Mendel University in Brno
Faculty of Business and Economics

Data path definition in software defined networks

Bachelor Thesis

Thesis supervisor:
Ing. Martin Pokorný, Ph.D.

Jan Zima

Brno 2016

Thesis assignment.

My thanks belong to God for His faithfulness, to my supervisor, Ing. Martin Pokorný, Ph.D., for his patient counseling, to my family for their constant support, and last but not least to my girlfriend for her encouragement.

Statutory declaration

Herewith I declare that I have written my final thesis: **Data path definition in software defined networks**
by myself and all sources and data used are quoted in the list of references. I agree that my work will be published in accordance with Section 47b of Act No. 111/1998 Sb. On Higher Education as amended thereafter and in accordance with the *Guidelines on the Publishing of University Student Theses.*
I am aware of the fact that my thesis is subject to Act No. 121/2000 Sb., the Copyright Act and that the Mendel University in Brno is entitled to close a licence agreement and use the results of my thesis as the "School Work" under the terms of Section 60 para. 1 of the Copyright Act.
Before closing a licence agreement on the use of my thesis with another person (subject) I undertake to request for a written statement of the university that the licence agreement in question is not in conflict with the legitimate interests of the university, and undertake to pay any contribution, if eligible, to the costs associated with the creation of the thesis, up to their actual amount.

In Brno on December 31, 2016 ....

Abstract

Zima, J. Data path definition in software defined networks. Bachelor thesis. Brno, 2016.

This thesis is concerned with visualization of OpenFlow-based software-defined networks, and definition of arbitrary traffic paths. For this purpose, an application called Visdan has been developed. It provides a visualization of the connected network, and allows the user to define custom traffic paths in the network using the graphical visualization. This work provides a broad explanation of OpenFlow-based networks and may serve as an introduction into software-defined networking. Visdan is an example of one of the approaches to programming applications for such networks.

Key words

software defined networking, sdn, openflow, visualization, path definition

Abstrakt

Zima, J. Definice datových cest v softwarově definovaných sítích. Bakalářská práce. Brno, 2016.

Tato závěrečná práce se zabývá viualizací softwarově definovaných sítí založených na OpenFlow a definicí vlastních datových cest. Za tímto účelem byla vyvinuta aplikace s názvem Visdan. Ta poskytuje vizualizaci připojené sítě a umožňuje uživateli definovat vlastní datové cesty v síti pomocí zmíněné grafické vizualizace. Tato práce nabízí obsáhlý popis sítí založených na OpenFlow, a může tak sloužit jako úvod do softwarově definovaných sítí. Visdan je ukázkou jednoho z přístupů k programování aplikací pro tyto sítě.

Klíčová slova

softwarově definované sítě, sdn, openflow, vizualizace, definice cesty

Contents

1 Introduction **13**
1.1 Aim and objectives 13

2 Reviews **15**
2.1 Previous academic works 15
2.2 Existing applications 20
2.3 Summary 23
2.4 Information sources 23
2.5 Documentation 28

3 Foundation **31**
3.1 Current networking 31
3.2 Software-defined networking 33
3.3 Tools 48

4 Application design **57**
4.1 Models of communication with the network 57
4.2 Application architecture 59
4.3 Model layer 60
4.4 View layer 63
4.5 Controller layer 64

5 Implementation **68**
5.1 Data retrieval 68
5.2 Data visualization 74
5.3 Path definition 78
5.4 Graphical user interface realization 80

6 Testing **84**
6.1 Virtual network infrastructure 84
6.2 Scenario 1: network visualization 84
6.3 Scenario 2: link utilization 88
6.4 Scenario 3: path definition via OpenFlow v. 1.0 89
6.5 Scenario 4: path definition via OpenFlow v. 1.3 90

7 Evaluation **93**

8 Conclusion **95**

9 Bibliography **96**
9.1 Literature review 96
9.2 Cited sources 100

Appendices 107

A Digital appendices 108

B OpenFlow Specification v. 1.3.0 109
B.1 Extensible match support 109
B.2 Multiple flow tables 110
B.3 Multiple flow tables illustration 111
B.4 Groups 114
B.5 Meters 116
B.6 Other improvements 117

C Mininet commands 119
C.1 Starting attributes 119
C.2 CLI commands 121

D HPE VAN SDN Controller REST API – Used structures 122
D.1 Data retrieval 122
D.2 Data pushing 127

E Network data – algorithms 130
E.1 Loading data 130
E.2 Updating data 133

F Tree-like data structure – algorithms 138
F.1 Creation 138
F.2 Drawing 140

G Mininet testing topology 143
G.1 Custom topology 143
G.2 Default topology 144

1 Introduction

Networking as it is usually done today has not significantly changed since the design settled down years ago. But as computing evolves, so does the way networks are used. Unfortunately, the development of networking technologies has been lagging behind compared to other technologies, such as server systems.

In modern cloud environments, a user asks for needed resources (for example computing power or storage space) and does not have to be concerned with how the resources are provisioned. Current networking devices are not meant for such applications, because they are based on rather static configurations, while the internet, where the user activity is ever-growing and ever-changing, is very dynamic.

Researchers have over time worked on various projects leading to better network automation, but none has got as much attention as software-defined networking has. The term software-defined networking or software-defined networks (SDN) is quite new and has been around for only a few years. Even though there is not an accepted-by-all definition of SDN and there are more approaches, networking is finally getting transformed into what the users need it to be.

The users need to focus more on the problems that are to be solved than on the ways of implementing the solutions. This approach to network control requires new tools for both automated operation and manual management of the network.

This thesis is concerned with facilitating basic management tasks performed by network administrators. It has been carried out in cooperation with the networking work group of Department of Informatics of Faculty of Business and Economics at Mendel University in Brno. It should serve as an introduction into the vibrant world of software-defined networking, and also show one way software-defined networks can be programmed.

1.1 Aim and objectives

The aim of this thesis is to create a software application that would provide a readable graphical visualization of a connected OpenFlow-based software-defined network, show the utilization of links between devices, and allow the user to conveniently define a custom traffic path between two end nodes in the network using a user-friendly graphical interface.

A review of source literature for and existing solutions concerned with the defined problem or relevant issues has to be worked out. To introduce the reader into software-defined networking and means involved in the thesis, a technological overview should be provided. Before implementing the application itself, user requirements have to be analyzed, and an application design has to be carried out based on the requirements. The application is then to be implemented according to the design. It has to be tested and its functionality verified in a virtual network,

and, if possible, in a physical network as well. At the end, the achieved results and their usefulness should be discussed.

2 Reviews

In this chapter, reviews that had to be done are described:

- a review of previous academic works
- a review of existing applications that serve a purpose similar to the one intended for the project of this thesis
- a review of useful information sources
- a review of relevant documentation

Before starting the work, it was necessary to build an informational base for studying related technologies and techniques. It was also necessary to search for existing solutions to find out whether it is meaningful to carry out the thesis project with the stated goals and whether any works exist that could be followed up.

2.1 Previous academic works

2.1.1 Sources

In the beginning, I searched for theses from Czech universities, which are listed on the website of Czech Ministry of Education, Youth, and Sports (Ministerstvo školství, mládeže a tělovýchovy, 2016). Many of the universities use the Czech central theses database (`http://theses.cz`), but since not all of them offer study programs concerned with computer science, I was only interested in theses from the following universities:

- Banking Institute College (Bankovní institut vysoká škola)
- College of Entrepreneurship and Law (Vysoká škola podnikání a práva)
- College of Polytechnics Jihlava (Vysoká škola polytechnická Jihlava)
- Comenius University in Bratislava (Univerzita Komenského v Bratislave)
- Czech University of Life Sciences Prague (Česká zemědělská univerzita v Praze)
- Charles University (Univerzita Karlova)
- Jan Evangelista Purkyně University in Ústí nad Labem (Univerzita Jana Evangelisty Purkyně v Ústí nad Labem)
- Masaryk University (Masarykova univerzita)
- Mendel University in Brno (Mendelova univerzita v Brně)
- Palacký University Olomouc (Univerzita Palackého v Olomouci)
- Silesian University in Opava (Slezská univerzita v Opavě)
- Tomas Bata University in Zlín (Univerzita Tomáše Bati ve Zlíně)
- University of Defence (Univerzita obrany)

- University of Economics, Prague (Vysoká škola ekonomická v Praze)
- University of Finance and Administration (Vysoká škola finanční a správní)
- University of Hradec Králové (Univerzita Hradec Králové)
- University of Chemistry and Technology, Prague (Vysoká škola chemicko-technologická v Praze)
- University of Ostrava (Ostravská univerzita)
- University of South Bohemia in České Budějovice (Jihočeská univerzita v Českých Budějovicích)
- University of West Bohemia (Západočeská univerzita v Plzni)
- VŠB – Technical University of Ostrava (Vysoká škola báňská - Technická univerzita Ostrava)

From schools not using this database, but also offering relevant study programs, these make their theses publicly available:

- Brno University of Technology (Vysoké učení technické v Brně)
  `https://dspace.vutbr.cz/`
- College of Logistics (Vysoká škola logistiky)
  `http://katalog.vslg.cz/eng/baze.htm`
- Czech Technical University in Prague (Česke vysoké učení technické v Praze)
  `https://dspace.cvut.cz/`
- European Polytechnic Institute (Evropský polytechnický institut)
  `https://www.edukomplex.cz/index.php/home/zaverecne-prace`
- University of Pardubice (Univerzita Pardubice)
  `http://dspace.upce.cz/`

As for the remaining universities, these may also have relevant theses in their archives, but do not make them publicly available:

- Karel Englis College (Vysoká škola Karla Engliše)
- Metropolitan University Prague (Metropolitní univerzita Praha)
- Technical University of Liberec (Technická univerzita v Liberci)
- Unicorn College (Unicorn College)
- University of New York in Prague (University of New York in Prague)

I also searched the Slovak central theses database (`http://cms.crzp.sk`). For all of these registers, I used following keywords to find relevant theses:

- software defined networking
- software defined networks
- sdn

- openflow

For a more general search and also to find theses from foreign universities, I used Google (`https://www.google.com`) to search the internet with the following keywords:

- software defined networking thesis
- software defined networking visualization thesis
- software defined networking path thesis
- software defined networking path definition thesis
- openflow thesis

Since keywords *software defined networking thesis* and *openflow thesis* are very general for global searching and produce results from diverse areas, more efforts were put in examining the more specific keywords. I always investigated at least the first 50 results of each search. None of the found papers is older than 5 years since the relevant technologies have been on the rise during the past few years.

