aware of the current flow counters, noticing the existent flows in the paths with delay, when that is the case.

#### **6.1.2.2 Controller application**

##### **Without additional path delay**

In figure 6.6 it is presented the results of the tests performed applying the Round-robin and Minimum-flows schedulers with TCP and MPTCP, when using the developed controller application, by presenting the mean transfer completion time in the y-axis against the number of parallel flows. Similarly to the results obtained with the static flow allocation, these schedulers have identical values when using TCP.

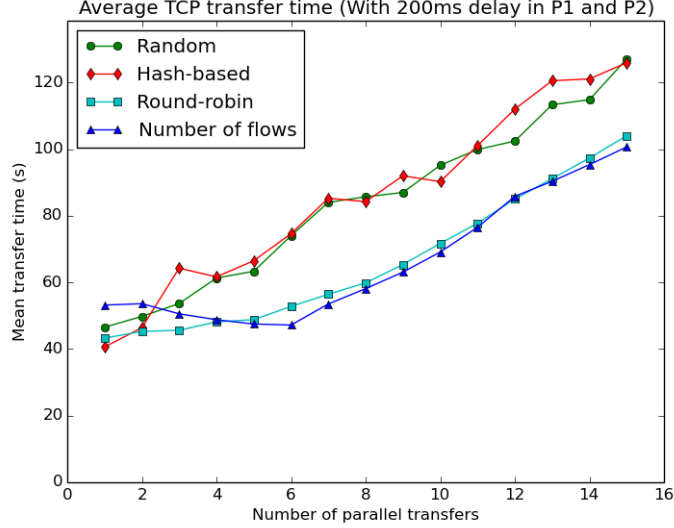


Figure 6.5: All scheduler evaluation results (with delay), without SDN controller

When using MPTCP, it is possible to notice significant lower transfer completion times when having a number of parallel transfers up to 8. This is explained by the saturation of the available network capacity. Additionally, it is observable that the average transfer time begins to increase after 2 parallel transfers are used. Since the current scenario configurations use 3 sub-flows per MPTCP connection, after 2 transfers the number of total created sub-flows is greater than 6, allocating more than one flow per path. In conclusion, when using MPTCP the outcome of these 2 flow pinning strategies was similar.

The mean transfer time over the number of parallel transfers when using the Random and Hash-based flow schedulers in the controller application with TCP and MPTCP is depicted in figure 6.7. Comparing to the results obtained from the tests without using the controller application, the Random scheduler also achieves the worst performance from all the remaining schedulers.

However, the Hash-based scheduler presents significantly better results in the tests using the controller application (e.g. when testing with 15 parallel transfers, the mean completion time in the tests without the controller application is 126s and with the controller application version the same result is 108s). This is explained by the fact that the implementation of both flow schedulers, despite

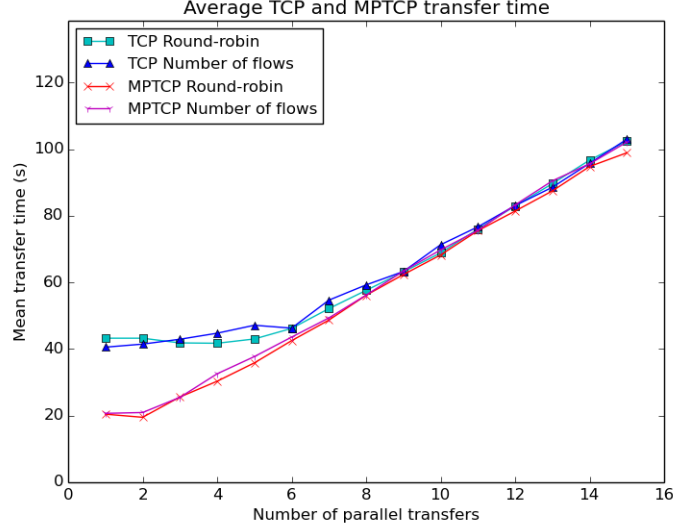


Figure 6.6: Round-robin and minimum flows evaluation results (TCP and MPTCP), using the SDN controller

using the same input fields in the hash function (refer to section 5.5.11.1), is made with different programming languages (Python and Java), having different built-in hash functions for **String** objects, resulting in different output results.

The usage of MPTCP results in diminished transfer times when using both schedulers, when comparing to TCP. Yet, similarly to the results with the Round-robin and Minimum-flows strategies, the obtained mean transfer times are close to each other, only varying by 3s in a worst case scenario.

### With 200ms path delay

When applying the 200ms delay in the shortest paths, the results obtained with TCP are identical to the ones without delay, as it can be observed in figures 6.8 and 6.9. However, the same does not happen when using MPTCP, as the respective results are notably greater when comparing to the first ones.

Both Round-robin and Minimum-flows schedulers achieve, in general, lower average transfer times with TCP, as MPTCP only outperforms TCP when having 1 to 3 parallel transfers. With the Hash-based and Random strategies, the performance of MPTCP is situated in the middle of the TCP Hash-based and

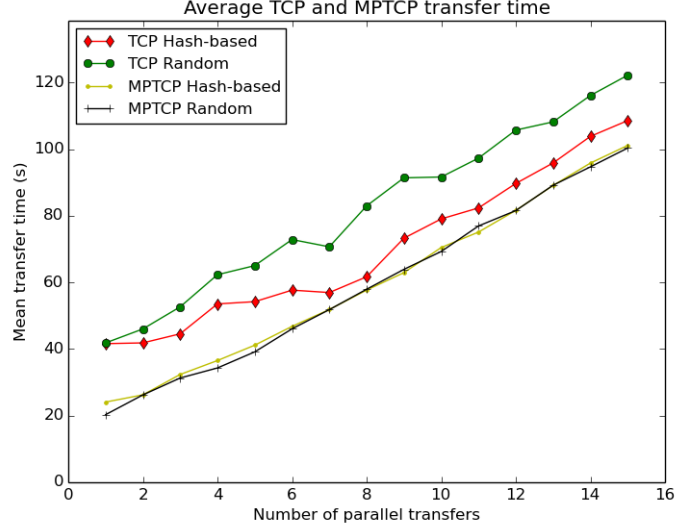


Figure 6.7: Hash-based and random, scheduler evaluation results (TCP and MPTCP), using the SDN controller

Random schedulers, having the best results in the Hash-based TCP tests, and the worst ones with Random TCP.

In conclusion, the mean transfer times were lower when using MPTCP in the tests performed without increased path delay, comparing to TCP. In the experiments with delay, it was observed that the performance of MPTCP significantly decreased, being in general, worse than TCP when considering the Hash-based, Round-robin and Minimum-flows strategies. However, this downgrade of performance was influenced by the lack of available network capacity in all the topology links. As the number of parallel transfers increased, the benefit of having multiple sub-flows for performing data transfer diminished. Additionally, when having paths with different delay configurations, the default packet scheduler used by MPTCP does not efficiently assign packets to the available paths, contributing to the increase of the application delay [Arzani et al., 2014; Grinnemo and Brunström, 2015].

Considering the different flow pinning strategies, the most significant differences in the obtained results were registered in the tests with TCP. The Random scheduler performed worst in the tests conducted with the controller, following up

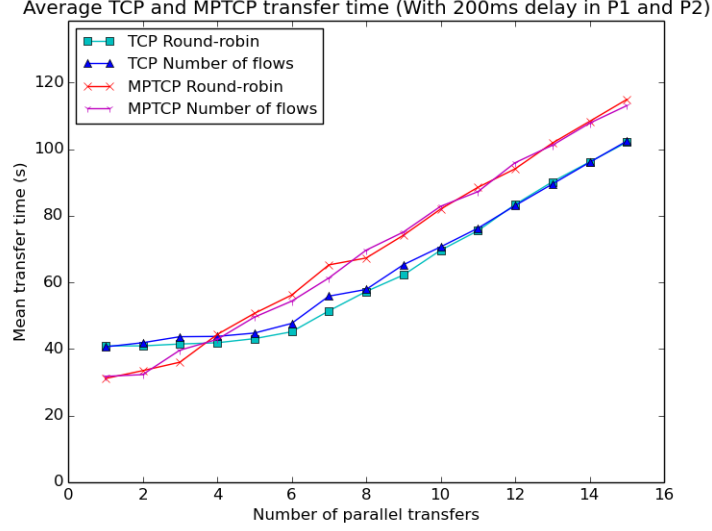


Figure 6.8: Round-robin and minimum flows evaluation results with delay (TCP and MPTCP), using the SDN controller

by the Hash-based flow scheduler. However, as it could be observed in the results with and without using the controller application, they can vary depending in the function used to calculate the corresponding hash path.

The Round-robin and Minimum-flows strategies were, in general, the ones with the best results. However, due to the flow counter monitoring restrictions explained in section 5.5.11.2, it is not always possible to have accurate values with the Minimum-flows approach. Consequently, this strategy did not outperform Round-robin, when comparing to the results presented in the original article by Bredel et al.

## 6.2 Path computation algorithms evaluation

Following up the previous experiments and given the different path computation algorithms implemented in the SDN controller application, a second evaluation phase engaged in validating, testing and analysing the behaviour of each algorithm.

Based in the results obtained in the last section, the experiments conducted

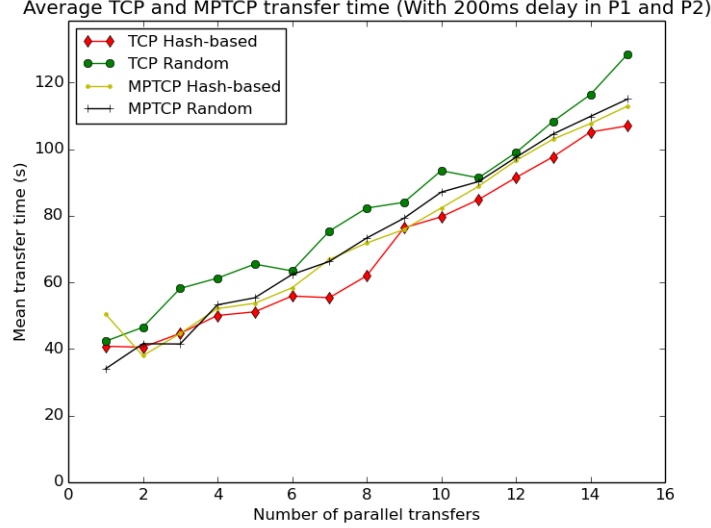


Figure 6.9: Hash-based and random, scheduler evaluation results with delay (TCP and MPTCP), using the SDN controller

in this section will be focused on the performance of MPTCP along with the combination of the usage of different flow pinning strategies with the implemented path computation algorithms. The results will be focused in presenting an evaluation of the quality of the established connections regarding network performance metrics.

### 6.2.1 Testing scenario

The topology used for performing this second evaluation phase is also formed by two endpoint nodes, connected through a network formed by multiple OF switches that provide a different number of paths between the two nodes, as it can be observed in figure 6.10.

The topology links connecting each node have different characteristics: the first-hop switches that connect the endpoint nodes (1 and 5) have 1 Gbps links, while the remaining have 100 Mbps. With the current topology configuration, it is possible to achieve a maximum throughput of 600 Mbps, following the possible paths formed by expanding the links from the 2, 9 and 16 nodes.

Concerning delay, there are 3 different configurations: The links that form a

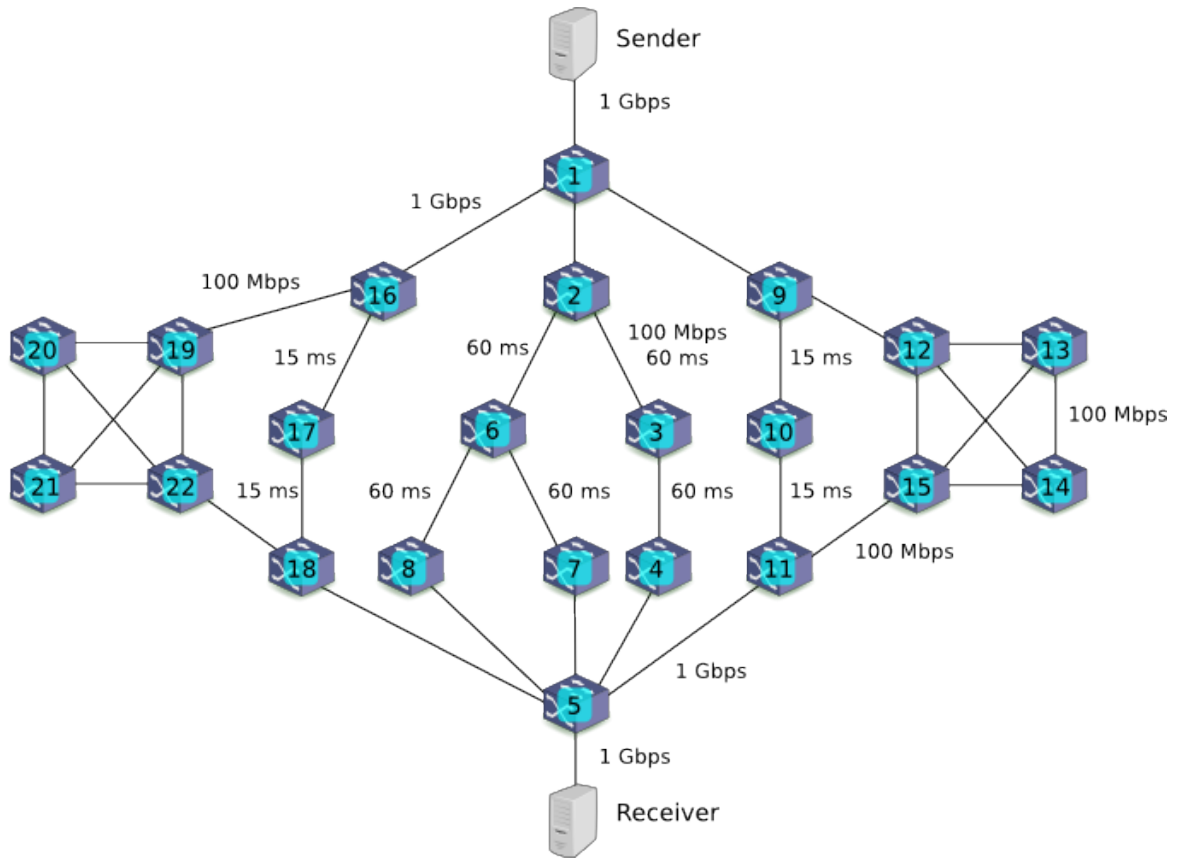


Figure 6.10: Topology used in the evaluation of path computation algorithms

path between node 2 and nodes 4, 7 and 8 have a one-way delay of 60ms, while the ones between nodes 9-10-11 and 16-17-18 have a 15ms delay. All the remaining network links have a default one-way delay of 2ms. Table 6.1 provides a detailed description of the characteristics of the topology links. The link latency values were selected as an assumption, in order to create 3 different types of paths, considering their end-to-end delay value.

These configurations create a scenario interesting for performing the evaluation of the different path computation strategies implemented in the developed SDN controller application, as through the existing different available paths, the shortest ones (considering a hop count metrics) are not the ones that have the least end-to-end delay. The evaluated path computation algorithms are the ones described in detail in 5.4.1.9 (Single path, Multiple paths, Link disjoint paths and SAMCRA).