2.1.2 Results

Interactive Monitoring, Visualization, and Configuration of OpenFlow-Based SDN – Pedro Heleno Isolani
Isolani (2015) is concerned with the impact of SDN control traffic and SDN-specific metrics on the overall network performance. First, he analyzes the control traffic and then designs and implements a management tool, which serves mainly for network visualization and monitoring of resource usage and control channel load.

SDN Interactive Manager, an application created as an outcome of this work, uses the D3.js JavaScript library for network visualization, which also depicts the level of control or data traffic on individual switches. This technique might be used to quickly identify which paths are highly-utilized and which have free capacity. The application also provides interactive charts with network statistics showing online resource usage.

The application was created for a specifically modified version of the Floodlight controller. The source code of the application is available, but it does not seem to be under active development anymore. Except for network visualization, which might serve for inspiration, the features of this application are at this point of no significant use for my project.

Extension of SDN platform available at FIIT STU – Michal Palatinus
Palatinus (2015) contributes to Unifycore, an existing project that builds a GPRS network architecture on software-defined networking, with a management tool. He designs and implements a web-based manager that, in a basic manner, provides network visualization, management of flow entries, switch statistics, and a few other

functions related to specifics of the GPRS network. The thesis is written in Slovak.

The visualization is realized using the vis.js JavaScript library, but it does not provide any interactions with the network. Flow entries are defined in a low-level manner by manually setting all match rules and actions. The application also provides traffic statistics for switches and their interfaces.

Since the relevant outcomes are elementary in their functionality, this thesis does not seem to offer much to base my project upon.

ViewNet: A Visualization Tool for Software Defined Networks – Prerna Ramachandra
Ramachandra (2014) has as well decided to create a visualization tool for software defined network. Unfortunately, the thesis is not freely available, and it seems that the project has been discontinued because I have not been able to find any information about the application.

Interactive Visualization of Software Defined Networks – Andreas Schmidt
Schmidt (2013) is concerned with interactive monitoring of software-defined networks. He proposes an architecture of an application, implements it, and evaluates. The application provides an interactive visualization of the monitored network together with detailed information and statistics for connected devices.

The application called SDN-Visualization has two parts. A server side, which cooperates with the controller, and a client side, which uses the server side to approach the network. It was designed for the Floodlight controller, but can be extended to support other controllers. The source code of the application is available, but it does not seem to be under active development anymore.

The visualization is realized through the D3.js library, but while it is effective, it does not seem very useful for interaction with the network. On the other hand, the way the detailed statistics are done might be potentially made use of in my project as well as the two-tier architecture design.

To better orient in the variety of available OpenFlow controllers and other support tools, the following thesis comes in handy:

Review of Available Tools for Control Plane of Software Defined Networks – Matej Leitner
Leitner (2015) presents a review of currently available controllers and other support tools operating the control plane of a software defined network. Open-source as well as commercial products are described and then separately compared. In the end, an experiment is performed to test a controller and a configuration tool with a virtual network. The thesis is written in Slovak.

As for other theses related to SDN, their topics include, for example, implementation of firewall, failure recovery, or defense against DDoS attacks, while some of them

only present SDN as a technology with results of simple experiments. Considering possible future extensions of my project, I found two topics particularly interesting – load balancing and high-level policy declaration. I mention theses concerned with these topics below, but do not describe specific useful parts since these features are not actual for my thesis at this point.

Following theses deal with load balancing, which might be used for automation of network control:

Load Balancing in OpenFlow Networks – Petr Marciniak
Marciniak (2013) is concerned with implementing a tool for load balancing in OpenFlow networks. He explains the best current load balancing practices and proposes several algorithms for the software-defined approach to load balancing – a Floyd-Warshall algorithm with adjusted weights and two variants of Dijkstra's shortest path algorithm. These algorithms are implemented and evaluated.

Dynamic Load Balancing in Software-Defined Networks – Martí Boada Navarro
Navarro (2014) aims to exploit the means of SDN in order to utilize network resources efficiently. He describes a load balancing algorithm called MPLS Dynamic Load Balancing and then presents his modification of this algorithm, which makes better use of the advantages and specific features of SDN. The algorithm is implemented, tested, and the results are evaluated.

Path Computation Enhancement in SDN Networks – Tim Huang
Huang (2015) focuses on load balancing in data center networks and its specifics. He proposes a path computation algorithm for learning all shortest paths in a network and a path selection algorithm for choosing the best of these paths. The choice is influenced by the congestion status of the individual paths. The algorithms are implemented and their results analyzed. The thesis is also concerned with hybrid SDN networks and proposes a solution to support their proper operation.

Load Balancing in Real Software Defined Networks – Gonçalo Miguel Alves Semedo
Semedo (2014) proposes an approach of combining several load balancing algorithms in order to achieve better efficiency. Depending on the type of the user request, the best server is chosen and then the best path to this server is determined. Semedo distinguishes three user request types - requiring low latency, high bandwidth, or a server with a lot of available processing power. The final solution is tested in a large-scale testing network environment with real equipment and users, and the testing is then evaluated.

CAFFEINE: Congestion Avoidance for Fairness & Efficiency in Network Entities – Pattanapoom Phinjirapong
Phinjirapong (2015) makes use of the capabilities of SDN and creates an algorithm for dynamic routing with the goal to fully and evenly utilize network capacity. His algorithm, CAFFEINE, constantly monitors the network state and then finds the

path with maximum available bandwidth. To find this path, a modified widest path algorithm constrained by path lengths is used. In the end, the solution is tested and evaluated.

Dynamic and performance driven control for OpenFlow networks – Tim Herinckx
Herinckx (2013) focuses on creating a load balancing mechanism with dynamic multipath routing. Based on the information about utilization of links, his algorithm temporarily reroutes certain flows to unburden the affected links. When necessary, the algorithm can decompose flow entries in switches and consequently route the flows more granularly over different paths. After the emergency situation is over, the flow entries get aggregated into more general rules again. Performance of this solution is measured and evaluated.

For more convenient implementation of requests, there are authors proposing network management based on declaring high-level policies instead of low-level flow entries. Following theses are concerned with this issue:

A Network Control Language for OpenFlow Networks – Dávid Antolík
Antolík (2013) focuses on developing a proprietary language for high-level declaration of policies in OpenFlow-driven networks. The specification of the control language and its elements is followed by implementation of its interpreter. The solution is then tested and evaluated. The thesis is written in Slovak.

Using Software-Defined Networking to Improve Campus, Transport and Future Internet Architectures – Adrian Lara
Lara (2015) focuses in his dissertation on making use of SDN on several levels of networking. After an introduction to SDN and OpenFlow, he presents a policy-based security management tool for networks of campus scale, a framework for dynamic network provisioning at WAN scale, and at the Internet scale is concerned with intra-domain cut-through switching and inter-domain routing using cut-through switching.

2.2 Existing applications

2.2.1 Sources

I searched for software tools that would provide a visualization of a software-defined network or the ability to define custom flow paths. First I explored HPE SDN App Store (Hewlett Packard Enterprise, 2016b), which is currently the only known gathering place for SDN applications, but it is limited to applications designed for the HPE SDN VAN Controller or newly also the OpenDaylight controller. Then I used Google (`https://www.google.com`) to search the internet with the following keywords:

- openflow visualization

- openflow path definition
- openflow flow management application
- openflow flow manager
- software defined networking visualization
- software defined networking path definition
- software defined networking flow management application
- software defined networking flow manager

2.2.2 Results

Avior
Avior (Marist SDN Lab, 2016) is a web-based management tool for OpenFlow networks. As for the main functionality, it provides a basic topology visualization, a static flow-entry pusher, and dynamic statistics about the connected controller, switches, and end-nodes. The application is intended to support any controller (currently supports Floodlight and OpenDaylight) and to work on any platform. The source code of the application is available, but it does not seem to be under active development anymore.

Regarding functionality relevant to this thesis, the provided network visualization is simple without any interactions. The flow-entry pusher serves as a convenient interface for the low-level operation, but it does not provide any abstraction. I do not therefore see much in this application to make use of for my project.

Brocade Flow Manager
Brocade Flow Manager (Brocade, 2016a) is an application tailored for Brocade SDN Controller and it integrates into the graphical user interface of the controller. As for its main functionality, it provides a visualization of the network topology, detailed information about switches, management of their flow entries, and the ability to define custom data paths in the visualization. Since this application is a commercial product, the source code is not available.

This application is a commercial product and its source code is not available. It could therefore serve only as an inspiration for my project, but not as something to build upon.

HPE Network Visualizer
HPE Network Visualizer (Hewlett Packard Enterprise, 2016a) is an application tailored for HPE SDN VAN Controller and it integrates into the graphical user interface of the controller. The application offers real-time network monitoring and dynamic traffic capture for brisk problem diagnosis and fixing. A topology monitor is also provided as one of the features. It mainly visualizes the connected network, displays detailed information about its devices, and allows for tracing packet flows

between nodes. Since this application is a commercial product, the source code is not available.

This application is a commercial product and its source code is not available. It could therefore serve only as an inspiration for my project, but not as something to build upon.

Hyperglance
Hyperglance (Hyperglance, 2016) is a professional commercial software for integrated monitoring and management of various platforms, such as cloud, virtualization, or networking, running on technologies like Amazon Web Services, Open Stack, or VMware vSphere. It provides complex visualization, statistics, diagnostics, and management tools for all controlled resources and across them. In current version 4.1, the application is accessible through a web browser and does not need any client software, but this version does not yet support SDN. For SDN, an older thick-client version 3.5 must be used that has support for HPE SDN VAN Controller and for the OpenDaylight controller. Since this application is a commercial product, the source code is not available.

This application is a commercial product and its source code is not available. It could therefore serve only as an inspiration for my project, but not as something to build upon.

Infinite Flow Manager
Infinite Flow Manager (Infinite Computer Solutions, 2016) is a single-purpose application with user-friendly GUI that allows management of flow entries in switches – viewing, adding, and removing. It is a web-based application that can be used with various controllers. The source code of the application is available for the basic version. A Pro version of the application exists that should also provide a visualization of the network topology and a few other features, but it is not freely available.

Since the only functionality of the available version of the application is pushing flow-entries into switches, there is not much I might make use of in my work.

JSFlowViz
JSFlowViz (Wallaschek, 2014) is a simple application that visualizes OpenFlow networks and dataflows. It is designed as a plugin for the Beacon controller. The source of the application is available, but it is not under active development anymore and is meant only as a proof-of-concept, not for production use.

This application offers only a basic functionality with possible bugs because of the ceased development. Therefore it does not look as a significant contribution for my project.

2.3 Summary

Concerning academic works, there are researchers who have tackled the issue of visualization of software-defined networks and come up with solutions of various level of complexity. However, none of the ones I found has used the visualized network topology to manipulate paths for data flows. Some researchers have been concerned with controlling traffic and changing its routes, but on a dynamic and automated level that only allows limited human intervention. On the other hand, this functionality might be well utilized to support decisions and actions of network administrators. Other researchers have focused on developing high-level approaches to programming software-defined networks. Such solutions might beneficially complement the approach of defining data paths using the network topology visualization, for example when defining policies with a rather global effect.

As for existing software applications, visualization has interested several developers, who have usually used it as a gateway for numerous monitoring or management instruments. It is also common that SDN controllers implement a simple visualization module in their graphical user interface. As for the definition of data paths, Infinite Flow Manager provides the functionality of manipulating flow entries in switches in a very simple way, but does not provide any kind of visualization of the network. Two of the applications – Hyperglance and Brocade Flow Manager – accommodate both visualization and its utilization for manipulation of the data path. Hyperglance is a fairly complex project, whose functionality and cost overly exceed the needs expressed for this thesis. Brocade Flow Manager provides the main requested functionality, but it cannot be used with any other controller than its parent one nor can its source code be accessed for inspiration.

Overall, I conclude that it is meaningful to carry out this thesis and its project as intended. I have not found a project that would be concerned with both visualization and interactive data path definition while at the same time being open for others to investigate it and improve. Additionally, there are projects that could be used in the future to help enhance the functionality of the proposed application in order to make it a more mature management tool for network administrators.

2.4 Information sources

To find books concerned with software-defined networking and relevant subjects, I searched the Amazon store (`https://www.amazon.com`) with following keywords:

- software defined networking
- sdn
- openflow

I found quite many books and based on their descriptions and tables of content tried to categorize them for better orientation.

2.4.1 Books theoretically-oriented, with a broad range of related topics

Software Defined Networks: A Comprehensive Approach – Paul Göransson and Chuck Black
Göransson and Black (2014) explain the background of SDN together with the needs for its inception and provide technical details and practical examples. They discuss the integration of SDN technologies into networks and the choice and development of its applications. The book also helps make a deliberate decision and explain the benefits and risks to non-technical managers.

Software Defined Networking (SDN): Anatomy of OpenFlow Volume I – Doug Marschke, Jeff Doyle and Pete Moyer
Marschke and his colleagues (2015) are concerned with what SDN is and its importance today, but mostly with its protocols - especially OpenFlow. They present the history of this protocol and its current status along with its outlook and use cases. Typical use cases are also provided for software defined networks based on technologies other than OpenFlow.

SDN: Software Defined Networks: An Authoritative Review of Network Programmability Technologies – Thomas D. Nadeau and Key Gray
Nadeau and Gray (2013) bring a practical view on SDN. They begin with a broad description of the key principle of separated data and control planes, and then continue with other fundamental parts of this networking concept such as OpenFlow, controllers, or programmability. The book includes several use cases for various scenarios.

Software Defined Networking (SDN) – a definitive guide – Rajesh Kumar Sundararajan
Sundararajan (2013) offers a presentation of the SDN technology, situations for which it is suitable, and practical reasons for using it as a solution in those situations. This book aims to present the information in a clear and simple way, and therefore also includes an explanation of different technological terms.

2.4.2 Books theoretically-oriented, primarily focused on specific subjects

SDN: Defining a Strategic, Business-Focussed Architecture – James J Connolly
Connolly (2015) focuses on SDN in relation to business needs of network operators and gives an overview of the many capabilities that are becoming possible. He explains the new technological concepts, adds business reasons for the change, and discusses what benefits SDN brings to solving identified business problems. The

book also contains chapters on networking components and how they can contribute to the business.

SDN and NFV Simplified: A Visual Guide to Understanding Software Defined Networks and Network Function Virtualization – Jim Doherty
Doherty (2016) takes a very specific story-telling approach. Using illustrations and visual expressions as the first means of communications, he wants to explain to people who are not networking experts why SDN and NFV matter, how they work, where they are used, and what problems they solve.

Software Defined Networking: Design and Deployment – Patricia A. Morreale and James M. Anderson
Morreale and Anderson (2014) provide a perspective on business and technology motivations for considering SDN solutions. The book addresses SDN principles and OpenFlow, explains the importance of virtualization of servers and networks, and discusses the impact of SDN implementation on service providers, legacy networks, and network vendors. It also investigates the initial SDN implementation at Google.

Introduction to Software Defined Networking – OpenFlow & VxLAN – Vishal Shukla
Shukla (2013) concentrates on two SDN implementations – OpenFlow and VxLAN. He gives an overview of SDN and then explains the principles of these technologies and their essential parts, such as events, or packets. The book is intended for educational purposes and therefore contains detailed explanations as well as case studies for both considered implementations.

Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud – William Stallings
Stallings (2015) points out five mutually related key technologies that are changing modern networks – SDN, NFV, QoE, IoT, and Cloud. First, he provides a review of current network ecosystems and the challenges they have to face, and then advances with a comprehensive description of each of the elect technologies. An analysis of emerging security issues follows as well as a discussion of networking careers.

2.4.3 Books practically-oriented, with a greater portion of examples

Software Defined Networking with OpenFlow – Siamak Azodolmolky
Azodolmolky (2013) provides an introduction to OpenFlow with the focus on its essential building blocks. The book covers the basics of setting up a laboratory environment or developing sample applications, and in addition offers an insight into existing controllers and available open-source resources.

SDN, Openflow, and Open vSwitch: Pocket Primer – Thomas F. Herbert
Herbert (2014) in his book presents SDN and discusses the transformation it causes in traditional networking. The introduction is followed by an exploration of the

internals of SDN switch and controller implementations, which is accompanied by practical examples and detailed illustrations. The book also includes a DVD with instructional videos and sample source code.

Network Innovation through OpenFlow and SDN: Principles and Design – Fei Hu
Hu (2014) addresses the principles and design of a software defined network. He gives an introduction of basic principles of SDN, OpenFlow, and their applications and then moves on to explain the design process. He shows how to create networks that are easy to design, more economical to build and maintain, and more flexible than the traditional ones. The book also presents network attacks that could occur in OpenFlow networks together with possible solutions to overcome them.

OpenFlow Cookbook – Kingston Smiler. S
Smiler. S (2015) deeply focuses on OpenFlow with development being the core subject. In 110 recipes to design and develop OpenFlow switches and controllers, the book approaches SDN from a low-level standpoint. It touches all aspects necessary for implementing the control software for these devices and provides example source code in every part.

SDN and OpenFlow for beginners with hands on labs – Vivek Tiwari
Tiwari (2013) brings a concise and practical approach to learning SDN. He explains what it is, how it works, what the relationship between SDN and OpenFlow is, and how it affects networks. Practical guidance and hands-on experience through various lab tasks are also included.

2.4.4 Chosen books

From all these books I had to choose a few that would serve as the initial guides and help me orient in the various elements and approaches of software-defined networking. Based on the descriptions and user reviews at Amazon, I chose one theoretically-oriented and one practically-oriented:

Software Defined Networks: A Comprehensive Approach – Paul Göransson and Chuck Black
Göransson and Black (2014) bring a complex approach to SDN, in which they cover all elementary aspects and many more in a easily readable form. These chapters are key to grasp the essentials of software-defined networking:

- Chapter 4 – is concerned with the principles and operation of SDN and its components.
- Chapter 5 – in great detail describes the building blocks of OpenFlow 1.0 and explains the changes and additions in subsequent versions of the specification up to version 1.3.

- Chapter 6 – presents alternative approaches to SDN that also tend to improve networks, but are not derived from the concepts behind OpenFlow-based SDN.
- Chapter 10 – is devoted to SDN applications. The basic kinds of applications are distinguished, and a sample application is explained with parts of the source code. Also, some controllers are introduced as well as a few use cases for SDN applications.

In other chapters, the interested reader can find more about current networking and the evolution that led to SDN, about the use of SDN in various environments, about what impacts it has on businesses and what are its prospects, and there is also a review of organizations influencing the development of SDN and a review of available open-source components.

Software Defined Networking with OpenFlow – Siamak Azodolmolky
Azodolmolky (2013) provides an introduction to OpenFlow and the tools used for application development. He covers the basics of OpenFlow, implementations of switches and controllers, preparation of laboratory environment, breaks down two sample applications, and the last chapters are devoted to network slicing, OpenFlow in cloud computing and to a review of open-source resources.

Unfortunately, I have not found this book very useful. It is sometimes difficult to orient in – the introduction to OpenFlow is uneasy to comprehend as it is short and quickly jumps into low-level details, and the practical part contains only a few examples, which are basically only explained examples shipped with particular software.

2.4.5 Other sources

To find recent news or explanations of various technologies or technological terms, the following websites provide useful content related not only to software-defined networking:

SDxCentral
SDxCentral (SDNCentral, 2016) is a portal focused on software-defined infrastructure technologies. News, research, and analyses are concerned with topics like SDN, NFV, or cloud and there is also a comprehensive database of resources for them.

TechTarget
TechTarget (2016) is a platform that connects technology buyers with the technical content they seek and providers of the technology. Its sub-websites bring a diverse spectrum of materials focused on a specific topic that are written by experts in that field. Relevant to this thesis is mainly the SearchSDN site.