Table 6.1: Characteristics of the link in the path computation algorithms evaluation topology

Node A	Node B	Bandwidth	Latency	Node A	Node B	Bandwidth	Latency
Sender	1	1 Gbps	2 ms	12	13	100 Mbps	2 ms
Receiver	5	1 Gbps	2 ms	12	14	100 Mbps	2 ms
1	9	1 Gbps	2 ms	12	15	100 Mbps	2 ms
1	2	1 Gbps	2 ms	13	15	100 Mbps	2 ms
1	16	1 Gbps	2 ms	13	14	100 Mbps	2 ms
2	3	100 Mbps	60 ms	14	15	100 Mbps	2 ms
2	6	100 Mbps	60 ms	16	17	100 Mbps	15 ms
3	4	100 Mbps	60 ms	17	18	100 Mbps	15 ms
4	5	1 Gbps	2 ms	18	5	1 Gbps	2 ms
6	7	100 Mbps	60 ms	16	19	100 Mbps	2 ms
7	5	1 Gbps	2 ms	18	22	100 Mbps	2 ms
6	8	100 Mbps	60 ms	19	20	100 Mbps	2 ms
8	5	1 Gbps	2 ms	19	21	100 Mbps	2 ms
9	12	100 Mbps	2 ms	19	22	100 Mbps	2 ms
9	10	100 Mbps	15 ms	20	22	100 Mbps	2 ms
10	11	100 Mbps	15 ms	20	21	100 Mbps	2 ms
11	15	100 Mbps	2 ms	21	22	100 Mbps	2 ms
11	5	1 Gbps	2 ms				

Considering the flow pinning strategies test results presented in section 6.1.2.2, the experiments were made by using the Round-robin and the Hash-based flow schedulers, as the respective outcomes presented greater stability, comparing to the remaining testing strategies. Besides being a solution aware of the OF switch flow configurations, the controller implementation limitations decreased the monitoring accuracy of the Minimum flows scheduler, comparing to the intended implementation results, making it not suitable for a further evaluation when using the different path computation algorithms.

The conducted experiments were based in the usage of MPTCP, focusing in maximizing the benefits of using multiple sub-flows for a single file transfer by splitting them across different network paths, comparing to the standard single flow approach as in TCP. Similarly to the previous experiments, each MPTCP connection generated 3 TCP sub-flows.

Identically to the experiments used to evaluate the flow scheduling strategies, these experiments also consisted in performing a different number of parallel file transfers between the sender and receiver node. However, the total amount of

data in these experiences was set to 500 MB due to the increase of the available bandwidth in the network links from this scenario, having the file sizes varying between 1 and 40 MB following a Zipf distribution, again with an exponent factor  $\alpha = 1.0$ , as it can be observed in figure 6.11. The waiting times before each file is sent also follow an exponential distribution with a rate parameter  $\lambda = 1.2$  (refer again to figure 6.3).

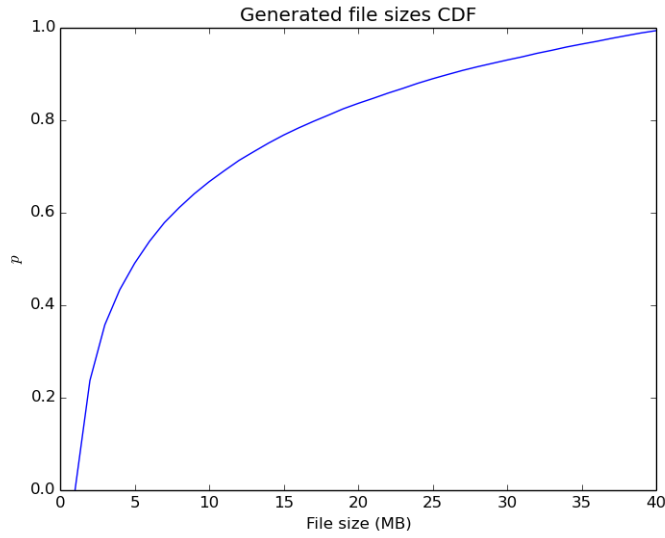


Figure 6.11: CDF of the used file sizes in the flow schedulers evaluation experiments

Regarding the parametrization of the configured algorithms, SAMCRA was configured to find at least 3 paths that would fill a 150ms delay constraint and a 5Mbps minimum available bandwidth constraint. Yen’s algorithm was configured to return a maximum number of 8 paths.

Additionally to the average transfer completed time, other factors were taken in consideration for specifying the evaluation of the used algorithms, such as:

**MPTCP Round-trip time** The Round-trip time of the connections created to transfer the files is used to perceive the end-to-end latency in the transport layer after the path calculation is performed. This metric was collected by using `mptcptrace` [Hesmans and Bonaventure, 2014], a tool that analyses

packet trace files generated by packet sniffing tools, such as `tcpdump` [Jacobson, Van and Leres, Craig and McCanne, S, 2010], and generates statistics from the MPTCP sub-flows, based in the packet header data.

**Throughput** This metric is presented as a complement to the average transfer completion time, as it can be obtained through the division of the size of the transferred files its respective completion time.

## 6.2.2 Results

The evaluation of the different path calculation algorithms is analysed by studying the variation of 3 different metrics: The average transfer completion time, the throughput of the completed file transfers, and the perceived round-trip time delay from the established MPTCP connections. The following sections describe each one of those parameters.

### 6.2.2.1 Transfer completion time

The obtained values for the average transfer time are depicted in figure 6.12. They are normalized, taking as baseline the values from the tests with SAMCRA, with Round-robin as the flow pinning scheduler, making all the other illustrated values the difference between the reference ones. Similarly as in the transport protocol and flow scheduler experiments, the detailed results with confidence intervals can be found in Appendix B.

The results obtained in the tests made with the single-path (Dijkstra’s algorithm) strategy were not presented in the figure, as they were vastly greater than all the other tests. Combining that with each test running time and the existing time constraints for this project, only the first 5 transfer tests were made with this algorithm. These values are displayed in table 6.2 instead.

Still, while analysing the values for this path computation algorithm, it can be observed that the mean transfer times oscillate without having a relation with a number of parallel transfers from the tests. Given the testing scenario topology, it is visible that there are a different number of shortest paths, considering a number of hops metric. However, these paths do not have the same delay characteristics

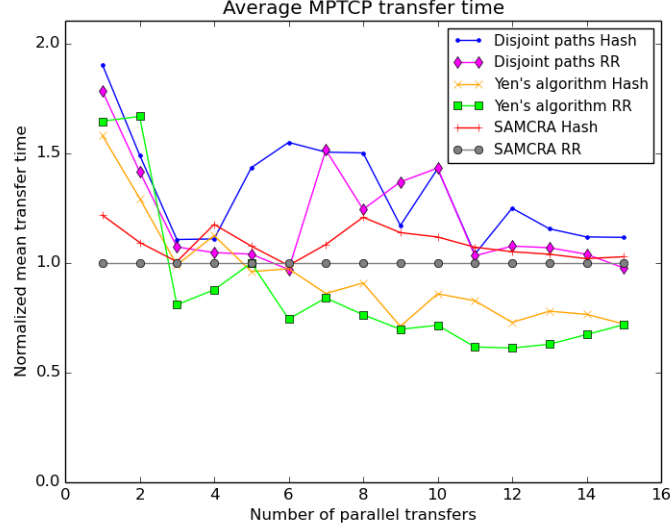


Figure 6.12: Average mean transfer times, normalized with SAMCRA (Round-robin)

(e.g. there are paths with a total of 10ms and paths with 124ms end-to-end delay), and the implemented Dijkstra's algorithm implementation can select different shortest paths from the existing mapped topology in Opendaylight, which can corresponding to paths with different delay characteristics, affecting the final values perceived during the file transfers conducted during each test. Since this algorithm only computes one path, the used flow scheduling approach does not influence the final results.

The Disjoint paths algorithm presents, in general, the third highest values, when comparing all the algorithms. Since this algorithm implementation computes fully link-disjoint paths by removing the links from the previously calculated paths, when combining it with the testing topology from this scenario, it is limited to a maximum number of 3 paths. When comparing the used flow scheduler, it is visible that the hash-based solution has worst performance, when comparing to the round-robin one, identically to the results obtained in the initial tests performed in section 6.1.

When comparing the path schedulers used with SAMCRA, Round-robin (the base values for the normalized function) reached lower transfer times, when con-

Table 6.2: Normalized average MPTCP transfer times using Dijkstra’s algorithm

Transfers	Round-robin	Hash-based
1	4.911	2.915
2	2.725	3.106
3	2.111	1.657
4	1.871	1.562
5	2.127	2.009

trasting the values with the Hash-based strategy.

For all tests made with more than 2 parallel transfers, Yen’s algorithm finished the tests with the minimum transfer times. Due to its implementation, it can calculate more paths than any of the previously described algorithms (limited by 8 paths in the current implementation), and consequently distribute new flows across a bigger number of different paths. Between the tests made using Round-robin and the Hash-based flow schedulers, the mean transfer completion time is lower in the Round-robin, due to its fairness when selecting the next path when a new flow is created.

Considering the average transfer time values for 1 and 2 parallel transfers with Yen’s algorithm, it can be concluded that during these tests the number of existing flows was still short (comparing to the tests with a greater of parallel transfers). Despite having a larger number of available paths, the ones selected by SAMCRA had lower delay, while the paths computed by Yen’s had different delay configurations. Hence, since the traffic demand for these scenarios was still not vast enough to create congestion in the different used paths, the usage of the paths with lower end-to-end delay values contributed to achieve lower transfer completion times.

### 6.2.2.2 Round-trip time delay

Equivalently to the results shown for the obtained throughput, the RTT delay was illustrated in 3 plots, corresponding to the value obtained with 1, 8 and 15 parallel file transfers.

The CDF distribution of the RTT delay for the experiments with 1 parallel transfer is displayed in figure 6.13. Analysing individually the results from Dijkstra's, and based in the preconfigured delay in the topology from the evaluation scenario, it is visible that for the experiments made with Round-robin (though the flow scheduler had no influence in the results), the algorithm selected one of the shortest paths with the greatest end-to-end delay (1-2-3-4-5, 1-2-6-7-5 or 1-2-6-7-5). On the other hand, in the tests with the Hash-based scheduler, in 3 out of 5 iterations of the test one of the paths with 34ms end-to-end delay (1-9-10-11-5 or 1-16-17-18-5) was used and in the remaining 2 the same paths were used as in the Round-robin experiments.

For the tests with the Disjoint path algorithms it can be observed that, independently of the flow scheduler, the majority of the packets were sent through the 34ms paths, having some packets using one of the largest delay paths. Yen's algorithm used a diversity of paths with all the delays (including the paths with the lowest delay, e.g 10ms), while SAMCRA calculated mostly paths with the lowest delay.

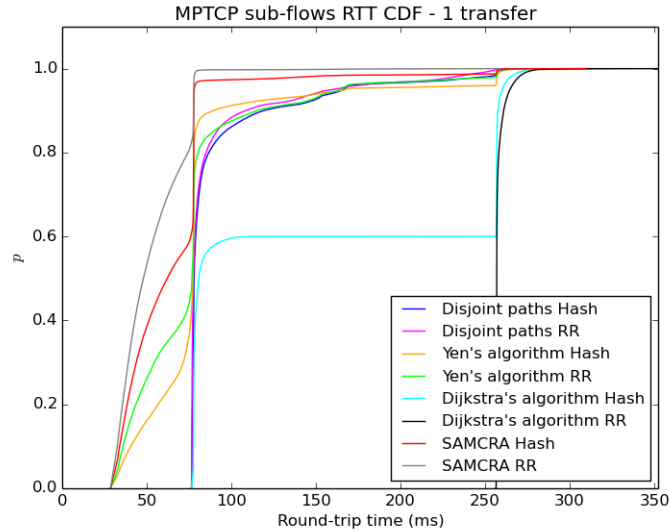


Figure 6.13: CDF for RTT delay with 1 parallel transfer

The RTT values obtained during the tests executed with 8 parallel transfers are depicted in figure 6.14. Though the results look similar when comparing Yen's

algorithm and SAMCRA, it is noticeable a bigger difference when considering packet delays from approximately 175ms to 200ms, consequence of the effort of SAMCRA from using the measured link delay as a metric for calculating paths and returning paths with the smallest available delay.

The Disjoint path algorithm presents the highest delay values, an outcome from always using the same 3 paths for all the new installed flow rules, one of them with the highest end-to-end path delay, configured in the scenario topology.

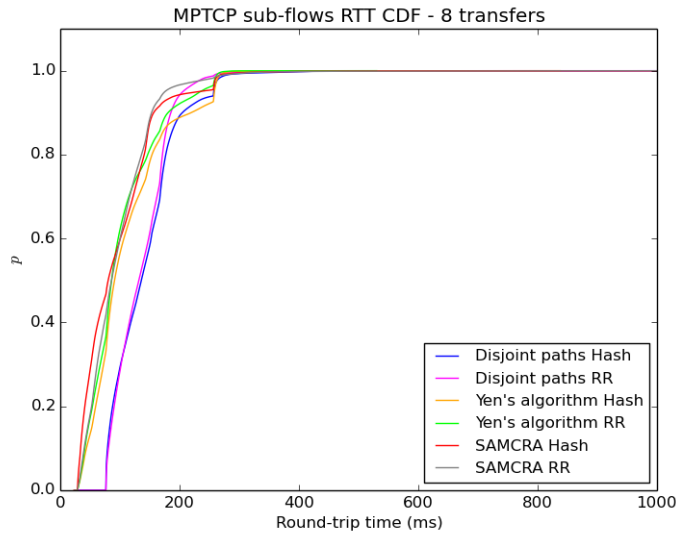


Figure 6.14: CDF for RTT delay with 8 parallel transfers

Finally, the delay values for the experiments with 15 parallel transfers are represented in the CDF distribution chart presented in figure 6.15. Despite the high maximum delay values (more than 1 second), it is observable that when considering the used path calculation algorithm, the obtained values are similar for both flow pinning solutions (Round-robin and Hash-based).

The Disjoint paths algorithm achieved the largest delay values, with nearly all packets having at least a RTT delay close to 100ms, following up by greater values, when comparing to the other algorithms. In general, it is visible a curve near the 240ms mark in the x-axis, corresponding to the delay from the packets routed through the paths that had a pre-configured 124ms end-to-way delay. However, the percentage of packets that match these values is small in the results

from SAMCRA, since it was programmed to always find paths with the least delay possible, despite its constraints, when comparing to Yen's algorithm, for example.

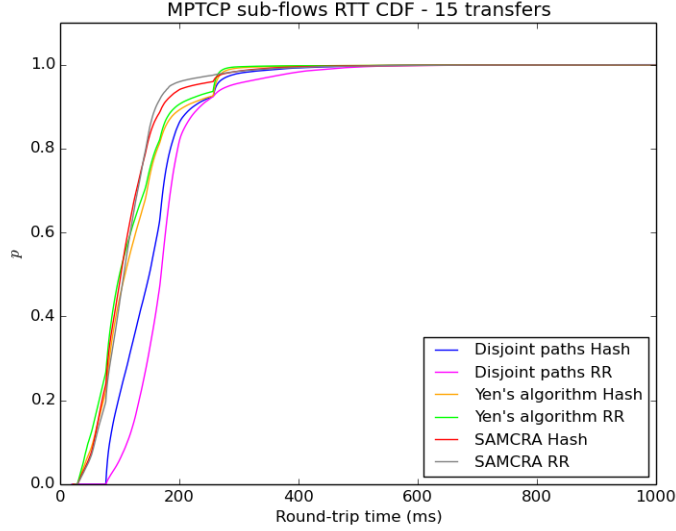


Figure 6.15: CDF for RTT delay with 15 parallel transfers

### 6.2.2.3 Throughput

The throughput from the file transfers performed during each test was plotted in different CDF charts and presented in this section, for a number of 1 (minimum), 8 (median) and 15 (maximum) parallel transfers.

The values for the tests made with 1 parallel transfer are illustrated in figure 6.16. While interpreting the results from Dijkstra's algorithm, and based in the RTT delay measurements described in the previous section, the low throughput values from the Round-robin tests are justified by the usage of paths with the highest delay, when comparing to the ones obtained in the tests with the Hash-based scheduler.

Following up the remaining algorithms, it is observable that in general, SAMCRA achieved the highest throughput, following up by Yen's and the Disjoint paths algorithms. Considering the flow schedulers used with SAMCRA and the Disjoint path algorithms, the tests with Round-robin achieved the highest



throughput while contrasting it with the Hash-based solution. However, when using Yen's algorithm, the Hashed flow scheduler had slightly better results than Round-robin. This can be explained by the existence of multiple paths with different delay configurations and, while Round-robin always sets up the next flow to the next available path (even if it is one of the paths with the highest end-to-end delay), while with the hashed paths can be assigned to paths with a lower delay, skipping the high delay paths.

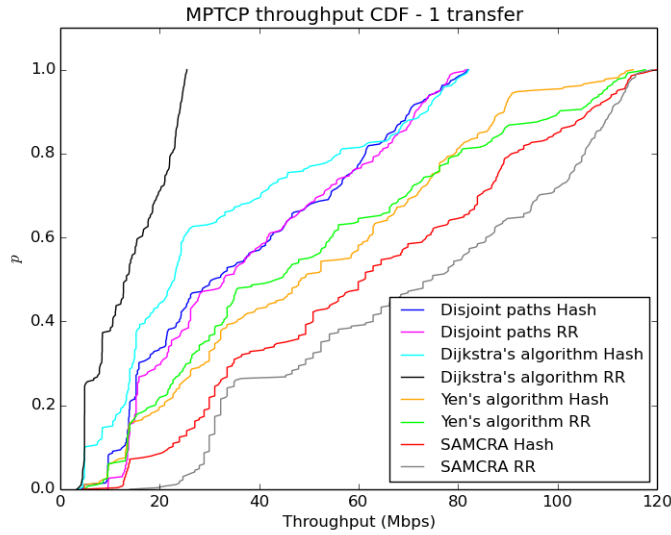


Figure 6.16: CDF for throughput with 1 parallel transfer

Figure 6.17 pictures the CDF distribution of the throughput measured during the tests with 8 parallel transfers. Contrary to the tests with 1 parallel transfer, Yen's algorithm achieves higher transfer rates than SAMCRA. Due to the existing number of sub-flows (24, corresponding 3 MPTCP sub-flows for each one of the 8 transfers), in order to achieve higher throughput measurements, it was more beneficial to use a wider number of different paths (even if with different characteristics). SAMCRA used a smaller number of paths with lower delay configurations, as it would only find a maximum number of 3 paths.

For this scenario, the Round-robin scheduler strategy achieved higher throughput measurements during the tests with Yen's algorithm, which showed that a more stabilized load-balancing approach was a more beneficial solution in a sce-

nario with a high traffic demand.

Following up the remaining algorithms with respective flow pinning strategies, the obtained results are similar as the ones in the previous tests, having the Disjoint paths algorithm completing the tests with lower transfer rates due to the flow pinning being made on only 3 paths (with the Hash-based scheduler having a worse behaviour than Round-robin), followed up by SAMCRA (again, with Round-robin achieving the best results).

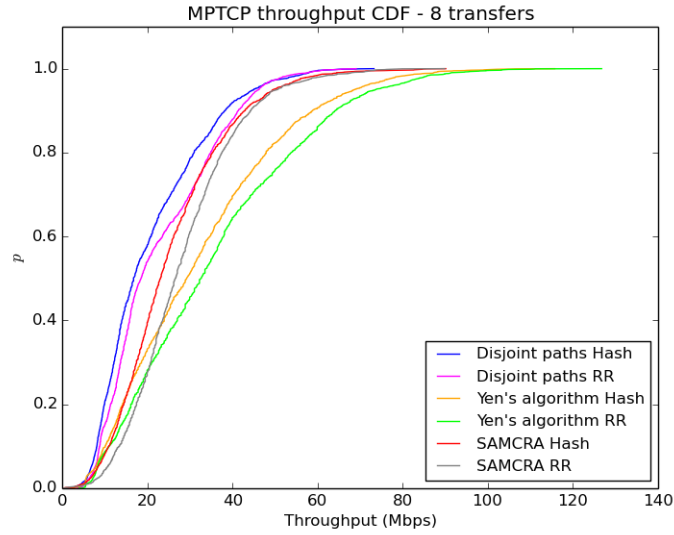


Figure 6.17: CDF for throughput with 8 parallel transfers

When analysing the throughput for the tests with 15 transfers, as presented in figure 6.18, it is visible that Yen's algorithm achieves the highest throughput values. When comparing the Disjoint paths and SAMCRA results, the difference between these two algorithms is not very significant. While for the Disjoint paths algorithm the obtained values can be easily explained by the limitation of using only a maximum number of 3 paths for all the installed flows in this scenario, the SAMCRA algorithm has a more complex behaviour. For these experiments, a 150ms end-to-end path delay constraint and a 5 Mbps available bandwidth constraint were configured in the algorithm, but, due to the high traffic demand, it was not always possible to discover paths that would fulfil these conditions, forcing the algorithm to relax the constraints and find paths with worse link

quality metrics. Additionally, in order to force the algorithm to find a different number of paths, the removal of the links present in the previous calculated paths contributed to obtain a limited number of solutions (in a similar manner as the Disjoint paths algorithm behaved), and consequently, limited the maximum throughput values achieved for each file transfer.

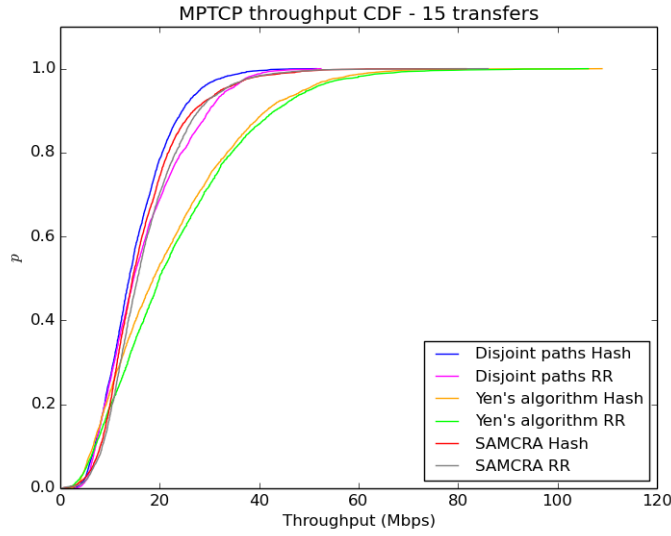


Figure 6.18: CDF for throughput with 15 parallel transfers

## 6.3 Summary

This chapter presented the results for all the experiments conducted, in order to evaluate the developed OpenFlow controller application.

Initially, the outcomes for tests using different transport protocols (TCP and MPTCP) showed that the usage of MPTCP contributed to lower transfer time values in almost every scenario, when comparing to TCP. However, when having a high traffic demand in a topology with different delay characteristics along the existing paths, with some flow pinning strategies (Round-robin and Minimum flows), TCP obtained better results than MPTCP, as the default packet scheduler used in the current MPTCP implementation performance decreases significantly in scenarios with these characteristics.

When analysing the different path scheduling strategies, it was showed that strategies that would guarantee fairness in the resulting solutions produced better results when comparing to strategies that would not take in consideration where the previous flows were allocated. Ideally, the Minimum flows scheduler promised better results, as it would assign new flows to the path with least installed flows (despite not taking in consideration other factors, such as the bandwidth utilization for each existing flow), but due to restrictions in monitoring the active flow counters in Opendaylight, it was not possible to use this flow pinning strategy with accurate values. Comparing the remaining tested strategies, the results achieved with Round-robin were the second best, following up by the Hash-based and the Random schedulers. In addition, the tests made with MPTCP showed that due to its coupled congestion control, the usage of a different flow scheduler did not have a large impact in the final results.

In the tests made with the different path computation algorithms, it was visible that using multiple paths for assigning new flows was more beneficial than using a single path, as it could be seen by the results obtained by using the single path Dijkstra's algorithm, while comparing it to the other evaluated solutions. The link disjoint algorithm presented paths with fully link-disjointness, but it was possible to see that in scenarios where complete disjointness cannot be achieved, such as the one with the evaluated topology, all the flows were distributed across a smaller number of paths, resulting in a lower achieved throughput values or higher RTT delays.

Considering the average transfer completion times and achieved throughput, Yen's algorithm presented the best results, as it would distribute the incoming new flows across a wide number of paths. However, when evaluating the measured RTT delay, SAMCRA presented lower values, as it would always compute paths using the link delay as one of the algorithm metrics, making it the most promising solution for scenarios where new flows need to be assigned to paths with low end-to-end delay.

# Chapter 7

## Additional Contributions

Along with the work performed directly for the main project of this thesis, other tasks were held by the author, which were not initially planned in the proposal. This chapter briefly describes the main outcomes reached for each one of these tasks.

### 7.1 Opendaylight OVSDb REST client

Opendaylight allows the configuration of managed OF devices that support the OVSDb protocol, typically software switches running Open vSwitch, through a REST API.

Having as main goal the development of a tool for testing different QoS configurations in OVS, a client for the OVSDb protocol was made, providing a GUI that implemented different features, which included:

- Connecting to new switches (from the controller);
- Managing (adding, editing or removing) bridges from a switch;
- Managing switch ports;
- Managing QoS configurations associated with existing ports, and respective queues. This feature added support for novel QoS schedulers in OVS, such as Stochastic Fairness Queueing (SFQ) or Fair Queueing with Controller

Delay (FQ Codel), which are not present in its current distribution version [Vestin and Kassler, 2015].

A screen capture of the application's GUI can be observed in figure 7.1.

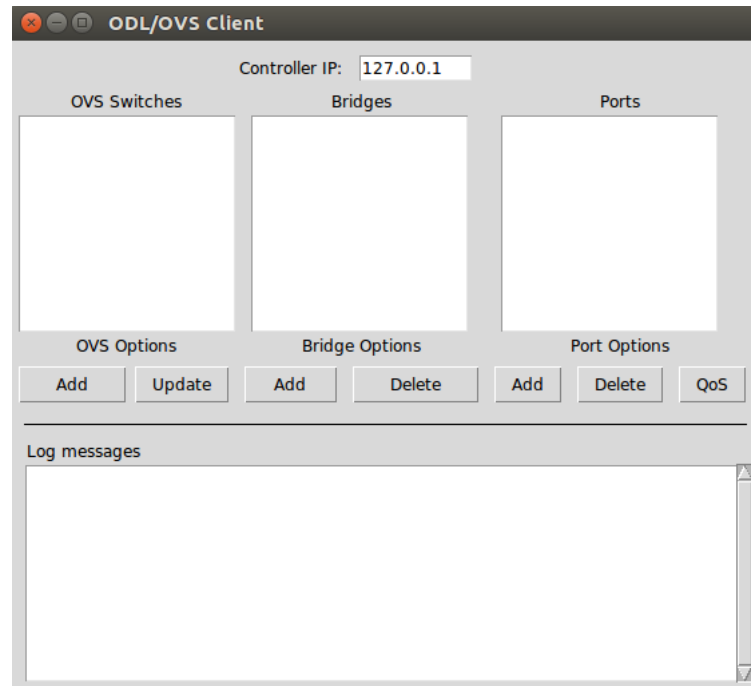


Figure 7.1: Example of the GUI of the Opendaylight OVSDDB REST client

## 7.2 Multiflow

Multiflow was a joint project between researchers from the University of Coimbra, Instituto Pedro Nunes and PT Inovação. Its main goal was the improvement of the gains obtained from connections in Wi-Fi hotspots, based in an efficient management of the existing and created flows between the users of the provided services, through the usage of SDN capabilities.

### 7.2.1 Used architecture

An illustration of the architecture used in the Multiflow project is depicted in figure 7.2.

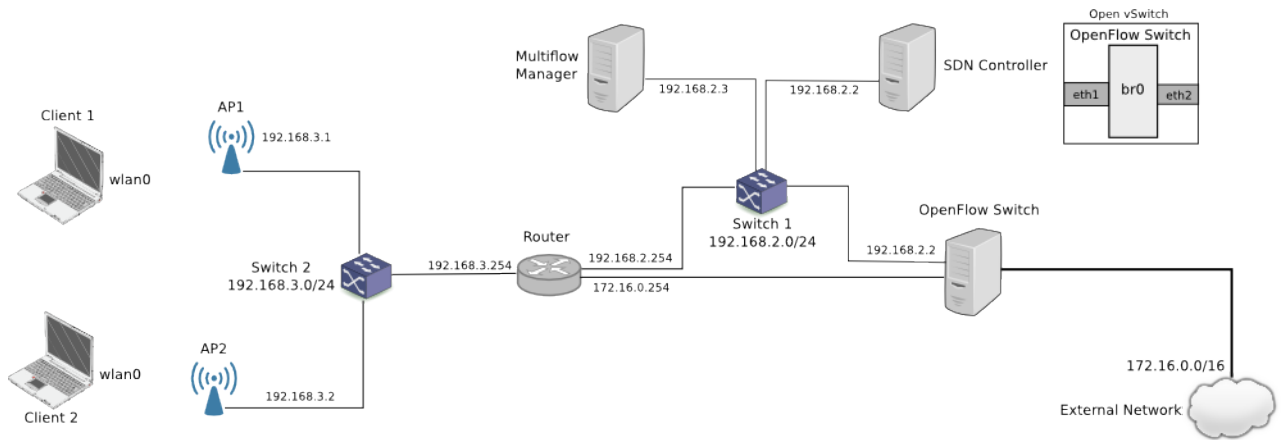


Figure 7.2: Multiflow architecture

The architecture was composed by different components, each with different roles:

**Clients** Mobile clients that would connect and use the network as a Wi-Fi hotspot, to communicate with an external network. Each client could be classified as *normal* or *premium*. *Normal* clients would use available network resources without benefiting from QoS guarantees, sharing the same QoS configurations with all the other clients from the same type. *Premium* clients would benefit from connectivity privileges, configured by using individual QoS queues with minimum assured bandwidth values;

**Access Points** Providers of network access to the clients. Internally they would provide statistics related to the managed network (e.g. number of received/transmitted bytes and packets) through the SNMP protocol;

**Router** Forwards packets between the external and internal networks. Runs a DHCP server for providing IP addresses to the connected clients;

**SDN Controller** Instance of Opendaylight, responsible for the connectivity to the OpenFlow Switch. Runs a REST API that is used for receiving commands related to the installation and configuration of flow rules and QoS

queues. Communicates with the OpenFlow switch through its OVSDB Southbound interface for managing the QoS configurations and uses its OpenFlow Southbound interface for managing flow rules;

**Multiflow Manager** Responsible for the management of the services offered to the Wi-Fi hotspot clients. Manages the authentication of *premium* clients and configuration of its respective QoS queues and installation of flow rules, through calls to the SDN controller's REST API. Collects QoS metrics from the network, through the monitoring of network statistics from the used access points, present in the network. With the collected metrics, it runs periodically an algorithm that indicates *premium* clients the access point with the most available resources, while considering the bandwidth utilization of each client wireless network interface at the moment;

**OpenFlow Switch** Interconnects the external network to the router present in the internal network. Runs Open vSwitch, which is configured through the OVSDB protocol, in order to manage the existing QoS configurations and respective queues;

**External Network** Used to represent an Internet connection to the inside network. It was used to run traffic generator servers that would receive connections from the clients and register the perceived connection quality values, such as throughput, delay and packet loss.

### 7.2.2 Obtained results

In order to evaluate the quality of the connections established by the clients, different tests were made, which were focussed in the measurement of the perceived throughput, delay and packet loss by the clients through the usage of traffic generators. This section presents some of the main obtained results regarding the measured end-to-end delay and packet loss percentage, when varying the number of *premium* clients in the network.

For each test, 2 clients were used. In a first client, a single connection representing a VoIP call was made using the Distributed Internet Traffic Generator (D-ITG), where in the second client, a different number of connections using the



Netperf traffic generator was made along different tests, in order to consume the most available network resources.