Concerning the programming side of the thesis, the following book and website form priceless resources:

Dive Into Python 3 – Mark Pilgrim
As a base for the implementation of the application, the book of Mark Pilgrim (2009) provides a comprehensive guide through all the basic elements of Python 3. It uses many sample code snippets accompanied by explanations to offer a practical and enjoyable approach. To the reader's advantage, the book is licensed under the Creative Commons license, which means it is freely available.

Stack Overflow
Stack Overflow (Stack Exchange, 2016) is a community platform for sharing knowledge among programmers that is based upon a questions-and-answers style. Solutions or suggestions for many common problems may be found in the answers.

2.5 Documentation

OpenFlow
In the technical library of Open Networking Foundation (2016a), many important or useful papers may be found. The following are relevant to this thesis:

- OpenFlow Switch Specification – defines the requirements for an OpenFlow switch as well as the OpenFlow protocol and its messages.
- SDN Architecture – is concerned with the architecture of software-defined networking as viewed by ONF.

HPE VAN SDN Controller
In the information library of Hewlett Packard Enterprise (2016c), datasheets, specifications, and guides for its products can be accessed. The following documents regarding the controller are essential, and their content fully reflects their name:

- HPE VAN SDN Controller Installation Guide
- HPE VAN SDN Controller Administrator Guide
- HPE VAN SDN Controller Troubleshooting Guide
- HPE VAN SDN Controller Programming Guide
- HPE VAN SDN Controller REST API Reference
- HPE VAN SDN Controller Java API Reference

Documents regarding additional HPE SDN applications for the controller are also present in the library.

Mininet documentation
Mininet's documentation (Mininet, 2016) provides introductory materials as well as more thorough ones. Following sections are useful for easy beginning:

- Download – this page is concerned with installation options and their descriptions.

- Introduction to Mininet – presents the essentials of what Mininet is and how it works. It also contains an introduction to the Python API.
- Walkthrough – serves as a basic demonstration of Mininet. Many commands are shown and explained on examples.
- Mininet Python API Reference – is a documentation of the Python API, which comes in handy when writing custom Python scripts for advanced functionality.

Python documentation The documentation for Python 3 (Python Software Foundation, 2016) is complex and consists of several parts. The essential sections are:

- Python Setup and Usage – documents how to install Python on different platforms and how to use its command line.
- Tutorial – provides an informal introduction to basic features that are most common in Python programs. Although going through all the sections is advisable for a Python newcomer, I would consider the following subsections as fundamental:
  - An Informal Introduction to Python
  - More Control Flow Tools
  - Data Structures
  - Classes
- Library reference – represents a more comprehensive manual to Python. It includes the exact syntax and semantics of the language.

Because Python usually uses programming-style conventions that sometimes significantly differ from those of other language, the style guide might come handy (Reitz, 2016).

For an interactive and more enjoyable approach to learning Python, Codecademy (2016) offers practical online courses.

HP SDN Client documentation
The documentation (Tucker, 2014) for this Python module consists of three main parts:

- Installation – describes the possible ways of installing the module.
- Quickstart – explains how to put the module to use.
- API Documentation – provides an overview of all methods and data types. The methods are briefly described together with their attributes.

Unfortunately, the documentation misses examples except for the short introduction.

Qt/PyQt documentation
The documentation of Qt (The Qt Company, 2016a) is comprehensive and covers

every part of the framework from several points of view. Key to this work is the section on widget-based user interfaces (The Qt Company, 2016b) and the pages to which it leads. Particular classes will be mentioned in the description of the development process, and the interested reader can easily find their specification in the documentation.

As for PyQt documentation (Riverbank Computing, 2015), it refers to the documentation of Qt when it comes to class references since the syntax is essentially the same. Concerning the specifics of PyQt, the following sections of the documentation should be seen:

- Installing PyQt5
- Support for Signals and Slots – signals and slots are a key Qt feature for communication between objects. This section explains how they are used in PyQt.

3 Foundation

3.1 Current networking

3.1.1 Architecture

The networking of these days is almost exclusively based upon the TCP/IP protocol architecture. It defines a stack of four layers of protocols (Microsoft, 2016):

- **Network interface layer** – provides the means for transmitting signals over various mediums between hardware devices. Its protocols are not defined by the architecture since TCP/IP should be independent of the low-level access methods and be able to connect different network types.
- **Internet layer** – takes care of getting data packets from one end node to another by routing them over and between physical networks. The only protocol used for transferring data is IP.
- **Transport layer** – establishes a communication between two end hosts. It uses either the TCP protocol for connection-oriented and reliable communication, or the UDP protocol for much simpler and faster, but connectionless and unguaranteed communication.
- **Application layer** – provides applications with the access to services of lower layers. It defines protocols applications use to exchange data, such as HTTP, FTP, SMTP, and many more.

For a more comprehensive approach to network communication, there is OSI Reference Model. Through its seven layers, it precisely describes the interaction of hardware and software components, and how each of them involves in the communication (Kozierok, 2005b).

More information on both TCP/IP and OSI architectures can be found in *The TCP/IP Guide* (Kozierok, 2005a).

3.1.2 Traffic control

For traffic control, mostly the bottom layers of the OSI model are involved. Layer 1 is concerned with sending data as signals over physical media and is not engaged in traffic control. Layer 2 is concerned with connecting hardware devices and transmitting data between them. Its information about network interface addresses is used for forwarding traffic in local networks. Layer 3 is concerned with dividing networks into subnetworks and connecting them. Its information about logical addressing is used for routing the traffic across networks (Microsoft, 2014). Higher-level information, such as application port numbers or even application data, may also be used for forwarding decisions, usually in specific applications like firewalls or load balancers (Göransson and Black, 2014, p. 17).

There are two elementary kinds of devices used for directing traffic in or between networks – switches and routers. Switches are based on a specifically designed hardware (application-specific integrated circuits – ASICs) and typically used for connecting devices in Layer 2, where the destination address of a packet is the only information influencing its output interface. Due to the hardware implementation of the functionality, when a data packet arrives at an interface, the switch is able to perform the lookups in the forwarding table at line rate, which ensures quick forwarding of the packet (Differences between Layer ..., 2016).

Routers are commonly based on a specialized software running on general-purpose chips and are typically used for connecting networks in Layer 3. To find the best way to the destination network of a packet, routers use routing protocols that take care of cooperation of networking devices and consequently fill the routing table of the router. Unfortunately, the software nature of routers makes the traffic processing slower and the transmission of packets is then delayed (Differences between Layer ..., 2016).

There are also switches capable of handling Layer 3 or even Layer 4 information. The first time a multilayer switch encounters a packet with a particular source-destination pattern, it uses software routing functions to process it and then caches the result. The next time a packet with the same source-destination pattern arrives at the switch, a quick hardware lookup in the cache is made to determine how to handle the packet (Gijare, 2004). Designed for backbone switches with high demands, there are also more advanced technologies, like Cisco Express Forwarding, that do not use a cache, but instead translate the whole routing table into a form processable by hardware forwarding mechanisms. This allows even for the first packet to be processed by hardware (Cisco, 2016).

Not considering the implementation details and additional functions, switches and routers work basically with the same goal – they analyze a certain part of the packet header to determine where to send the packet. For the sake of simplicity, in this work, I will use the name *switch* for networking devices that direct traffic.

3.1.3 Drawbacks

Contemporary networks have several stumbling blocks in both their management and their operation, which make it challenging to control them. As for network management, networking devices have to be configured on a per-device basis using vendor-specific proprietary interfaces. While network administrators need to define high-level policies and apply them over the whole network, these interfaces only allow low-level configuration of individual devices (Kim and Feamster, 2013). And although tools for centralized management exist, they serve rather for monitoring of the network than for its configuration as a whole (Rouse, 2013).

Concerning network operation, typical networking devices use routing protocols to fill their forwarding tables, but may also allow for network administrators to manually configure additional rules. These rules may, for example, provide application port filtering or different treatment for particular quality-of-service classes. Unfortunately, there is no protocol to automatically distribute these more complex policies over the network (Göransson and Black, 2014, p. 17).

With packet forwarding based only on destination addresses or statically defined rules, the network cannot react to the dynamics of the traffic or to the occurring abnormalities. Be it peak loads, applications with high demands for Quality of Service, or applications requiring high bandwidth, with the static setting, the network has no instruments to appropriately utilize its resources unless equipped with specialized devices like load balancers.

To be fitted for demands of modern deployments, in both campuses and data centers, a network should provide means for automation, so it could react to occurring events on its own and efficiently use available resource while ensuring resilience. Such network should also be virtualized in order to provide a high-level abstraction for convenient management of the network regardless of the underlying physical layer and its specifics (Göransson and Black, 2014, p. 32–34).

Software-defined networking, further described in the next chapter, is bound to bring a solution to the various problems networking is facing today.

3.2 Software-defined networking

This thesis is based upon Open SDN, an approach that is most often connected with software-defined networking and which is sometimes also called revolutionary. It emerged from Clean Slate Program at Stanford University and was created as a solution to today's networking problems and demands with current technological advancement in mind while forgetting about the legacy network design and its constraints (Clean Slate Design …, 2016). It is a non-proprietary approach promoted by Open Networking Foundation (ONF), an organization responsible for the development of this approach and its standards. Members of the foundation include technological leaders such as Google, Facebook, Yahoo!, Microsoft, and other. More information about ONF can be found on its website (Open Networking Foundation, 2016b) or in *What is ONF?* (Open Networking Foundation, 2016c).

Sections dedicated to this approach are followed by brief descriptions of two alternative approaches.

3.2.1 Characteristics and principles

Open SDN is based on four principles (Göransson and Black, 2014, p. 59–61). The first is a separation of the forwarding plane (also called data plane), which takes

care of traffic processing, from the control plane, which examines the network and defines rules for forwarding decisions of the data plane. This is a prerequisite for the second principle – logically centralized control – that subsequently leads to simpler networking devices. Such devices are deprived of their own intelligence as they only follow rules defined by the central control plane. The third principle is network programmability.

Shenker (2011) stated that Open SDN is all about three abstractions – of distribution, forwarding, and configuration, which allow convenient automation of the network. These abstractions provide the programmer with a global view of the network independent of its complex nature, allow unified control of networking devices from different vendors and with various functionality, and enable the programmer to focus more on the goals than on the way a physical network will implement them, which is the purpose of virtualization.