The results when having 2 *regular* clients connected to the network (without any *premium* clients connected) can be observed in figures 7.3a and 7.3b. When considering the measured delay, high values were always perceived in the first client, for all the tests with a different number of connections in the second client (between approximately 1100ms and 1200ms), while the packet loss percentage varied between 1.6% and 3.0%.

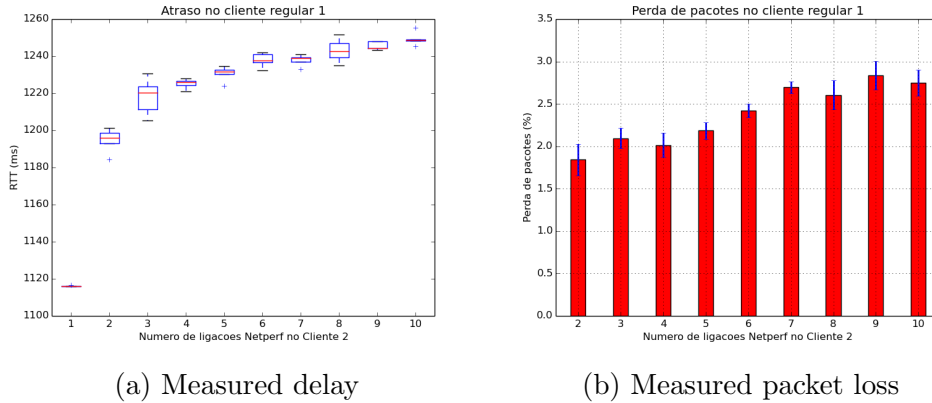


Figure 7.3: Multiflow results when not using any premium clients

When using one *premium* client connected to the network, the results were significantly better, as it can be seen in figures 7.4a and 7.4b. For all the tests made, the delay had values approximately between 4.8ms and 6ms, while no lost packets were perceived.

When having two *premium* clients connected in the network, the average delay was greater than when having only one *premium* client (reaching a maximum value of approximately 15ms, which is still acceptable for VoIP calls [ITU-T, 2003]), but still extremely lower than the first scenario, where there were not any *premium* clients. The packet loss percentage was also lower in the first client, varying between 0.3% and 0.6%. These values are depicted in figures 7.5a and 7.5b, respectively.

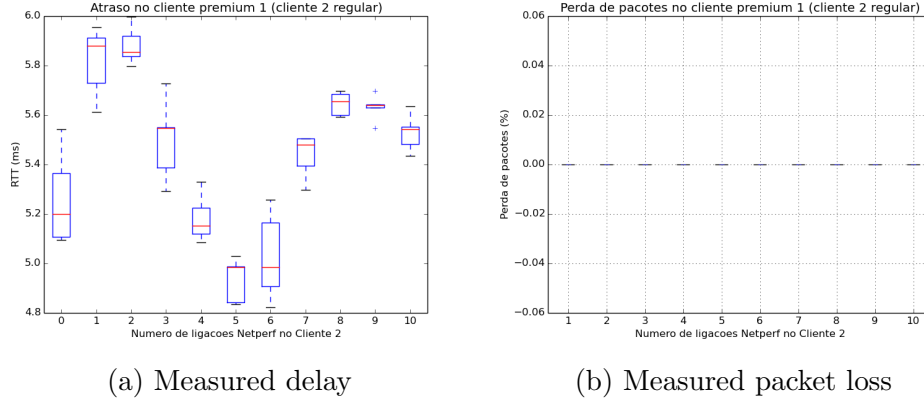


Figure 7.4: Multiflow results when using one premium client

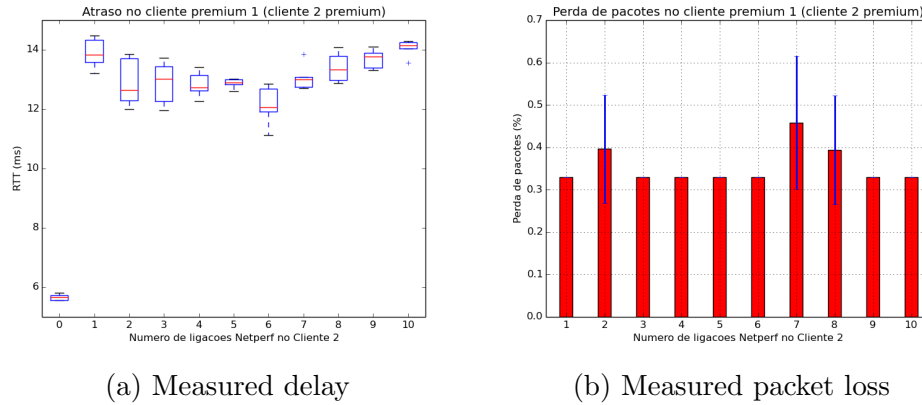


Figure 7.5: Multiflow results when using two premium clients

### 7.3 Opendaylight workshop

An Opendaylight application development workshop was organized and attended by several members of Karlstad's University faculty and a visiting researcher from the Technical University of Denmark. The workshop had as main goal the explanation of the initial steps required to develop customized modules for ODL in a step-by-step tutorial approach. Therefore, during the workshop different topics related to Opendaylight were covered:

**Creation of new ODL modules** The procedures necessary for creating and using new modules in Opendaylight were covered, along with creating a new project from a Maven archetype and configure the required dependencies.

**Specification of YANG models** An overview of YANG models was made, which included the specification of the data types used by YANG and its respective Java mapping, the generation of code from a YANG model by using the plugins provided in Opendaylight, and building and using objects from the created data structures in the internal application code.

**MD-SAL data storage usage** The basic concepts of Opendaylight's database storage were covered, consisting in creating and using instance identifier objects in order to retrieve or add storage data. Practical examples were made, combining the data structures previously created by a YANG model.

The workshop took place in Karlstad's University and lasted two days. The first day covered the creation of new ODL modules and the remaining tasks were held during the second day. By the end of the workshop, a working copy of the created and used Opendaylight distribution code was shared to all the workshop participants.

## 7.4 Summary

This chapter described the additional tasks held by the author in parallel with the main topics described in this document.

While the performed work did not contribute directly to the final obtained results, they were still related to its main covered research fields, such as Software-Defined Networking and the development of applications in Opendaylight, resulting in a complementary increase of the achieved experience from the related research topic fields.



# Chapter 8

## Project Management

In this chapter it is presented the planning of the work for both first and second semesters, regarding the Master Thesis described in this document.

In section [8.1](#) it is presented the work plan for the first semester and in section [8.2](#) it is described the planning of the work performed during the second semester.

### 8.1 First Semester

The work performed during the first semester focused initially on studying the state-of-the-art regarding the concepts related to the practical work to be done, following by an initial specification of the controller application to be implemented and its respective implementation.

However, due to delays on some of the study of the state-of-the-art concepts and some problems concerning the implementation of new controller modules in Opendaylight, the original work plan suffered some modifications, as it can be seen in figure [8.1](#).

A more detailed description of each of the tasks completed during this semester is presented next:

- **Task 1 - Study of the state-of-the-art on SDN (22/09/2014 - 06/10/2014):** The basic SDN architecture concepts were described, following up by research work regarding the existing surveys and applications

used in this environment. This task was concluded with a study about the existing SDN controllers and the features that distinguished each of them.

The greatest difficulties during this task was finding proper documentation about the controllers and determining if some of the analysed controllers were still with an active development and with an user community;

- **Task 2 - Study of the state-of-the-art on transport protocols**

**(29/09/2014 - 20/10/2014):** For this task, initially it was made a study of the existing and most used single-path transport protocols, which included UDP, TCP (and some of its different congestion control mechanisms) and DCCP.

The second phase of this task involved a more detailed study regarding multi-homed transport protocols (SCTP and MPTCP), where they are currently used, and their differences, advantages and drawbacks comparing to previously studied single-homed protocols;

- **Task 3 - Study of the state-of-the-art on path computation algorithms**

**(06/10/2014 - 03/11/2014):** This task was the one that suffered the greatest delay, concerning the original work plan. Due to the complexity of this field and the abundant number of existing of proposed algorithms, a greater effort had to be done when studying the state-of-the-art of this field, specially when analysing the scientific papers related to solving problems with multiple constraints, because of the original complexity of the type of problem;

- **Task 4 - Specification of the first version of the routing approaches to be developed**

**(03/11/2014 - 17/11/2014):** Based on the study performed in the previous tasks, an initial version of the specification of the application to be developed was made. This involved the selection of Opendaylight as the SDN controller for building the application, and the path calculation algorithms to be used in the different routing strategies (Dijkstra's for the single-path routing, Yen's algorithm for the multiple-path routing and SAMCRA for the constrained multiple-path routing problem).

Additionally, the construction of a baseline testing topology for further results comparison was defined, regarding the experimentation of different flow-load balancing approaches;

- **Task 5 - Implementation of the first version of the controller application (17/11/2014 - 27/01/2015):** For this task, initially it was required to learn and understand the internal architecture of Opendaylight. Since this controller is under development, between the previous official release and the latest one, the developers changed a significant amount of features and steps required to build an internal module, due to the adoption of the Karaf OSGi-based framework.

However, the existing documentation, by the time the intermediate thesis report was written, did not cover these new required steps that followed the new version, increasing the effort required to understand how to successfully deploy customized modules;

- **Task 6 - Writing of the first semester report (29/09/2014 - 27/01/2015):** The intermediate report was written from the beginning of this semester, as the state-of-the-art tasks began. This was followed up by describing the steps taken in the specification of the application to develop and in the tools and frameworks required to build and run a testing platform used to evaluate the behaviour and performance of the application.

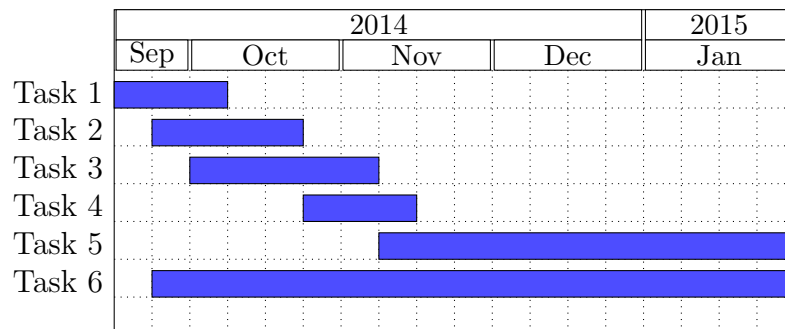


Figure 8.1: First semester work plan

## 8.2 Second Semester

During the second semester the final part of the work of this Master thesis was done. This work was performed concerning more practical aspects of the implementation and evaluation of the SDN controller application, followed by an analytical evaluation of the obtained results. A diagram presenting how this work was performed can be seen in figure 8.2 and the illustrated tasks are presented next:

- **Task 1 - Implementation of the first version of the controller application (09/02/2015 - 02/04/2015):** This task consisted in finishing the ongoing work from the first semester. This included finishing creating a service to be run in Opendaylight, which needed the domain of the required dependencies and steps for integrating the created modules inside the application. Following the creation of the service, it was necessary to instantiate and access data and methods from other modules (e.g. Topology Manager), which required accessing the MD-SAL data storage system. By having the knowledge required for accessing other modules, one of the biggest obstacles was passed, facilitating the remaining work to be done when implementing applications.

The implementation continued by creating the main application modules that were responsible for obtaining the topology details from the network, installing new flow rules and keeping in storage the existing endpoint address data, following by the implementation of the path computation algorithms previously described.

The created application was later tested on the previously created topology, in order to analyse its behaviour when adding new flows based on a computed path;

- **Task 2 - Specification of the final version of the controller application (02/04/2015 - 16-04-2015):** The specification of the final version of the controller application defined the flow allocation strategies that were used, for each of the multiple path routing strategies when multiple flow rules needed to be installed. In addition, with the intent of specifying the



behaviour of the Metrics Collector module, a set of network performance metrics were defined, including how they were collected and the frequency of the data collecting;

- **Task 3 - Implementation of the final version of the controller application (16/04/2015 - 20/05/2015):** This final version of the controller application consisted in adding the features referred in the previous task, to the first version of the application. Hence, firstly problems regarding the last version were fixed, following by the implementation of the functionality that allowed the previously specified flow allocation strategies to be used in the Path Calculator module.

The Metrics Collector module was created, along with the methods required to successfully retrieve the metrics specified in the previous task;

- **Task 4 - Evaluation of the final version of the controller application (30/04/2015 - 15/06/2015):** This evaluation phase initially specified the scenarios where the controller application was tested. This specification includes the topology that was used, the duration of the experiments, the traffic demand (which included aspects as the number of flows to be created, the size and rate of the packets to be sent, the total amount of data to transfer and the interval between data transfers).

The evaluation was followed by running the application along with the traffic generators and analysing of the obtained results. This analysis contributed to understand aspects as the importance of the used collected metrics in the algorithms' decisions and their impact in the application behaviour, the time interval that was used for data collecting, the application's performance over the testing time when modifying the parameters previously marked as evaluation criteria and, most importantly, the quality of the obtained results in the traffic generator receiver log files.

The results obtained from the previously built flow load-balancing application will also be compared with the developed application, in order to verify if improvements were made with the new path routing approaches;

- **Task 5 - Writing of the final report (11/05/2015 - 03/07/2015):**

This final report included a fully-detailed version of all the steps that were required to complete all the previous tasks, presenting the final conclusions obtained through the year, during the execution of this thesis.

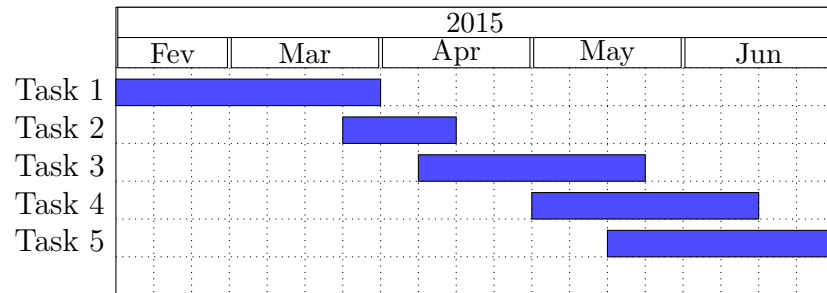


Figure 8.2: Second semester work plan

# Chapter 9

## Final Considerations

This document approached the concepts of Software-Defined Networking as a new networking paradigm, where the control plane is separated from the forwarding plane, having the first one present on a single (or more, if it is a distributed solution) controller, and the second one on switches managed by the controller. Relevant ongoing research work in this field was analysed, including existing SDN controllers and their included features. One of the most relevant controllers is Opendaylight, an open source controller with a significant developer and user community, built over a structured service-based architecture.

The increase of the number of network interfaces in the existing devices contributed significantly to the exploration of the usage of transport protocols that are aware of more than one interface when establishing and transmitting data over a connection. In this document, through the analysis of MPTCP, it was shown that in a scenario where multiple network paths are available, the throughput of a single connection can greatly improve when having multiple sub-flows over a MPTCP connection.

When it comes to path calculation in a network, there are multiple variants of the problem to be solved. One of the most simple and most studied is the shortest single-path problem. Between the many existing alternatives to solve this problem, Dijkstra's remains one of the most simpler and efficient algorithms. The same problem can exist, but with the addition of constraints that need to be respected in order to obtain a valid solution. The constrained shortest path problem brings a higher type of computational complexity, since exact solutions

cannot be solved in polynomial time. Hence, it is required to find solutions that use approximations that reduce the problem's complexity.

In the topic of having multiple possible paths between a source and destination node, it is also required to use algorithms that are able to compute different paths with a maximum degree of disjointness, either considering only the involved links, or expanding that constraint to the nodes in the calculated paths. If one desires to solve this problem with added constraints, the same approach needs to be taken as in the single-path problem, as it is required to simplify the original problem to be able to perform fast calculations. SAMCRA was presented as a solution that uses a linear approximation of the existing constraints and variables, computing multiple paths while respecting the imposed constraint rules.