The last principle is openness of interfaces. Since standard and non-proprietary interfaces pose no obstacles, they help create competition and encourage both vendors and the community to develop new solutions. This should eventually accelerate the development and innovation of networking technologies.

More details about the architecture of Open SDN can be found in its specification (Open Networking Foundation, 2016d, p. 12–22).

3.2.2 Components

A software-defined network, as viewed by this approach, has tree main components or layers as depicted in Figure 1. An infrastructure layer comprised of networking devices, a control layer represented by a logically centralized controller, and an application layer with network applications (Göransson and Black, 2014, p. 61–64). These components are further explained in following sections.

3.2.3 OpenFlow

Concerning this section, I based as much information as possible on offical materials by ONF and extended it primarily by information from the book *Software Defined Networks: A Comprehensive Approach* (Göransson and Black, 2014), which provides insightful explanations.

The OpenFlow standard is a key product of Open Networking Foundation and the first standard for the southbound interface. It defines the communication interface between the control plane and the data plane that allows direct access to and manipulation of the forwarding layer of networking devices (Open Networking Foundation, 2016e).

The standard and its elements are described in OpenFlow Switch Specification. The specification covers the functional requirements of an OpenFlow-capable network-

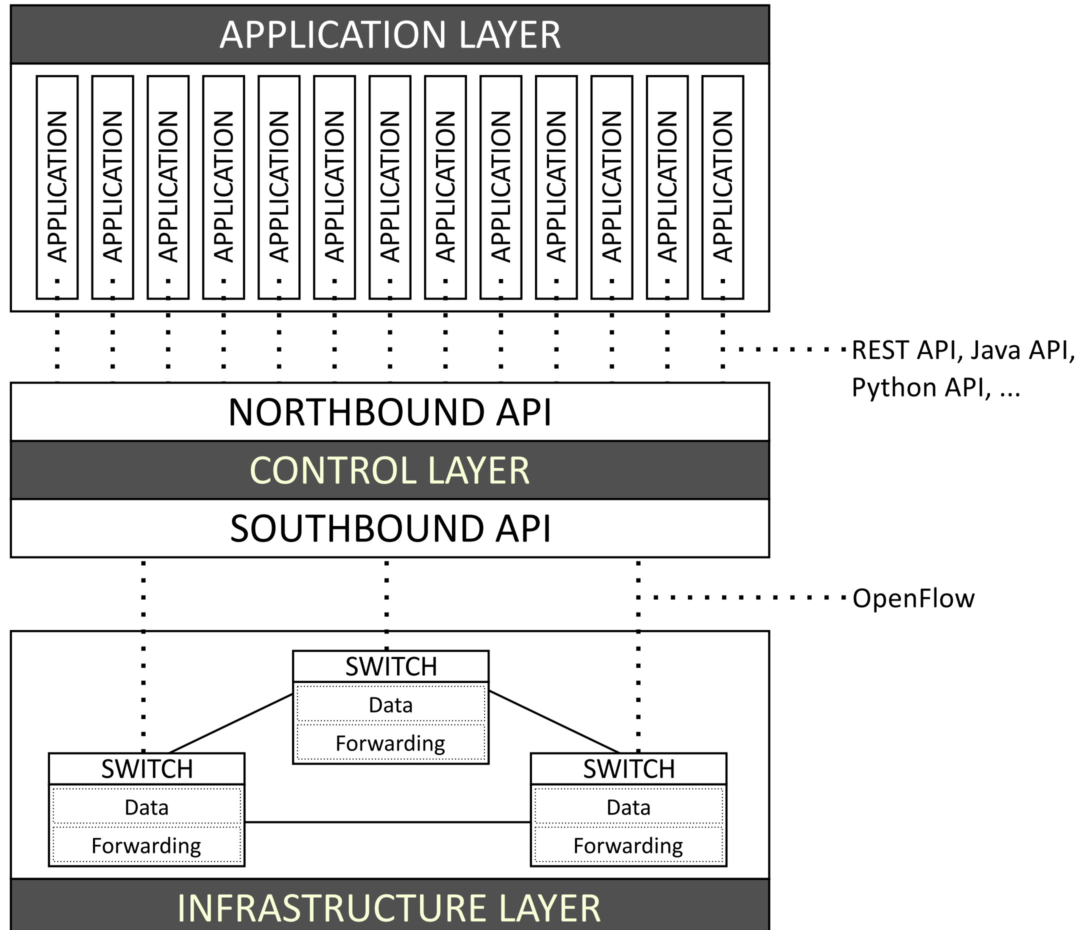


Figure 1: Open SDN architecture. Inspired by Figure 1 from *Software-Defined Networking: The New Norm for Networks* (Open Networking Foundation, 2012a, p. 7)

ing device, the OpenFlow channel used for communication between a switch and a controller, and the OpenFlow protocol that defines this communication.

Several versions of the OpenFlow specification have been released. Version 1.0.0 was the first version meant to be implemented by vendors. It represents the elementary functionality and is production-ready. Versions preceding 1.0.0 were purely experimental and not meant for production (Open Networking Foundation, 2009, p. 1). Another milestone version was version 1.3.0. It brought important new features and was designated as another stable release to be employed by software and hardware vendors. Versions 1.1.0 and 1.2.0 have not been widely implemented and the same applies for versions succeeding 1.3.0 at the time of writing (Kerner, 2012).

Due to the research nature of OpenFlow, many new features were added in the released versions, but there was little concern over backward compatibility in the beginning (Göransson and Black, 2014, p. 86). It should not be so for new versions of the specification as commercial deployments are not a rarity anymore, but current

SDN products can be found marketed as supporting OpenFlow v. 1.0 and v. 1.3, not v. 1.3 and earlier. Therefore, careful consideration should be given to the choice of the appropriate version for an application.

In the following section, I will introduce the essential functionality of OpenFlow v. 1.0.0. A summary of major innovations that have been added up to OpenFlow v. 1.3.0 can be found in Appendix B.

3.2.4 OpenFlow Specification v. 1.0.0

OpenFlow Switch Specification defines a term “OpenFlow Switch”, which represents functionality required of or proposed for an OpenFlow-enabled forwarding device. The OpenFlow Switch has two main components – a traffic processing logic and a secure channel to communicate with a controller (Open Networking Foundation, 2009, p. 2).

An Openflow switch may be implemented in either hardware, or software (Göransson and Black, 2014, p. 64–68). It must implement features stated by the specification as required, but does not have to implement optional features. A switch may be OpenFlow-only or OpenFlow-enabled (also called OpenFlow-hybrid) in case it accommodates traditional packet-processing functions as well (Open Networking Foundation, 2009, p. 3, 6; Göransson and Black, 2014, p. 84).

Every switch is uniquely identified by a Datapath ID (DPID). DPID is a 64-bit identifier, where the lower 48 bits are the MAC address of the switch, and the top 16 bits are left for the implementer. For example, the top bits could serve to identify multiple virtual switches residing on a single physical switch (Open Networking Foundation, 2009, p. 25).

Traffic processing

The rules for forwarding decisions are stored in a *flow table* in the form of *flow entries*. A flow entry consists of several fields. The main fields, ensuring the forwarding functionality, are *header fields* (also called *match fields*) and *actions*.

Match fields define a pattern against which an incoming packet is compared. Either all of them are set to a specific value or some of them may be set to ANY, which behaves like a wildcard and matches any value. Following list encompasses all match fields available in this version of OpenFlow (Open Networking Foundation, 2009, p. 2–4; Göransson and Black, 2014, p. 89):

- Input port
- Ethernet source addresses
- Ethernet destination address
- Ethernet frame type
- VLAN ID

- VLAN priority (VLAN PCP field)
- IPv4 source addresses (possibly subnet masked)
- IPv4 destination address (possibly subnet masked)
- IP protocol
- IP Type of Service (ToS) bits
- TCP/UDP source port or ICMP type
- TCP/UDP destination port or ICMP code

Every flow entry is associated with zero or more actions that form an action list. An action list designates what should happen with a matching packet. Some actions are required to be implemented in the switch, while other actions are optional. Possible actions are (Open Networking Foundation, 2009, p. 3, 6–7; Göransson and Black, 2014, p. 90–92):

- **Forward** – forward the packet to a physical port or to a virtual port. Required virtual ports are:
  - ALL – send the packet out all interfaces except the incoming one.
  - CONTROLLER – encapsulate the packet and send it to the controller for further processing by a proper application.
  - IN_PORT – send the packet out the incoming interface. This action is useful for example with wireless ports, where data is received from one user and then sent to another user through the same port. On the other hand, it must be used carefully as it may create unintended loopbacks.
  - LOCAL – send the packet to the local OpenFlow control software. This action is used when OpenFlow messages from the controller are received on a regular port that receives packets from the network as well. The local OpenFlow control software serves to process messages from the controller and react on them accordingly.
  - TABLE – process the packet using the normal OpenFlow packet-processing pipeline. This action is only for PACKET-OUT messages (described later) sent from the controller and indicates that the packet should be matched against the flow table and then treated according to the matching flow entry.

  Optional virtual ports are:
  - FLOOD – flood the packet along the minimum spanning tree, not including the incoming interface.
  - NORMAL – process the packet using traditional forwarding techniques as a usual switch. This action may be implemented in an OpenFlow-hybrid switch.

- **Drop** (required) – drop all matching packets. This action automatically applies for flow entries with no specified actions.
- **Enqueue** (optional) - send the packet to a specific queue of a particular port. Queues are associated with different priorities and serve to achieve the desired Quality of Service.
- **Modify-Field** (optional) – modify certain header fields of the packet.

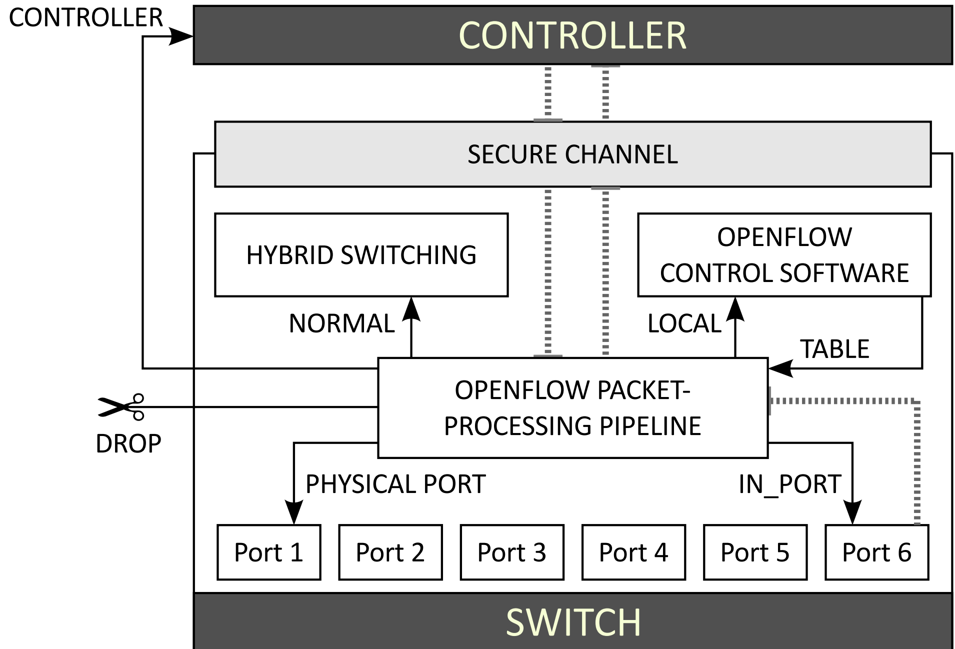


Figure 2: Elementary forwarding possibilities for a processed packet. Black solid lines represent the destinations. Gray dashed lines represent data communication – from the input port to the processing pipeline and then between the pipeline and the controller in case of an unmatched packet or a packet destined to the controller. Inspired by Figure 5.7 from *Software Defined Networks: A Comprehensive Approach* (Göransson and Black, 2014, p. 90)

From other fields of a flow entry, *priority*, *idle timeout*, *hard timeout*, and *counters* are elementary to know for basic comprehension.

Priority is a 16-bit number (0-65 535) that influences the order in which flow entries are evaluated. When matching the incoming packet, the processing logic aims to find a match with the highest priority. Exact matches, that is flow entries with all match fields set to a specific value and with no use of wildcards, are always the highest priority since they are unambiguous and the priority field is irrelevant for them. On the other hand, matches with wildcards need to have a priority defined in case more flow entries match the same packet. If multiple flow entries match a packet and have the same priority, the choice of the winning flow entry is up to the switch. An unmatched (missed) packet is forwarded to the controller (Open Networking Foundation, 2009, p. 9, 27–28).

Idle timeout defines a period of inactivity after which the flow entry expires. Inactivity represents the time when the flow entry has not matched any incoming packet. Hard timeout defines a fixed lifetime of the flow entry after which it expires regardless of its activity. Both timeouts are a 16-bit number (0-65 535) representing time in seconds (Open Networking Foundation, 2009, p. 15, 27–28).

Counters provide various statistics per table, per flow, per port, and per queue. The records may include, for example, packet lookups, packet matches, received packets, transmitted bytes, receive errors, and more (Open Networking Foundation, 2009, p. 3, 5).

Traffic processing illustration

To illustrate the matching process, I will use a simple scenario with the network topology depicted in Figure 3. There are three switches interconnected in a tree, all of them are connected to an OpenFlow controller, and each leaf switch has two end-nodes connected. The root switch A has two flow entries A1 and A2 in the flow table, which are described in Table 1 and Table 2.

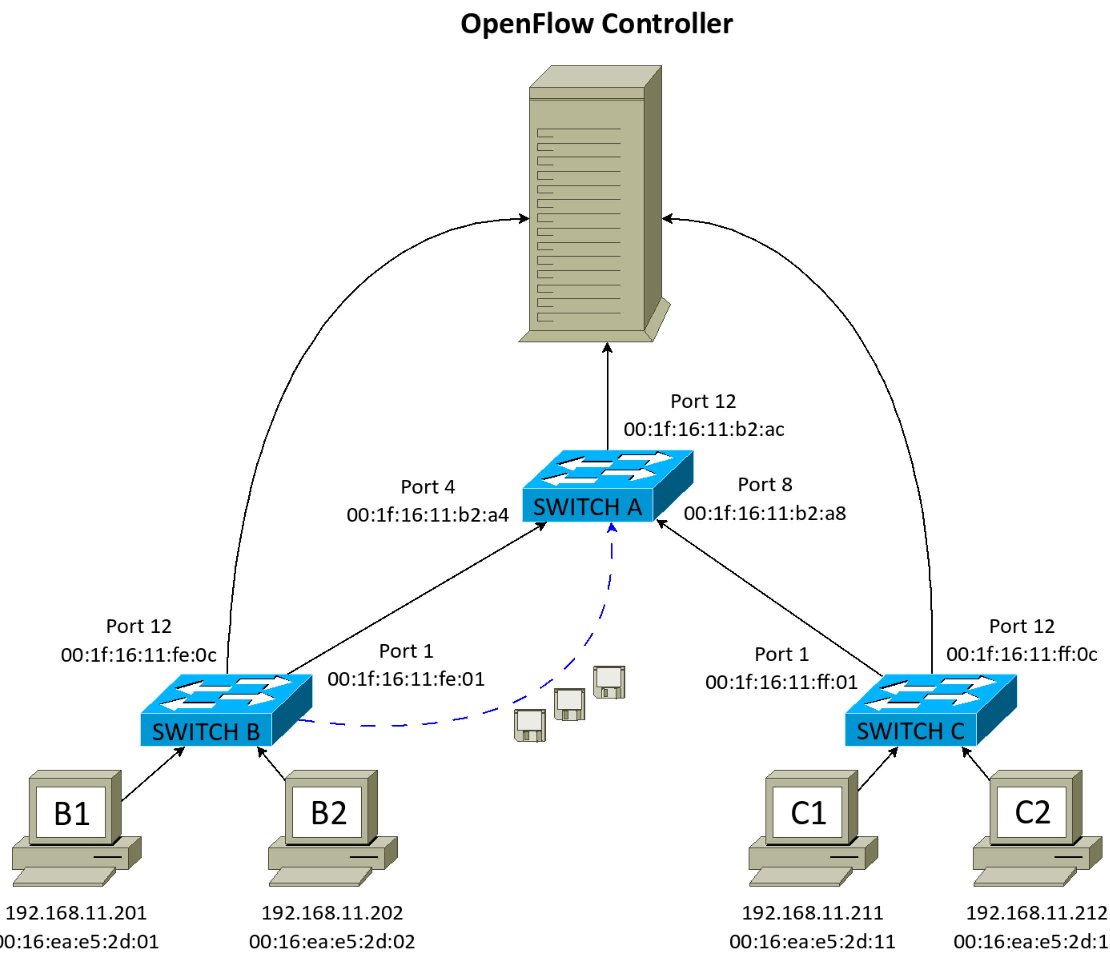


Figure 3: Example of the matching process – network topology.

Flow entry A1 matches traffic between two network interfaces – port 4 of switch A and port 1 of switch B – in case it arrives at port 4 and then sends it out through port 8. Due to the timeouts, this entry is temporary and will expire either after 60 seconds of inactivity or after 600 seconds from its inception. Flow entry A2 matches traffic between two logical nodes – B1 and C1 – and sends it to the controller for further processing. This entry never disappears at it has both timeouts set to zero. Header fields that are not mentioned in the flow entries are wildcarded and corresponding packet information is therefore irrelevant for the matching process.

Table 1: Example of the matching process – flow entry A.

Header fields	
Input port	4
Ethernet source address	00:16:ea:e5:2d:80
Ethernet destination address	00:1f:16:11:b2:6e
Actions	
Forward	Physical port 8
Priority	55 000
Idle timeout [seconds]	60
Hard timeout [seconds]	600

Table 2: Example of the matching process – flow entry B.

Header fields	
IPv4 source address	192.168.11.201
IPv4 destination address	216.58.214.227
Actions	
Forward	Virtual port CONTROLLER
Priority	60 000
Idle timeout [seconds]	0
Hard timeout [seconds]	0

Table 3 then depicts a packet coming into the switch A, where flow entries A1 and A2 are configured. Concerning the match fields defined in these flow entries, both entries would match this packet, but according to their priorities, flow entry A2 would be the first to be evaluated and therefore the one used for the processing of the packet.

Secure channel

The secure channel is the interface that connects an OpenFlow switch with an OpenFlow controller for configuration and management purposes. The communication is typically encrypted using TLS, even though unencrypted TCP connections are also

Table 3: Example of the matching process – incoming packet.

Header fields	
Ethernet source address	00:16:ea:e5:2d:80
Ethernet destination address	00:1f:16:11:b2:6e
IPv4 source address	192.168.11.201
IPv4 destination address	216.58.214.227
IP protocol	TCP
TCP source port	50213
TCP destination port	80
Input port	4

possible. The connection between a controller and a switch may be out-of-band or in-band. Out-of-band means a dedicated management port is used for the communication, which is not processed by the data plane of the switch. In-band, on the other hand, means a regular port is used for the communication and the data plane has to forward the packets to the internal OpenFlow control software (Göransson and Black, 2014, p. 86–87; Open Networking Foundation, 2009, p. 9, 12–13).

Between a controller and a switch, three types of messages are interchanged, each having several sub-types: *symmetric*, *asynchronous*, and *controller-to-switch*.

Symmetric messages are sent without solicitation in either direction, i.e. from the controller to the switch or from the switch to the controller, and have following sub-types (Open Networking Foundation, 2009, p. 11; Göransson and Black, 2014, p. 94):

- HELLO – is exchanged between the switch and the controller as the connection is being established. It also serves to determine the highest common supported version of OpenFlow.
- ECHO – may be sent from either side and must be replied to by the other side. It serves to check the liveness of the connection and to measure latency or bandwidth.
- VENDOR– is provided as a standard way for experiments or proprietary extensions.

Asynchronous messages are sent from the switch to the controller in reaction to occurred events without the controller soliciting them. There are four sub-types for asynchronous messages (Open Networking Foundation, 2009, p. 10–11; Göransson and Black, 2014, p. 94):

- PACKET_IN – is sent whenever there is a packet that does not match any flow entry or that matches a flow entry with the action set to forward the packet to the controller.