Following these introductory topics regarding SDN, transport protocols and path computation algorithms, a proposal for a SDN controller application running on OpenDaylight was presented. This application was designed to deliver path computation in a SDN network, where multiple devices are managed and multiple paths can be formed between the existing endpoints. This would allow paths to be computed in real time, following events where the controller receives an unmatched packet from one of the managed switch devices, and offering different path computation strategies that would include single-path routing (using Dijkstra's algorithm), multiple path computation (with Yen's algorithm), link-disjoint path calculation (by using an iterative Dijkstra's approach) and latency-aware multiple-path computation (using SAMCRA), by monitoring QoS metrics in the managed network. When more than one path was available, it was possible to configure the application to use 4 different flow pinning strategies: Round-robin, Hash-based, Minimum flows and Random.

The performed evaluation initially compared the behaviour of the different multiple flow scheduler solutions, along with the usage of TCP and MPTCP, taking in consideration the completion time of multiple file transfers across a topology where multiple paths were available. A second phase of the evaluation combined the usage of MPTCP with the different implemented path computation algorithms in a different topology where multiple paths were also available, but with different delay configurations. For this evaluation, the average transfer completion time was also considered, along with the individual file throughput and

the measured end-to-end delay. Contrary to the previous experiments, during this second phase only the Round-robin and Hash-based flow pinning strategies were used.

The obtained results showed, in general, lower transfer times when comparing MPTCP and TCP, specially when using the Minimum flows and Round-robin flow schedulers. The interpretation of the results for the tests made with the different path computation algorithms produced different outcomes. When considering the measured average transfer time and throughput from the multiple file transfers, the results from the experiments with the link disjoint path algorithm showed that using an algorithm that computes multiple fully link-disjoint paths is only efficient in a topology where there are a high number of paths that fulfil that condition. Otherwise, as seen in the conducted experiments, the algorithm will only obtain a short number of paths and, for scenarios with large traffic demands, the used paths, despite being disjointed, will easily be congested.

On the other hand, Yen's algorithm produced the lower transfer times and highest throughput almost every test, proving that for high traffic demand scenarios, using a greater number of different paths, even if with different end-to-end delay configurations or repeated links (across the different paths) was beneficial. For the evaluation of SAMCRA, while analysing the throughput and average transfer time, the obtained results were not as good as Yen's algorithm, due to restrictions in its implementation when finding different paths, since the links from the previous calculated paths were not taken in consideration for the new paths every time the algorithm needed to run multiple times. However, by the evaluation of the round-trip time delay from each one of the established MPTCP sub-flows, SAMCRA presented lower delay values in its results, even in scenarios with a high traffic demand, by taking in consideration this type of metric when calculating new paths (even if the imposed constraint could not be fulfilled). The other algorithms, since they operated with a shortest path metric, did not consider the existing path delays, always preferring scenarios paths with the least hop count first, even despite its high end-to-end delay.



# References

- [Ahrenholz, 2010] Jeff Ahrenholz. Comparison of CORE network emulation platforms. In *MILITARY COMMUNICATIONS CONFERENCE, 2010-MILCOM 2010*, pages 166–171. IEEE, 2010. [48](#)
- [Allan et al., 2010] David Allan, Peter Ashwood-Smith, Nigel Bragg, János Farkas, Don Fedyk, Michel Ouellete, Mick Seaman, and Paul Unbehagen. Shortest path bridging: efficient control of larger ethernet networks. *Communications Magazine, IEEE*, 48(10):128–135, 2010. [32](#)
- [Arzani et al., 2014] Behnaz Arzani, Alexander Gurney, Shuotian Cheng, Roch Guerin, and Boon Thau Loo. Impact of path characteristics and scheduling policies on mptcp performance. In *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*, pages 743–748. IEEE, 2014. [91](#)
- [Becke et al., 2013] M. Becke, H. Adhari, E.P. Rathgeb, Fu Fa, Xiong Yang, and Xing Zhou. Comparison of multipath tcp and cmt-sctp based on intercontinental measurements. In *Global Communications Conference (GLOBECOM), 2013 IEEE*, pages 1360–1366, Dec 2013. [21](#)
- [Bellman, 1956] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956. [28](#), [32](#)
- [Big Switch Networks, 2012] Big Switch Networks. Project Floodlight. <http://http://www.projectfloodlight.org/>, 2012. Accessed: 2015-01-12. [11](#)
- [Brakman et al., 1999] Steven Brakman, Harry Garretsen, Charles Van Marrewijk, and Marianne Van Den Berg. The return of zipf: towards a further understanding

- of the rank-size distribution. *Journal of Regional Science*, 39(1):183–213, 1999. [85](#)
- [Brakmo and Peterson, 1995] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, 1995. [16](#)
- [Brander and Sinclair, 1996] Andrew William Brander and Mark C Sinclair. *A comparative study of k-shortest path algorithms*. Springer, 1996. [33](#)
- [Bredel et al., 2014] Michael Bredel, Zdravko Bozakov, Artur Barczyk, and Harvey Newman. Flow-based load balancing in multipathed layer-2 networks using openflow and multipath-tcp. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 213–214. ACM, 2014. [83](#)
- [Cardei and Wu, 2006] Mihaela Cardei and Jie Wu. Energy-efficient coverage problems in wireless ad-hoc sensor networks. *Computer communications*, 29(4):413–420, 2006. [35](#)
- [Case et al., 1989] Jeffery Case, Mark Fedor, Martin Schoffstall, and C Davin. A simple network management protocol (snmp), 1989. [5](#)
- [Cerf and Icahn, 2005] Vinton G Cerf and Robert E Icahn. A protocol for packet network intercommunication. *ACM SIGCOMM Computer Communication Review*, 35(2):71–82, 2005. [15](#)
- [Chap et al., 2011] Tithra Chap, Xin Wang, Sugang Xu, and Yoshiaki Tanaka. Link-disjoint routing algorithms with link-disjoint degree and resource utilization concern in translucent wdm optical networks. In *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, pages 357–362. IEEE, 2011. [35](#), [39](#)
- [Chawanat et al., 2014] Nakasan Chawanat, Ichikawa Kohei, and Uthayopas Putchong. Performance evaluation of mptcp over openflow network. *Information Processing Society of Japan SIG Notes 2014*, 2014(30):1–6, 2014. [21](#)



- [Chen et al., 2013] Shengyang Chen, Zhenhui Yuan, and G-M Muntean. An energy-aware multipath-tcp-based content delivery scheme in heterogeneous wireless networks. In *Wireless Communications and Networking Conference (WCNC), 2013 IEEE*, pages 1291–1296. IEEE, 2013. 19
- [Chen and Nahrsted, 1998] Shigang Chen and Klara Nahrsted. An overview of quality of service routing for next-generation high-speed networks: problems and solutions. *Network, IEEE*, 12(6):64–79, 1998. 30, 31
- [Chen and Nahrstedt, 1998] Shigang Chen and Klara Nahrstedt. On finding multi-constrained paths. In *Communications, 1998. ICC 98. Conference Record. 1998 IEEE International Conference on*, volume 2, pages 874–879. IEEE, 1998. 30, 32
- [Cherkassky et al., 1996] Boris V Cherkassky, Andrew V Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996. 26
- [Claise, 2004] Benoit Claise. Cisco systems netflow services export version 9. 2004. 47
- [CPqD, 2012] CPqD. nox13oflib. <http://github.com/CPqD/nox13oflib>, 2012. Accessed: 2015-01-12. 10
- [De Neve and Van Mieghem, 2000] Hans De Neve and Piet Van Mieghem. Tamcra: a tunable accuracy multiple constraints routing algorithm. *Computer Communications*, 23(7):667–679, 2000. 38, 39
- [Dijkstra, 1959] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. 26, 32
- [Diop et al., 2012] Codé Diop, Guillaume Dugué, Christophe Chassot, and Ernesto Exposito. Qos-oriented mptcp extensions for multimedia multi-homed systems. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 1119–1124. IEEE, 2012. 22
- [Dobrijević et al., 2014] Ognjen Dobrijević, Andreas Kassler, Lea Skorin-Kapov, and Maja Matijašević. Q-point: Qoe-driven path optimization model for multi-

- media services. *Computer Communication Networks and Telecommunications*, 8458:134–147, 2014. [41](#)
- [Egilmez et al., 2012] H.E. Egilmez, S.T. Dane, K. T. Bagci, and A. M. Tekalp. Open-qos: An openflow controller design for multimedia delivery with end-to-end quality of service over software-defined networks. In *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*, pages 1–8. IEEE, 2012. [41](#)
- [Egilmez et al., 2013] Hilmi E Egilmez, Seyhan Civanlar, and A Murat Tekalp. An optimization framework for qos-enabled adaptive video streaming over openflow networks. *Multimedia, IEEE Transactions on*, 15(3):710–715, 2013. [41](#)
- [Enns et al., 2011] R Enns, M Bjorklund, J Schoenwaelder, and A Bierman. Network configuration protocol (netconf). *Internet Engineering Task Force, RFC*, 6241, 2011. [5](#)
- [Eppstein, 1998] David Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673, 1998. [33](#), [39](#)
- [Erickson, 2013] David Erickson. The Beacon OpenFlow Controller. In *HotSDN*. ACM, 2013. [10](#)
- [Feamster et al., 2014] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014. [5](#)
- [Fisher, 2004] Marshall L Fisher. The lagrangian relaxation method for solving integer programming problems. *Management science*, 50(12.supplement):1861–1871, 2004. [31](#)
- [Ford et al., 2011] Alan Ford, Costin Raiciu, Mark Handley, Olivier Bonaventure, et al. Tcp extensions for multipath operation with multiple addresses. RFC RFC 6824, RFC Editor, 2011. [15](#)
- [Fredman and Tarjan, 1987] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987. [26](#)

- [Funasaka et al., 2005] Junichi Funasaka, Kenji Ishida, Hiroyasu Obata, and Yuki Yoshi Jutori. A study on primary path switching strategy of sctp. In *Autonomous Decentralized Systems, 2005. ISADS 2005. Proceedings*, pages 536–541. IEEE, 2005. [19](#)
- [Foundation, 2012] Open Networking Foundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012. [5](#)
- [Garey et al., 1980] Michael R Garey, David S Johnson, Gary L Miller, and Christos H Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic Discrete Methods*, 1(2):216–227, 1980. [30](#)
- [Giotis et al., 2014] K Giotis, Christos Argyropoulos, Georgios Androulidakis, Dimitrios Kalogeras, and Vasilis Maglaris. Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments. *Computer Networks*, 62:122–136, 2014. [8](#)
- [Grinnemo and Brunström, 2015] Karl-Johan Grinnemo and Anna Brunström. A first study on using mptcp to reduce latency for cloud based mobile applications. In *6th IEEE International Workshop on Performance Evaluation of Communications in Distributed Systems and Web based Service Architectures (PEDIS-WESA)*. IEEE Computer Society, 2015. [91](#)
- [Guérin and Orda, 1999] Roche A Guérin and Ariel Orda. Qos routing in networks with inaccurate information: theory and algorithms. *IEEE/ACM Transactions on Networking (TON)*, 7(3):350–364, 1999. [30](#)
- [Guo et al., 2003] Yuchun Guo, Fernando Kuipers, and Piet Van Mieghem. Link-disjoint paths for reliable qos routing. *International Journal of Communication Systems*, 16(9):779–798, 2003. [38](#), [39](#)
- [Hanks et al., 2000] Stan Hanks, David Meyer, Dino Farinacci, and Paul Traina. Generic routing encapsulation (gre). 2000. [47](#)
- [Hart et al., 1968] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968. [27](#), [32](#)

- [Hayes et al., 2008] David A Hayes, Jason But, and Grenville Armitage. Issues with network address translation for sctp. *ACM SIGCOMM Computer Communication Review*, 39(1):23–33, 2008. 19
- [Hedrick, 1988] C. L. Hedrick. RFC 1058: Routing information protocol, June 1988. 8
- [Hesmans and Bonaventure, 2014] Benjamin Hesmans and Olivier Bonaventure. Tracing multipath tcp connections. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM ’14, pages 361–362, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2631453. 96
- [Hopps, 2000] C. Hopps. Analysis of an equal-cost multi-path algorithm. *Internet Engineering Task Force, RFC*, 2000. 40
- [Huang and Lin, 2013] Chung-Ming Huang and Ming-Sian Lin. The unreliable-concurrent multipath transfer (u-cmt) protocol for multihomed networks. *Telecommunication Systems*, 52(1):245–259, 2013. 19
- [Humernbrum et al., 2014] Tim Humernbrum, Frank Glinka, and Sergei Gorlatch. A northbound api for qos management in real-time interactive applications on software-defined networks. *Journal of Communications*, 9(8), 2014. 8
- [IEEE Standards Association, 2004] IEEE Standards Association. IEEE Standard for Local and Metropolitan Area NetworksMedia access control (MAC) Bridges. <http://standards.ieee.org/getieee802/download/802.1D-2004.pdf>, 2004. Accessed: 2014-11-27. 32
- [Ishiguro et al., 2007] Kunihiro Ishiguro, T Takada, Y Ohara, AD Zinin, G Natapov, and A Mizutani. Quagga routing suite, 2007. 7
- [ITU-T, 2003] ITU-T. G.114 : One-way transmission time. <https://www.itu.int/rec/T-REC-G.114-200305-I/en>, 2003. Accessed: 2015-06-30. 111
- [ITU-T, 2008] ITU-T. Definitions of terms related to quality of service. Recommendation E.800, International Telecommunication Union, Geneva, 2008. 8

- [Iyengar et al., 2006] Janardhan R Iyengar, Paul D Amer, and Randall Stewart. Concurrent multipath transfer using sctp multihoming over independent end-to-end paths. *Networking, IEEE/ACM Transactions on*, 14(5):951–964, 2006. 18
- [Jacobson, 1988] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988. 16
- [Jacobson, Van and Leres, Craig and McCanne, S, 2010] Jacobson, Van and Leres, Craig and McCanne, S. tcpdump. <http://www.tcpdump.org/>, 2010. Accessed: 2015-06-28. 97
- [Jain et al., 2013] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 3–14. ACM, 2013. 7
- [Johnson, 1977] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977. 28, 32
- [Juttner et al., 2001] Alpar Juttner, Balazs Szviatovski, Ildikó Mécs, and Zsolt Rajkó. Lagrange relaxation based method for the qos routing problem. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 859–868. IEEE, 2001. 31, 32
- [Kanaumi et al., 2010] Yoshihiko Kanaumi, Shuichi Saito, and Eiji Kawai. Deployment of a programmable network for a nation wide r&d network. In *Network Operations and Management Symposium Workshops (NOMS Wksp), 2010 IEEE/IFIP*, pages 233–238. IEEE, 2010. 7
- [Katz, 2010] David Katz. Bidirectional forwarding detection (bfd). *Internet Engineering Task Force, RFC*, 2010. 40

- [Kim and Feamster, 2013] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2):114–119, 2013. 6
- [Kohler, 2006] Eddie Kohler. Datagram congestion control protocol (dccc). *Internet Engineering Task Force, RFC*, 2006. 15
- [Kohler et al., 2006] Eddie Kohler, Mark Handley, and Sally Floyd. Designing dccc: Congestion control without reliability. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 27–38. ACM, 2006. 17
- [Konsgen et al., 2012] A Konsgen, Amanpreet Singh, Ma Jun, Thushara Weerawardane, and Carmelita Goerg. Responsiveness of future telecommunication networks under disaster situations. In *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2012 4th International Congress on*, pages 892–899. IEEE, 2012. 22
- [Kreutz et al., 2014] Diego Kreutz, Fernando Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *arXiv preprint arXiv:1406.0440*, 2014. xv, 9, 12
- [Kuipers et al., 2002] Fernando Kuipers, Piet Van Mieghem, Turgay Korkmaz, and Marwan Krunz. An overview of constraint-based path selection algorithms for qos routing. *IEEE Communications Magazine*, 40(12):50–55, 2002. 31
- [Kulcloud, 2012] Kulcloud. OpenMUL Controller. <http://www.openmul.org/openmul-controller.html>, 2012. Accessed: 2015-01-12. 11
- [Lantz et al., 2010] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010. 49
- [Lara et al., 2013] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using openflow: A survey. *Communications Surveys & Tutorials, IEEE*, 16(1):493 – 512, 2013. xv, 9, 12