- FLOW_REMOVED – is sent when a flow entry expires due to its timeouts, either idle or hard, or when a flow entry is removed by a FLOW_MOD message. In the FLOW_MOD message, there is a flag that determines whether the switch should or should not send the FLOW_REMOVED message when a flow entry is removed.
- PORT_STATUS – is sent when a change in the port configuration state occurs. It may be a configuration change carried out by an administrator or a change in the physical medium, such as a change in the spanning tree, if spanning tree is supported by the switch.
- ERROR – is used by the switch to notify the controller of problems.

Controller-to-switch messages are sent from the controller to the switch and may or may not require a reply. Their sub-types can be divided into several categories (Open Networking Foundation, 2009, p. 10; Göransson and Black, 2014, p. 94):

- **Switch configuration messages**
  - FEATURES_REQUEST/REPLY – the request message is sent by the controller during the establishment of the connection between the controller and the switch. In the reply, the switch specifies its capabilities and supported features.
  - GET_CONFIG_REQUEST/REPLY – is sent by the controller to retrieve the actual configuration parameters from the switch.
  - SET_CONFIG – is used by the controller to set configuration parameters in the switch. The switch does not reply to this message.
- **Controller command messages**
  - PACKET_OUT – serves to send a data packet, received or created, to the switch for forwarding. The packet may be directly sent out a specific port or it may be processed in the flow table.
  - FLOW_MOD – is crucial for the data plane programming of networking devices. Using this message, the controller adds, modifies, and deletes flow entries in a switch.
  - PORT_MOD – is used to modify the state of an OpenFlow-managed port. Different behavior may be configured regarding spanning tree protocol[1] and treating packets. A port may also be taken down administratively.

[1] An OpenFlow switch may optionally support 802.1D Spanning Tree Protocol (STP). It is then expected to process STP packets locally before performing a lookup in the flow table. More details can be found in sections 4.5 (Open Networking Foundation, 2009, p. 13) and 5.2.1 (Open Networking Foundation, 2009, p. 17) of the OpenFlow specification.

- **Statistic messages**
  - STATS_REQUEST/REPLY – is sent by the controller to retrieve the statistics gathered by counters in the switch.
- **Barrier messages**
  - BARRIER_REQUEST/REPLY – is sent by the controller to ensure that particular commands have been executed. If a switch receives a BARRIER_REQUEST message, it must execute all commands received prior to the message before it can continue executing any commands received afterwards. Upon completing the execution, the switch sends a BARRIER_REPLY message to the controller.
- **Queue configuration messages**
  - QUEUE_GET_CONFIG_REQUEST/REPLY – serves to query the switch for queues configured on a port and their parameters. The queue configuration itself is not a part of the OpenFlow protocol and is provided by an unspecified external instrument.

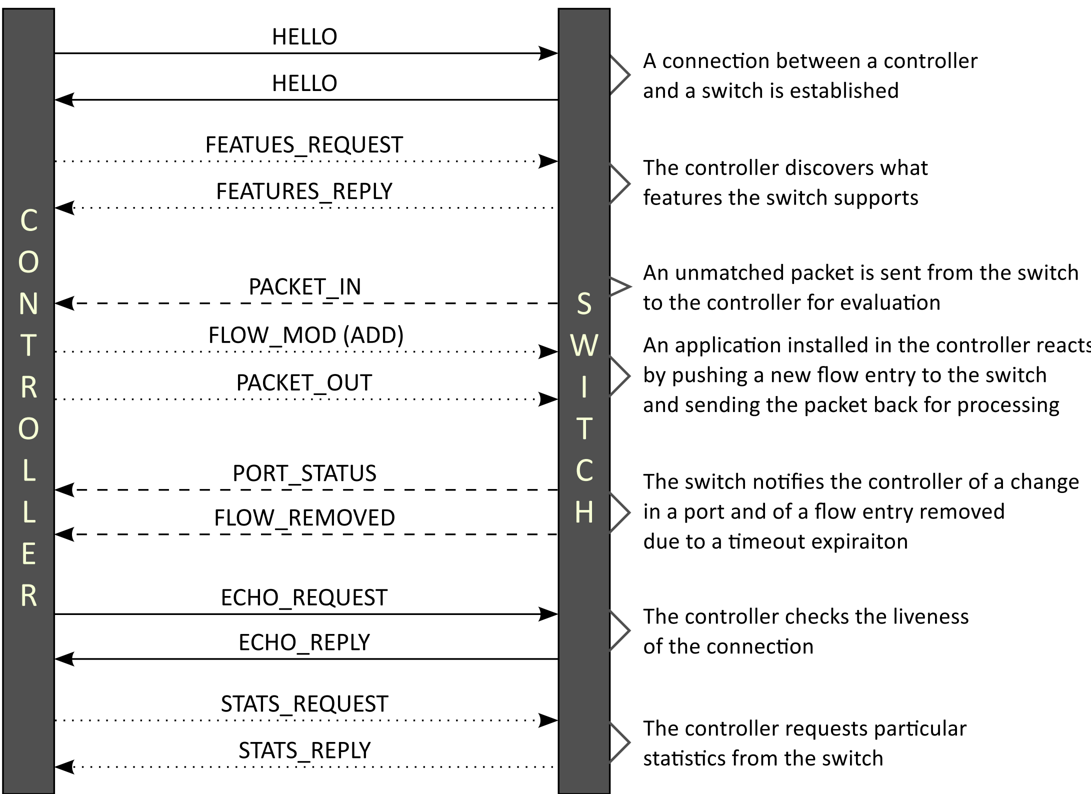


Figure 4: Example of the secure channel communication that takes place after establishing the TCP connection between the switch and the controller. Solid lines represent symmetric messages, dotted lines represent controller-to-switch messages, and dashed lines represent asynchronous messages. Inspired by Figure 5.8 from *Software Defined Netowrks: A Comprehensive Approach* (Göransson and Black, 2014, p. 93).

More information and details regarding OpenFlow v. 1.0.0 can be found in its complete specification (Open Networking Foundation, 2009).

3.2.5 Controller

In the middle layer of the architecture is a controller, whose structure is depicted in Figure 5. The controller is a software that serves as a central control point that overlooks the network and through which applications can access and manage the network.

When the controller is said to be a central point of the network, it is only meant to be logically centralized. The controller software is typically deployed on a high-performance server machine, but to distribute the load or to ensure high availability and resilience, more servers may be involved and connected in various topologies (Göransson and Black, 2014, p. 121–126).

The controller is responsible for following tasks (Göransson and Black, 2014, p. 70):

- **Device discovery** – the controller takes care of the discovery of switches and end-user devices, and their management.
- **Network topology tracking** – the controller investigates the links interconnecting devices in the network and keeps a view of the underlying resources.
- **Flow management** – the controller maintains a database mirroring the flow entries configured in the switches it manages.
- **Statistics tracking** – the controller gathers and keeps per-flow statistics from the switches.

It is important to emphasize that the controller does neither control the network in any way nor does it replace any networking devices. Even the basic switching or routing functionality has to be provided by specific applications that approach the network through the controller (Göransson and Black, 2014, p. 72).

Communication with networking devices is realized through a southbound interface, for which Open SDN promotes the OpenFlow protocol. This interfaces is used to configure and manage the switches and to receive messages from them. The connection is realized via a secure channel and depending on the setting is either encrypted or unencrypted. Details on the secure channel were explained in previous sections (Göransson and Black, 2014, p. 69).

Applications communicate with the controller using a northbound interface. Through this interface, they retrieve information about the network and send their requests, while the controller uses it to share information about occurring events. Depending on the implementation, the interface may be low-level, providing a unified access to individual devices, or high-level, abstracting much of the underlying layer and rather presenting the network as whole. There is no standard for the

northbound interface and every controller implements its own APIs – be it Java API, Python API, REST API or else. This current lack of a standard northbound interface makes it difficult to create controller independent applications (Göransson and Black, 2014, p. 69–70).

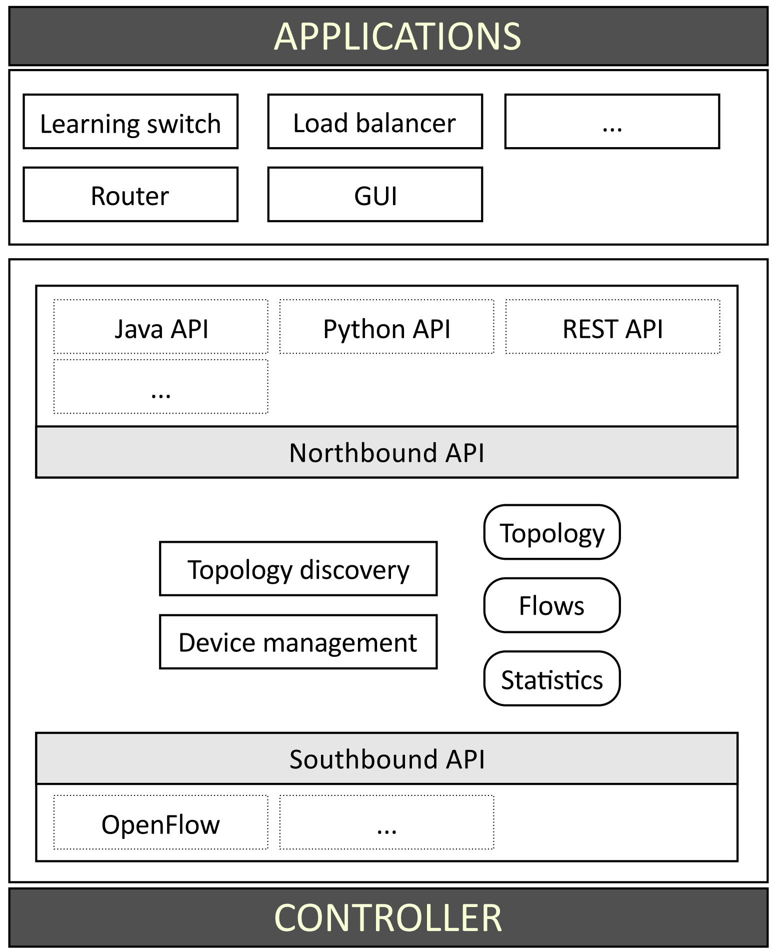


Figure 5: Elements of the SDN controller. Inspired by Figure 4.5 from *Software Defined Networks: A Comprehensive Approach* (Göransson and Black, 2014, p. 69).