- [Lawler, 1972] Eugene L Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972. [33](#)
- [Lee et al., 2014] Steven SW Lee, Kuang-Yi Li, Kwan-Yee Chan, Guan-Hao Lai, and Yao-Chuan Chung. Path layout planning and software based fast failure detection in survivable openflow networks. In *Design of Reliable Communication Networks (DRCN), 2014 10th International Conference on the*, pages 1–8. IEEE, 2014. [40](#)
- [Lee and Gerla, 2001] Sung-Ju Lee and Mario Gerla. Split multipath routing with maximally disjoint paths in ad hoc networks. In *Communications, 2001. ICC 2001. IEEE International Conference on*, volume 10, pages 3201–3205. IEEE, 2001. [36](#)
- [Lee et al., 1995] Whay C Lee, Michael G Hluchyi, and Pierre A Humblet. Routing subject to quality of service constraints in integrated communication networks. *Network, IEEE*, 9(4):46–55, 1995. [30](#)
- [Levin et al., 2012] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012. [10](#)
- [Li et al., 1990] Chung-Lun Li, S Thomas McCormick, and David Simchi-Levi. The complexity of finding two disjoint paths with min-max objective function. *Discrete Applied Mathematics*, 26(1):105–115, 1990. [38](#)
- [Li et al., 1992] Chung-Lun Li, S Thomas McCormick, and David Simchi-Levi. Finding disjoint paths with different path-costs: Complexity and algorithms. *Networks*, 22(7):653–667, 1992. [34](#), [35](#), [38](#), [39](#)
- [Li and Pan, 2013] Yu Li and Deng Pan. Openflow based load balancing for fat-tree networks with multipath support. In *Proc. 12th IEEE International Conference on Communications (ICC13), Budapest, Hungary*, pages 1–5, 2013. [40](#)

- [Liu and Ramakrishnan, 2001] Gang Liu and KG Ramakrishnan. A\* prune: an algorithm for finding k shortest paths subject to multiple constraints. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 743–749. IEEE, 2001. [37](#), [39](#)
- [Mahalingam et al., 2014] M Mahalingam, D Dutt, K Duda, P Agarwal, L Kreeger, T Sridhar, M Bursell, and C Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. *Internet Req. Comments*, 2014. [47](#)
- [Mascolo et al., 2001] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sana-didi, and Ren Wang. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297. ACM, 2001. [16](#)
- [McKeown et al., 2008] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008. [5](#)
- [Medved et al., 2014] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Open-daylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014. [11](#)
- [Meghanathan, 2007] Natarajan Meghanathan. Stability and hop count of node-disjoint and link-disjoint multi-path routes in ad hoc networks. In *Wireless and Mobile Computing, Networking and Communications, 2007. WiMOB 2007. Third IEEE International Conference on*, pages 42–42. IEEE, 2007. [36](#)
- [Moy, 1989] J. Moy. RFC 1131: OSPF specification, October 1989. [8](#)
- [Nascimento et al., 2010] Marcelo Ribeiro Nascimento, Christian Esteve Rothenberg, Marcos Rogério Salvador, and Maurício Ferreira Magalhães. Quagflow: partnering quagga with openflow. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 441–442. ACM, 2010. [7](#)



- [Németh et al., 2013] Felicián Németh, Balázs Sonkoly, Levente Csikor, and András Gulyás. A large-scale multipath playground for experimenters and early adopters. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 481–482. ACM, 2013. 22
- [Nicira, 2008] Nicira. NOXRepo. <http://www.noxrepo.org/nox/about-nox/>, 2008. Accessed: 2015-01-12. 10
- [Nunes et al., 2014] B Nunes, Marc Mendonca, X Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *Communications Surveys & Tutorials, IEEE*, 16(3):1617–1634, 2014. xv, 9, 12
- [Pfaff and Davie, 2013] B Pfaff and B Davie. The open vswitch database management protocol. Technical report, RFC 7047, IETF, December, 2013. xi, 9, 48
- [Pfaff et al., 2009] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009. 9
- [Phaal et al., 2001] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporations sflow: A method for monitoring traffic in switched and routed networks. Technical report, RFC 3176, 2001. 8
- [Phemius and Bouet, 2013] Kévin Phemius and Mathieu Bouet. Monitoring latency with openflow. In *Network and Service Management (CNSM), 2013 9th International Conference on*, pages 122–125. IEEE, 2013. 72
- [Porras et al., 2012] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 121–126. ACM, 2012. 7
- [Postel, 1980] Jon Postel. User datagram protocol. *Internet Engineering Task Force, RFC*, 1980. 15
- [Postel, 1981] Jon Postel. Transmission control protocol. *Internet Engineering Task Force, RFC*, 1981. 16

- [R. Stewart, 2007] Ed. R. Stewart. Stream control transmission protocol. IETF Request for Comments: 4960, September 2007. 15
- [Raiciu et al., 2011] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 266–277. ACM, 2011. 21
- [Raumer et al., 2014] Daniel Raumer, Lukas Schwaighofer, and Georg Carle. Mon-samp: A distributed sdn application for qos monitoring. *Federated Conference on Computer Science and Information Systems*, 2014. 8
- [Rekhter and Li, 1995] Y. Rekhter and T. Li. RFC 1771: A Border Gateway Protocol 4 (BGP-4), March 1995. 8
- [Russell et al., 1995] Stuart Russell, Peter Norvig, and Artificial Intelligence. Artificial intelligence: A modern approach. *Prentice-Hall, Egnlewood Cliffs*, 25, 1995. 27
- [Ryu SDN Framework Community, 2012] Ryu SDN Framework Community. Ryu SDN Framework. <http://osrg.github.io/ryu/>, 2012. Accessed: 2015-01-12. 11
- [Santos et al., 2015] Ricardo Santos, Marilia Curado, and Andreas Kasser. Multipathing in software defined networking: Interaction between sdn and mptcp. In *Swedish Communication Technologies Workshop (Swe-CTW) 2015*, 2015. 3
- [Sezer et al., 2013] S. Sezer, S. Scott-Hayward, P.K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao. Are we ready for sdn? implementation challenges for software-defined networks. *IEEE M COM*, 51(7): 36–43, 2013. doi: 10.1109/MCOM.2013.6553676. 7
- [Shiloach and Perl, 1978] Y Shiloach and Y Perl. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the ACM (JACM)*, 25(1):1–9, 1978. 34
- [Sidhu et al., 1991] Deepinder Sidhu, Raj Nair, and Shukri Abdallah. Finding disjoint paths in networks. *ACM SIGCOMM Computer Communication Review*, 21(4): 43–51, 1991. 36

- [Sipser, 2006] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2006. 25
- [Smith et al., 2014] M Smith, M Dvorkin, Y Laribi, V Pandey, P Garg, and N Weidenbacher. Opflex control protocol. *IETF*, <http://datatracker.ietf.org/doc/draft-smith-opflex>, 2014. 9
- [Song, 2013] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM, 2013. 9
- [Sousa et al., 2013] Bruno Sousa, Ricardo Santos, Marilia Curado, Soila Pertet, Rajeev Gandhi, Carlos Silva, and Kostas Pentikousis. Expedient reconfiguration in the cloud. In *Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2013 IEEE 18th International Workshop on*, pages 243–247. IEEE, 2013. 19
- [Stewart et al., 2004] R Stewart, M Ramalho, Q Xie, M Tuexen, and P Conrad. Stream control transmission protocol (sctp) partial reliability extension. Technical report, RFC 3758 (Proposed Standard), 2004. 19
- [Suh et al., 2014] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 228–237. IEEE, 2014. 8
- [Suurballe and Tarjan, 1984] John W Suurballe and Robert Endre Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984. 34, 39
- [Suurballe, 1974] JW Suurballe. Disjoint paths in a network. *Networks*, 4(2):125–145, 1974. 36
- [Taft-Plotkin et al., 1999] Nina Taft-Plotkin, Bhargav Bellur, and Richard Ogier. Quality-of-service routing using maximally disjoint paths. In *Quality of Ser-*

- vice*, 1999. *IWQoS'99. 1999 Seventh International Workshop on*, pages 119–128. IEEE, 1999. [38](#)
- [Tootoonchian and Ganjali, 2010] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3. USENIX Association, 2010. [6](#)
- [Upadhyaya and Dhingra, 2010] Shuchita Upadhyaya and Gaytri Dhingra. Exploring issues for qos based routing algorithms. *International Journal on Computer Science and Engineering*, 2(5), 2010. [31](#)
- [Van Adrichem et al., 2014] Niels LM Van Adrichem, Christian Doerr, Fernando Kuipers, et al. Opennetmon: Network monitoring in openflow software-defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–8. IEEE, 2014. [72](#)
- [van der Pol et al., 2011] Ronald van der Pol, S Boele, F Dijkstra, J Mambretti, J Chen, FI Yeh, M Savoie, B Ho, and L Sun. Monitoring and troubleshooting openflow slices with an open source implementation of iee 802.1 ag, 2011. [40](#)
- [van der Pol et al., 2012] Ronald van der Pol, Sander Boele, Freek Dijkstra, Artur Barczyk, Gerben van Malenstein, Jim Hao Chen, and Joe Mambretti. Multipathing with mptcp and openflow. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion.*, pages 1617–1624. IEEE, 2012. [21](#)
- [van der Pol et al., 2013] Ronald van der Pol, Michael Bredel, Artur Barczyk, B Overeinder, NLM van Adrichem, and FA Kuipers. Experiences with mptcp in an intercontinental multipathed openflow network. In *Proceedings of the 29th Trans European Research and Education Networking Conference, D. Foster, Ed. TERENA*, 2013. [21](#)
- [Van Mieghem et al., 2001] Piet Van Mieghem, Hans De Neve, and Fernando Kuipers. Hop-by-hop quality of service routing. *Computer Networks*, 37(3):407–423, 2001. [38](#), [39](#)

- [Vestin and Kassler, 2015] Jonathan Vestin and Andreas Kassler. Qos enabled wifi mac layer processing as an example of a nfv service. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–9. IEEE, 2015. [108](#)
- [Wang and Crowcroft, 1996] Zheng Wang and Jon Crowcroft. Quality-of-service routing for supporting multimedia applications. *Selected Areas in Communications, IEEE Journal on*, 14(7):1228–1234, 1996. [29](#), [38](#)
- [Widyono et al., 1994] Ron Widyono et al. *The design and evaluation of routing algorithms for real-time channels*. International Computer Science Institute Berkeley, 1994. [37](#)
- [Wischik et al., 2011] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, volume 11, pages 8–8, 2011. [21](#)
- [Xia et al., 2014] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A survey on software-defined networking. *Communications Surveys & Tutorials, IEEE*, 2014. [xv](#), [9](#), [12](#)
- [Yang et al., 2004] Lily Yang, Ram Dantu, T Anderson, and Ram Gopal. Forwarding and control element separation (forces) framework. Technical report, RFC 3746, April, 2004. [9](#)
- [Yeganeh et al., 2013] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *Communications Magazine, IEEE*, 51(2):136–141, 2013. [10](#)
- [Yen, 1971] Jin Y Yen. Finding the k shortest loopless paths in a network. *management Science*, 17(11):712–716, 1971. [32](#), [39](#)
- [Yuan, 1999] Xin Yuan. On the extended bellman-ford algorithm to solve two-constrained quality of service routing problems. In *Computer Communications and Networks, 1999. Proceedings. Eight International Conference on*, pages 304–310. IEEE, 1999. [31](#), [37](#), [39](#)



# Appendix A - Transport protocol and flow scheduler evaluation results

This appendix presents all the additional figures and tables containing results from the transport protocol and flow scheduler evaluation experiments, detailed in section 6.1.

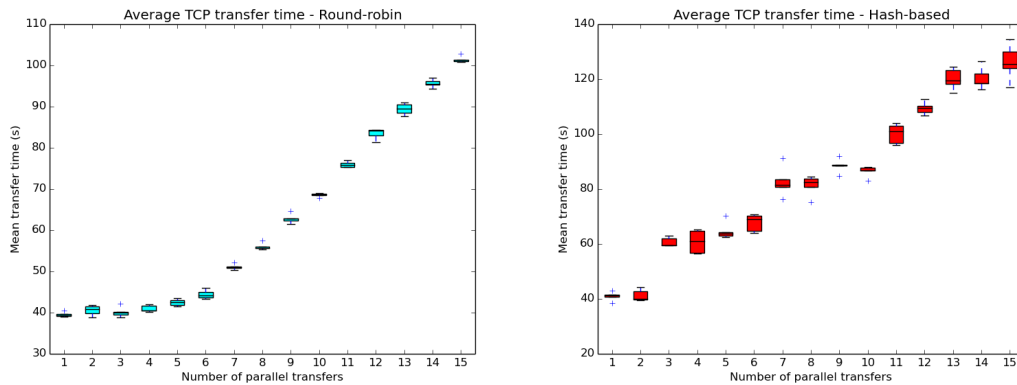


Figure A.1: Round-robin evaluation re-Figure A.2: Hashed-based evaluation results, no SDN controller results, no SDN controller

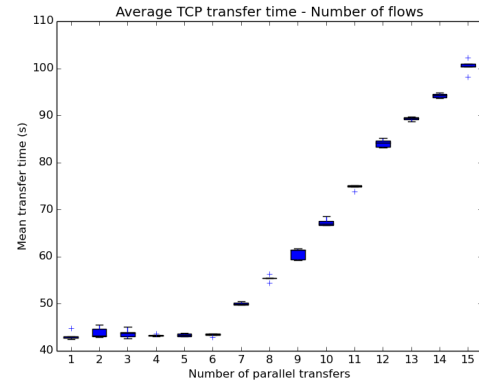
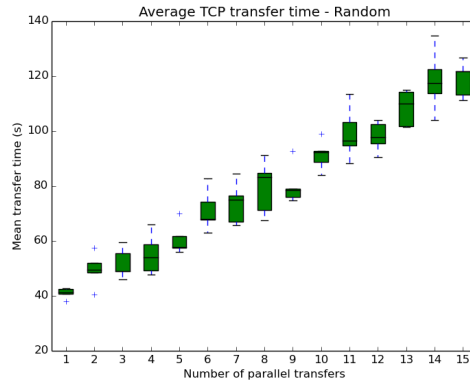


Figure A.3: Random scheduler evaluation results, no SDN controller

Figure A.4: Minimum flow evaluation results, no SDN controller

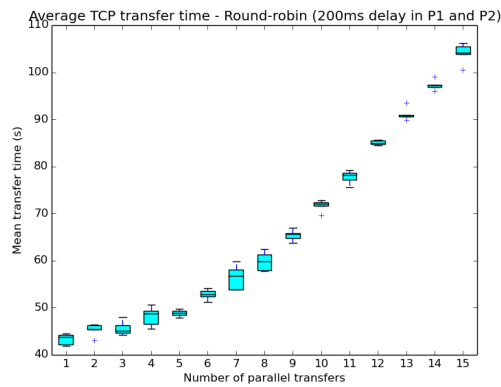


Figure A.5: RR scheduler evaluation results (with delay), no SDN controller

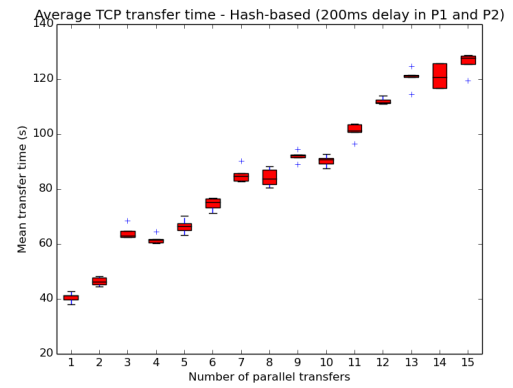


Figure A.6: Hash-based scheduler evaluation results (with delay), no SDN controller



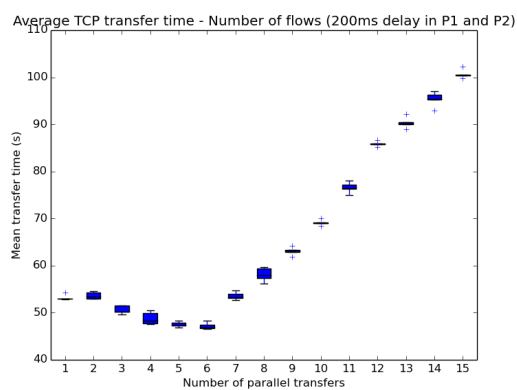
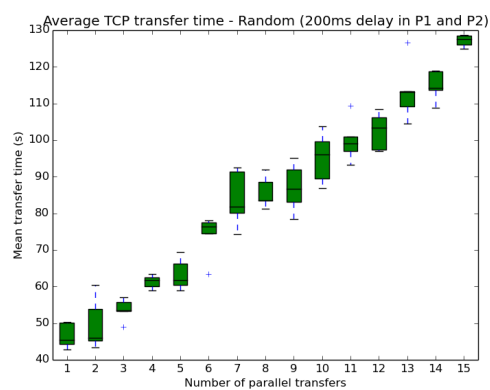


Figure A.7: Random scheduler evaluation results (with delay), no SDN controller  
 Figure A.8: Min flows scheduler evaluation results (with delay), no SDN controller

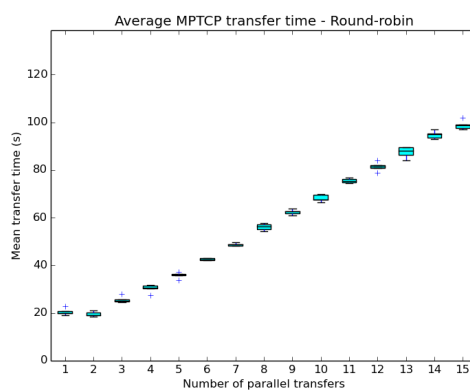
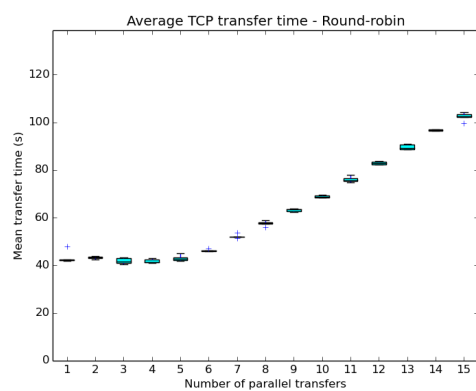


Figure A.9: RR scheduler evaluation results with TCP, with SDN controller  
 Figure A.10: RR scheduler evaluation results with MPTCP, with SDN controller

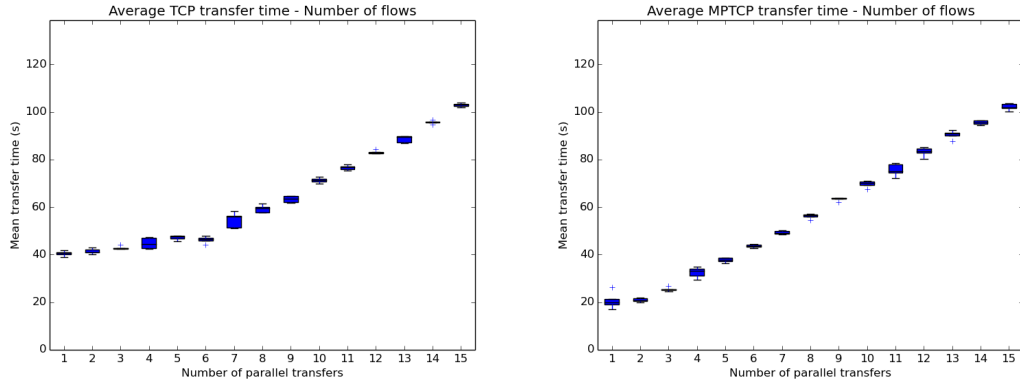


Figure A.11: Min flows scheduler evaluation results with TCP, with SDN controller  
 Figure A.12: Min flows scheduler evaluation results with MPTCP, with SDN controller

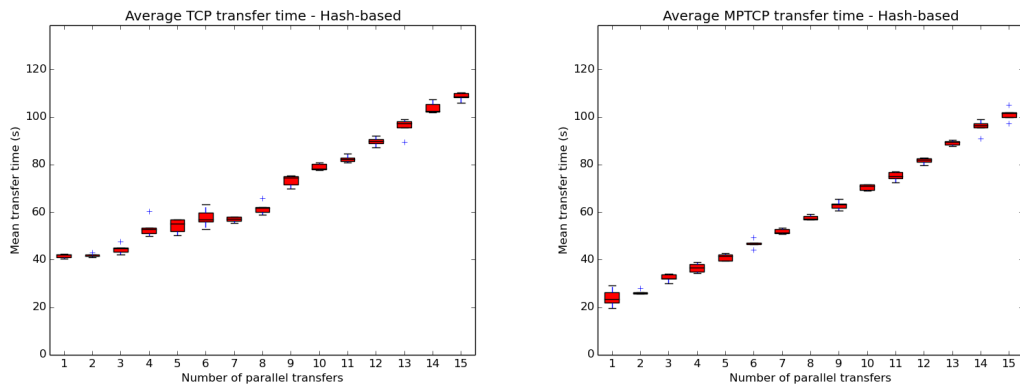


Figure A.13: Hash-based scheduler evaluation results with TCP, with SDN controller  
 Figure A.14: Hash-based scheduler evaluation results with MPTCP, with SDN controller

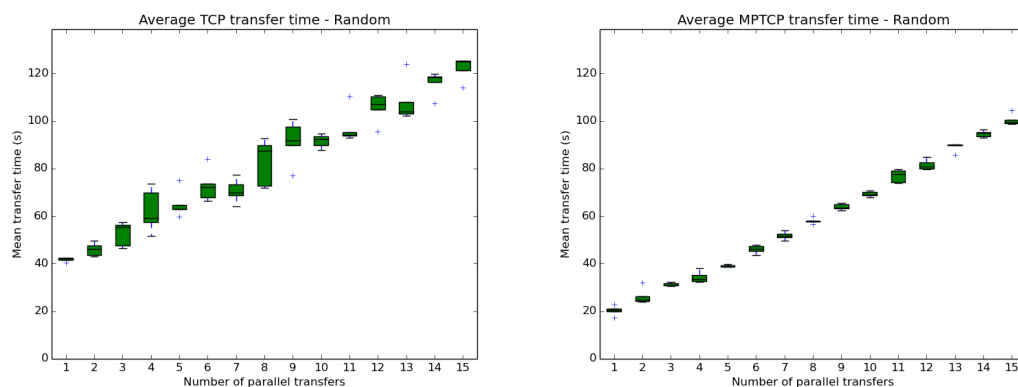


Figure A.15: Random scheduler evaluation results with TCP, with SDN controller

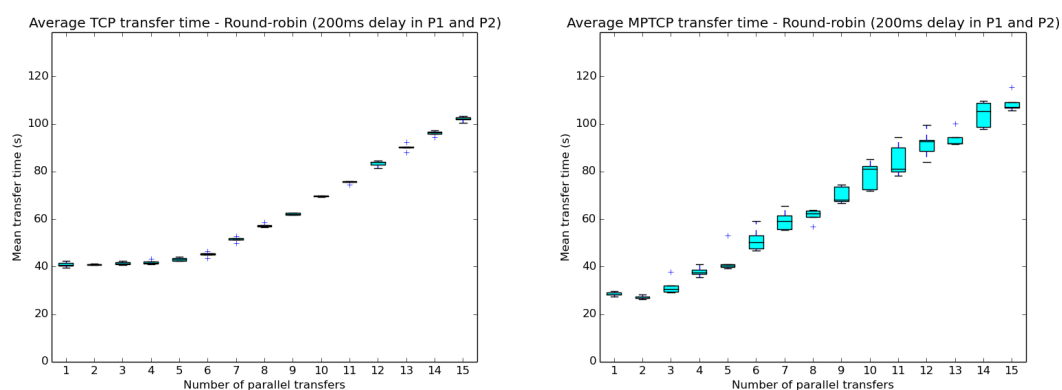


Figure A.17: RR scheduler evaluation results with delay and TCP, with SDN controller

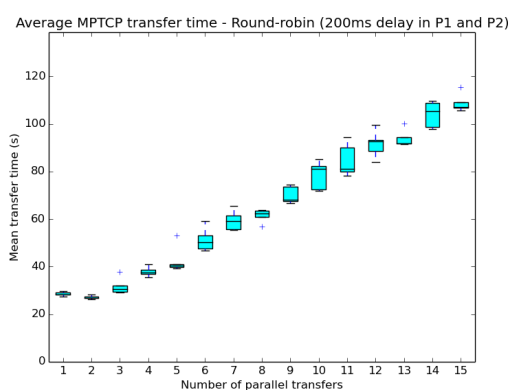


Figure A.18: RR scheduler evaluation results with delay and MPTCP, with SDN controller

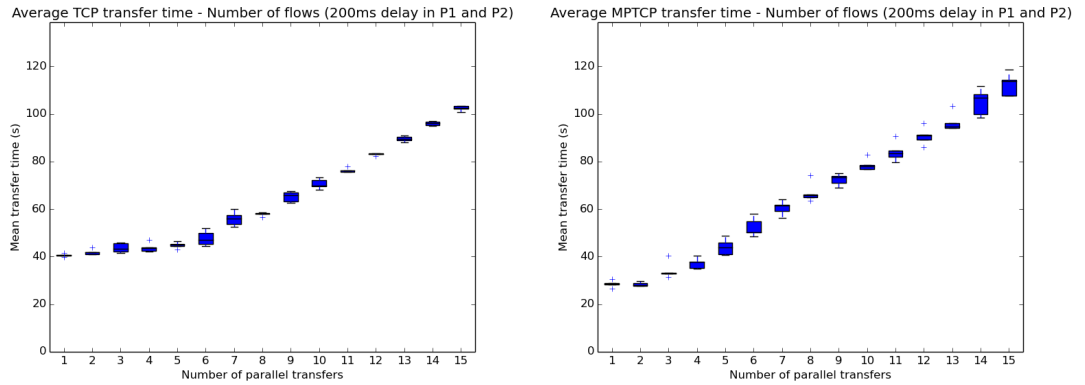


Figure A.19: Min flows scheduler evaluation results with delay and TCP, with SDN controller

Figure A.20: Min flows scheduler evaluation results with delay and MPTCP, with SDN controller

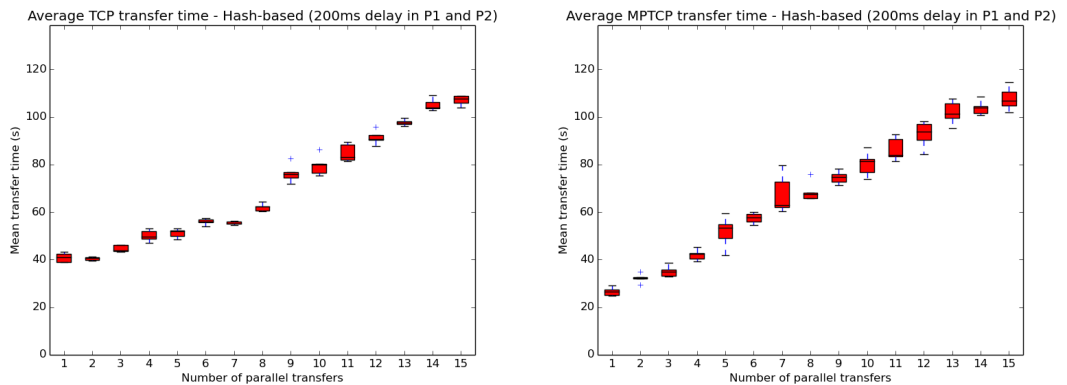


Figure A.21: Hash-based scheduler evaluation results with delay and TCP, with SDN controller

Figure A.22: Hash-based scheduler evaluation results with delay and MPTCP, with SDN controller

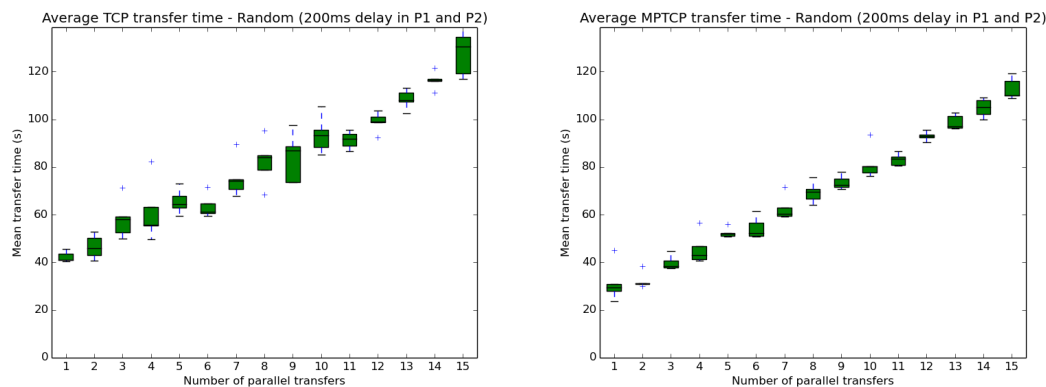


Figure A.23: Random scheduler evaluation results with delay and TCP, with SDN controller  
 Figure A.24: Random scheduler evaluation results with delay and MPTCP, with SDN controller

Table A.1: Average TCP transfer times with static flow allocation and no path delay

Transfers	Random	Hash-based	Number of flows	Round-robin
1	41.046	40.938	43.186	39.536
2	49.620	41.294	43.872	40.579
3	51.800	60.715	43.755	40.136
4	55.171	60.847	43.318	40.987
5	60.592	64.740	43.374	42.451
6	71.208	67.758	43.431	44.469
7	73.785	82.662	50.059	51.088
8	79.582	81.391	55.415	56.010
9	80.221	88.484	60.680	62.840
10	91.363	86.446	67.281	68.608
11	99.194	100.135	74.793	75.922
12	98.045	109.472	84.122	83.477
13	108.485	120.166	89.278	89.416
14	118.501	120.402	94.221	95.692
15	118.935	126.195	100.419	101.461

Table A.2: Average TCP transfer times with static flow allocation and path delay

Transfers	Random	Hash-based	Number of flows	Round-robin
1	46.570	40.616	53.208	43.306
2	49.803	46.413	53.630	45.286
3	53.684	64.291	50.596	45.661
4	61.319	61.693	48.807	48.176
5	63.365	66.511	47.524	48.839
6	74.034	74.660	47.213	52.809
7	84.041	85.291	53.535	56.458
8	85.725	84.257	58.147	59.884
9	87.019	92.018	63.106	65.379
10	95.180	90.272	69.135	71.715
11	99.918	101.076	76.553	77.785
12	102.451	112.062	85.862	85.052
13	113.360	120.569	90.368	91.154
14	114.884	121.118	95.388	97.348
15	127.099	125.976	100.671	104.063

Table A.3: Average TCP transfer times using the controller application and no path delay

Transfers	Random	Hash-based	Number of flows	Round-robin
1	41.850	41.630	40.576	43.268
2	46.035	41.858	41.491	43.264
3	52.676	44.580	42.954	41.850
4	62.332	53.559	44.764	41.747
5	65.056	54.235	47.153	43.037
6	72.883	57.696	46.281	46.205
7	70.704	56.974	54.693	52.150
8	82.989	61.700	59.268	57.620
9	91.412	73.291	63.289	63.162
10	91.669	79.082	71.329	68.871
11	97.329	82.378	76.700	76.029
12	105.748	89.721	83.067	82.972
13	108.218	95.882	88.586	89.659
14	116.143	103.864	95.779	96.740
15	122.226	108.604	103.007	102.382

Table A.4: Average MPTCP transfer times using the controller application and no path delay

<b>Transfers</b>	<b>Random</b>	<b>Hash-based</b>	<b>Number of flows</b>	<b>Round-robin</b>
1	20.302	24.090	20.702	20.452
2	26.233	26.253	20.955	19.519
3	31.313	32.374	25.501	25.605
4	34.333	36.557	32.578	30.342
5	39.156	41.171	37.744	35.858
6	46.072	46.812	43.596	42.526
7	51.841	51.868	49.357	48.700
8	57.964	57.708	56.196	56.159
9	63.937	62.952	63.420	62.371
10	69.299	70.489	69.777	68.261
11	76.966	75.133	75.630	75.542
12	81.583	81.637	83.280	81.429
13	89.192	89.122	90.515	87.492
14	94.711	95.847	95.624	94.814
15	100.362	101.146	102.141	98.932

Table A.5: Average TCP transfer times using the controller application with path delay

<b>Transfers</b>	<b>Random</b>	<b>Hash-based</b>	<b>Number of flows</b>	<b>Round-robin</b>
1	42.350	40.852	40.668	40.950
2	46.543	40.550	41.912	40.962
3	58.204	44.714	43.714	41.519
4	61.302	50.093	43.805	41.897
5	65.512	51.189	44.800	43.128
6	63.484	55.932	47.702	45.189
7	75.424	55.426	55.896	51.474
8	82.332	61.988	57.901	57.285
9	84.060	76.399	65.320	62.226
10	93.559	79.707	70.682	69.673
11	91.413	84.895	76.281	75.523
12	98.960	91.474	83.091	83.404
13	108.380	97.727	89.545	90.220
14	116.405	105.161	96.056	96.189
15	128.595	107.036	102.469	102.092

Table A.6: Average MPTCP transfer times using the controller application with path delay

Transfers	Random	Hash-based	Number of flows	Round-robin
1	31.414	26.594	28.572	28.578
2	32.333	32.343	28.430	27.119
3	39.802	35.150	34.187	31.809
4	45.631	42.135	37.231	37.927
5	52.394	51.716	44.103	42.718
6	54.402	57.528	52.334	51.458
7	62.726	67.561	60.600	59.471
8	69.313	68.659	66.898	61.484
9	73.550	74.585	72.468	70.097
10	81.649	80.299	78.600	78.659
11	83.182	86.523	84.178	84.791
12	93.015	92.760	90.752	91.730
13	98.762	101.861	96.556	93.981
14	104.868	103.846	105.069	104.156
15	112.900	107.788	112.414	108.862



# Appendix B - Path computation algorithms evaluation results

This appendix presents all the additional figures and tables containing results from the path computation algorithms evaluation experiments, detailed in section 6.2.

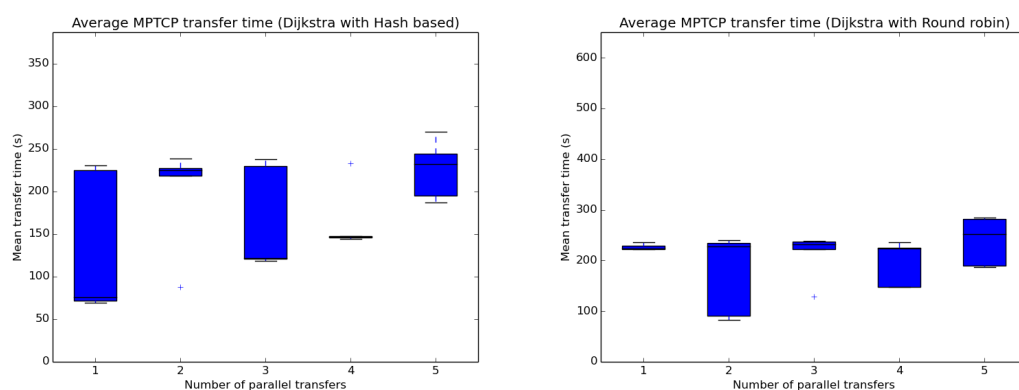


Figure B.1: Transfer times with Dijkstra's (Hash-based)      Figure B.2: Transfer times with Dijkstra's (Round-robin)

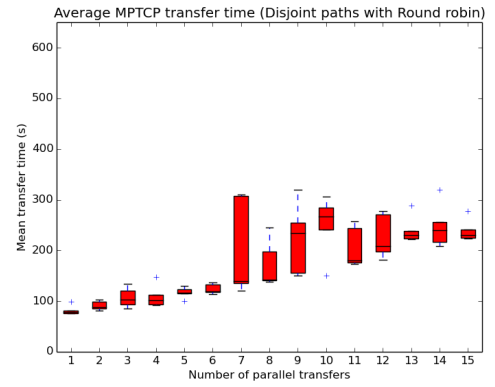
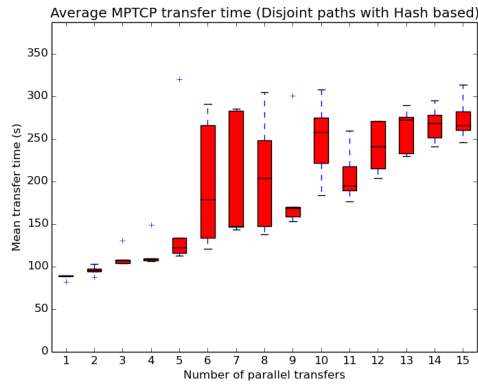


Figure B.3: Transfer times with Disjoint (Hash-based)

Figure B.4: Transfer times with Disjoint (Round-robin)

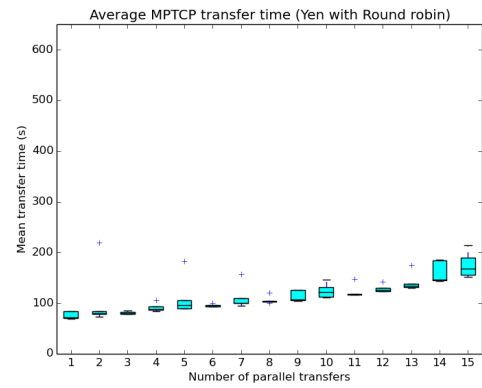
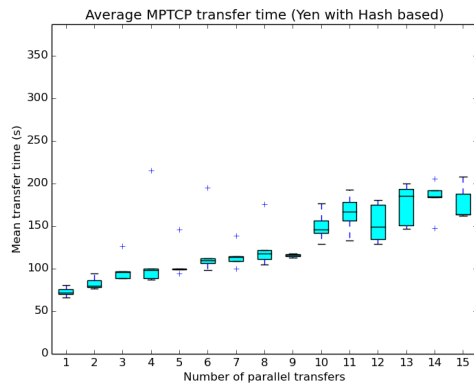


Figure B.5: Transfer times with Yen's (Hash-based)

Figure B.6: Transfer times with Yen's (Round-robin)

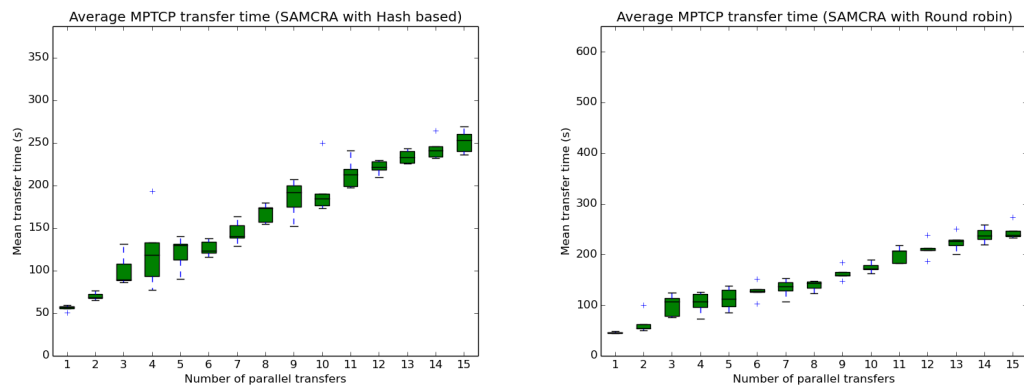


Figure B.7: Transfer times with SAM-  
CRA (Hash-based)

Figure B.8: Transfer times with SAM-  
CRA (Round-robin)

Table B.1: Average MPTCP transfer times (ms) for different path computation algorithms

Transfers	Dijkstra RR	Dijkstra Hash	Disjoint RR	Disjoint Hash	Yen RR	Yen Hash	SAMCRA RR	SAMCRA Hash
1	226.480	134.428	82.208	87.786	75.886	72.886	46.120	56.148
2	175.107	199.592	91.009	95.849	107.165	83.180	64.268	70.254
3	211.311	165.867	107.371	110.773	81.007	99.141	100.115	100.644
4	195.980	163.623	109.652	116.183	91.893	117.805	104.766	123.146
5	238.946	225.774	116.823	161.125	112.313	107.889	112.365	120.907
6	-	-	123.998	198.044	95.144	124.396	127.849	126.326
7	-	-	202.726	201.344	112.412	115.139	133.817	145.036
8	-	-	172.793	208.526	106.067	126.202	138.939	167.823
9	-	-	222.848	190.462	113.538	115.703	162.780	185.211
10	-	-	250.004	249.519	124.918	150.016	174.621	195.207
11	-	-	206.283	207.687	123.274	165.327	199.821	214.078
12	-	-	227.089	263.555	129.086	153.737	210.995	221.720
13	-	-	240.545	259.950	141.559	175.448	225.019	233.872
14	-	-	248.142	267.038	160.844	182.785	238.813	243.406
15	-	-	239.483	273.584	176.011	176.856	245.138	251.965

Table B.2: CDF throughput values (Mbps) with 1 parallel transfer

Algorithm	Scheduler	Minimum	25%	Median	75%	Maximum
Disjoint	Hash	4.237	15.252	30.320	58.075	81.945
Disjoint	RR	9.533	15.534	32.897	58.187	81.526
Dijkstra's	Hash	3.369	13.752	22.897	45.756	82.156
Dijkstra's	RR	3.316	4.993	13.315	21.363	25.461
Yen's	Hash	4.877	23.967	48.396	74.750	115.196
Yen's	RR	5.084	22.192	43.019	75.122	117.668
SAMCRA	Hash	4.438	31.775	60.787	87.970	119.837
SAMCRA	RR	13.981	34.953	74.565	102.527	119.182

Table B.3: CDF throughput values (Mbps) with 8 parallel transfers

Algorithm	Scheduler	Minimum	25%	Median	75%	Maximum
Disjoint	Hash	2.383	11.185	16.777	28.356	73.253
Disjoint	RR	3.264	12.710	18.516	32.708	69.053
Yen's	Hash	1.980	16.132	29.263	44.151	115.705
Yen's	RR	3.938	18.641	32.622	49.345	126.734
SAMCRA	Hash	0.448	16.029	23.069	32.577	90.126
SAMCRA	RR	0.449	19.284	26.631	35.349	89.878

Table B.4: CDF throughput values (Mbps) with 15 parallel transfers

Algorithm	Scheduler	Minimum	25%	Median	75%	Maximum
Disjoint	Hash	1.436	9.754	13.865	19.065	51.509
Disjoint	RR	2.943	9.908	14.717	22.221	52.429
Yen's	Hash	1.885	10.585	18.641	30.254	108.943
Yen's	RR	1.334	11.984	19.973	31.536	106.185
SAMCRA	Hash	0.435	10.755	14.538	20.214	81.659
SAMCRA	RR	0.427	11.336	15.534	21.409	86.037

# Appendix C - Application configuration instructions

## Main application configurations

The configurations for the main application can be found in the file named `65-pathcalculator.xml` in the `distribution-karaf/target/assembly/etc/opendaylight/karaf` directory. The application need to be at least be initialized once after being compiled with `maven`.

The following options can be configured:

- `scheduling-strategy`

**Description:** Flow strategy used by the application;

**Type:** String;

**Allowed values:** `rr` (Round-robin), `hash` (Hash-based), `flows` (Minimum flows), `random` (Random).

- `path-strategy`

**Description:** Used path computation algorithm;

**Type:** String;

**Allowed values:** `singlepath` (Dijkstra's), `multipath` (Yen's), `disjoint` (Disjoint paths), `samcra` (SAMCRA).

- `graph-refresh-delay`

**Description:** Delay (in ms) for updating the topology when a new link appears in the network (in the `TopologyChangeHandler` class);

**Type:** Integer.

- `local-flow-counters-duration`

**Description:** Duration (in ms) of the local flow counters in the minimum flows scheduler;

**Type:** Integer.

- `path-flows-idle-timeout`

**Description:** Timeout (in seconds) of the flows installed;

**Type:** Integer.

- `samcra-length-function`

**Description:** Function used to calculate the length of the paths in SAMCRA;

**Type:** String;

**Allowed values:** `mcp` (Multiple constrained path), `dc1c` (Delay constrained least cost - not fully implemented), `hcmb` (Hop constrained maximum bandwidth - not fully implemented).

- `samcra-metrics`

**Description:** Metrics used by SAMCRA;

**Type:** String, separated by commas;

**Allowed values:** `delay` (Delay), `availableBw` (Available Bandwidth), `hops` (Number of hops), `usedBw` (Used bandwidth).

- `samcra-constraints`

**Description:** Constraints used for each metric used by SAMCRA;

**Type:** Double, separated by commas.

- `min-samcra-paths`

**Description:** Number of paths that SAMCRA will always try to find;

**Type:** Integer.

- `samcra-path-timeout`

**Description:** Duration of the paths calculated by SAMCRA, until it computes new ones;

**Type:** Integer.

## Metrics Collector module configurations

The configurations for the metrics collector module can be found in the file named `62-metricscollector.xml` in the `distribution-karaf/target/assembly/etc/opendaylight/karaf` directory. The application need to be at least be initialized once after being compiled with `maven`.

The following options can be configured:

- `collect-delay`

**Description:** Perform link delay monitoring;

**Type:** Boolean.

- `controller-switch-delay-interval`

**Description:** Delay (in ms) for collecting the delay between the controller-switch connections;

**Type:** Integer.

- `link-delay-interval`

**Description:** Delay (in ms) for collecting the link-delay;

**Type:** Integer.

- `delay-packet-send-max-interval`

**Description:** Maximum delay (in ms) for sending delay monitoring packets;

**Type:** Integer.

- `delay-packet-send-min-interval`

**Description:** Minimum delay (in ms) for sending delay monitoring packets;

**Type:** Integer.

- `delay-packet-send-interval-strategy`

**Description:** Strategy for defining the interval between sending delay monitoring packets;

**Type:** String;

**Allowed values:** `static` (Static, using only the maximum interval), `random` (Randomized, with values between configured maximum and minimum)

- `collect-bandwidth`

**Description:** Perform used bandwidth monitoring;

**Type:** Boolean.

- `bandwidth-interval`

**Description:** Delay (in ms) for collecting the used bandwidth statistics;

**Type:** Integer.