More on the concepts regarding SDN controller can be found in the SDN Architecture specification (Open Networking Foundation, 2016d, p. 22–36).

3.2.6 Applications

The application layer is at the top of the architecture. As was mentioned before, the controller itself does not control the network in any way – it only provides an abstracted approach, applications are therefore crucial to the operation of the whole network. Applications communicate with the controller through a northbound interface and are responsible for carrying out the functionality present in traditional

networking devices, such as Layer 2 switching, Layer 3 routing, and other. Applications are also used for management purposes, for example, a graphical user interface for the controller would be implemented as an SDN application (Göransson and Black, 2014, p. 72–73).

There are two kinds of applications – reactive and proactive. Reactive applications, simply said, react to events occurring in the network. For that reason, they need to be notified of those events by the controller. These applications are typically written in the native language of the controller and register listeners in the controller to be asynchronously, i.e. without requests, informed of all events. There are three basic kinds of listeners (Göransson and Black, 2014, p. 213):

- **Switch Listener** – receives notifications about added or removed switches, or about a change in the port status of a switch.
- **Device Listener** – receives a notification whenever an end-user device is added, removed, moved to another switch, or when its IP address or VLAN membership changes.
- **Message Listener** – is informed of all packets the controller receives.

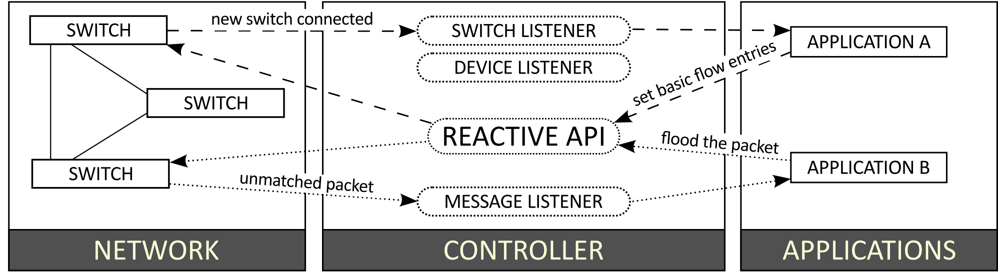


Figure 6: An illustration of a reactive application. Application A reacts to a newly connected switch by configuring it with a basic set of flow entires. Application B processes an unmatched packet sent to the controller and decides to flood it out all interfaces.

When the application receives a message about an event, it may take a proper action. A typical reaction might be one of the following (Göransson and Black, 2014, p. 213–214):

- Packet-specific action – the switch is told what to do with the particular package, whether to drop it or to forward it to a specific port.
- Flow-specific action – in this case, the application programs a new flow entry into the switch. The next time the switch encounters a packet from the same flow, it will process it on its own using the new flow entry.

Proactive applications, as opposed to reactive applications, have no listening functionality. Their methods may be evoked by external events or they may make periodical calls to the northbound interface. Requests sent to the controller mainly serve to either receive data or program switches. The controller itself has no means for contacting a proactive application (Göransson and Black, 2014, p. 215).

While reactive APIs are typically low-level and close to the OpenFlow protocol, proactive APIs may vary in the degree of abstraction they provide. The API may be as well quite low-level, for example allowing flow entry configuration on individual switches, or it may be high-level, shielding the implementation details and providing a view in the form of a virtual network (Göransson and Black, 2014, p. 215).

A proactive application may, for example, be used to push general flow entries to switches when a virtualization service informs about a migration of a virtual machine (Göransson and Black, 2014, p. 215). Another use case may be a monitoring tool that periodically retrieves information about the network and processes it.

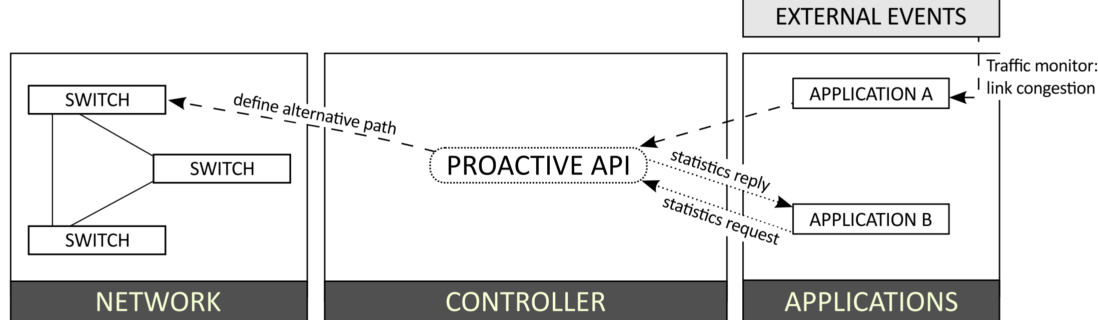


Figure 7: An illustration of a proactive application. Application A is evoked by an external event – a traffic monitor informs it about a congested link. Based on the received information, application A defines a temporary alternative path for the affected traffic. Application B sends a request for statistics and then receives a reply, which it can further process.

3.2.7 SDN via APIs

This approach, sometimes also called evolutionary, strives to enhance software control and programmability of networks by exploiting the existing network infrastructure devices. The same traditional switches are employed, the control plane remains distributed over individual devices, but their software is improved to support more powerful APIs. The switches are then controlled either on a per-device basis or via a controller (Göransson and Black, 2014, p. 74–75).

While conventional means like CLI or SNMP may be engaged, they are best suited for rather static configurations. Modern APIs such as RESTful API, appropriate for dynamic and automated control, therefore come into play. Applications running above the controller may then make use of the extended APIs and provide a certain degree of the desired programmability and automation. The set of features provided by the APIs is dependent upon the particular implementation of each vendor since these solutions are proprietary (Göransson and Black, 2014, p. 74–76).

More details on this approach can be found in the book of Göransson and Black (2014, p. 74–76, 130–135).

3.2.8 SDN via hypervisor-based overlay networks

In this approach, the existing physical infrastructure is left completely untouched and instead a virtual overlay network is established above it. Suited for virtualized environments, this overlay network is comprised of virtual machines (VMs), virtual switches running on hypervisors, and a controller (Göransson and Black, 2014, p. 76–77).

The communication between VMs is realized through MAC-in-IP tunneling. The packet a VM wants to send to another VM is taken from Ethernet header inward and at a virtual switch encapsulated within another IP datagram, as depicted in Table 4. The switch, also called a virtual tunnel endpoint, than sends the encapsulated packet over the underlying physical network to the virtual switch that is connected to the destination VM. At this point, that is at the other end of the virtual tunnel, the packet is decapsulated and the inner datagram is delivered to the corresponding VM according to its headers. This way, the virtual machines work in an isolated reality behaving as if they were connected by a normal network and do not know anything about what is happening on the outside (Göransson and Black, 2014, p. 77).

Table 4: MAC-in-IP packet encapsulation.

Outer Ethernet header	Outer IP header	Tunneling protocol header	Inner Ethernet header	Inner IP header	Inner IP payload

The controller is responsible for keeping necessary networking information about all virtual switches and the end-nodes connected to them in order to ensure connectivity between all virtual machines in a particular virtual network (Göransson and Black, 2014, p. 136). Numerous virtual networks can separately operate on the same physical network (Göransson and Black, 2014, p. 78).

As for the MAC-in-IP tunneling mechanism, there are several technologies in use and each vendor supports one or more of them in their solutions based on this approach. For the main three technologies – VXLAN, NVGRE, and STT – they all work on the same principal and differ only in the implementation of this method (Göransson and Black, 2014, p. 77. 152–155).

More details on SDN via hypervisor-based overlay networks can as well be found in the book of Göransson and Black (2014, p. 76–78, 135–139).

3.3 Tools

3.3.1 HPE VAN SDN Controller

Virtual Application Networks (VAN) SDN Controller, a product of Hewlett Packard Enterprise (HPE), is a commercial SDN controller well-designed to serve in demanding environments like campuses or data centers (Hewlett Packard Enterprise, 2016a).

HPE VAN SDN Controller, latest version 2.7.10 at the time of writing, supports OpenFlow v. 1.0 and v. 1.3, Netconf, and SNMP protocols for southbound communication, which is used to control connected networking devices. As for northbound interfaces, there is a RESTful HTTPS interface brining an abstract representation of the network to external (proactive) applications, and for reactive applications there is a native Java API allowing the applications to run within the controller. The controller provides a graphical user interface for the controller administration (Hewlett Packard Enterprise, 2016a).

Data Path ID	Address	Negotiat…	Manufacturer	H/W Version	S/W Version	Serial #
00:00:00:00:00:00:00:01	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:02	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:03	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:04	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:05	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:06	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:07	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:08	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:09	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:0a	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:0b	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:0c	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:0d	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:0e	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:0f	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:10	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:11	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:12	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:13	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:14	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:15	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:16	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:17	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:18	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:19	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None
00:00:00:00:00:00:00:1a	192.168.1.89	1.3.0	Nicira, Inc.	Open vSwitch	2.0.2	None

Total Rows: 31

Figure 8: HPE SDN VAN Controller GUI – OpenFlow Monitor.

This controller was used in previous experiments of the networking work group (Pokorný and Zach, 2015) and is employed in this work to continue in the preceding efforts because it also conforms to potential future needs. In case a deployment of SDN is considered in the university network, this controller solution is suited for such environments and its producer can provide adequate support.

According to the support matrix (Hewlett Packard Enterprise, 2016b, p. 5), the controller requires a well-equipped machine (2.2 GHz x86-64 4-core processor, 8 GB RAM available to the controller, 75 GB of disk space) even for development and test deployments, but it can function under worse conditions for less demanding scenarios. With growing demands for controller operation, the demands for server performance grow as well.

The controller software is provided in two forms (Hewlett Packard Enterprise, 2016c, p. 5):

- as a virtual appliance, which is prepared for immediate usage with everything preinstalled, from HPE Linux operating system to concrete packages.
- as a Debian package for installation on Ubuntu Linux.

The installation process is well documented in the Installation Guide (Hewlett Packard Enterprise, 2016c).

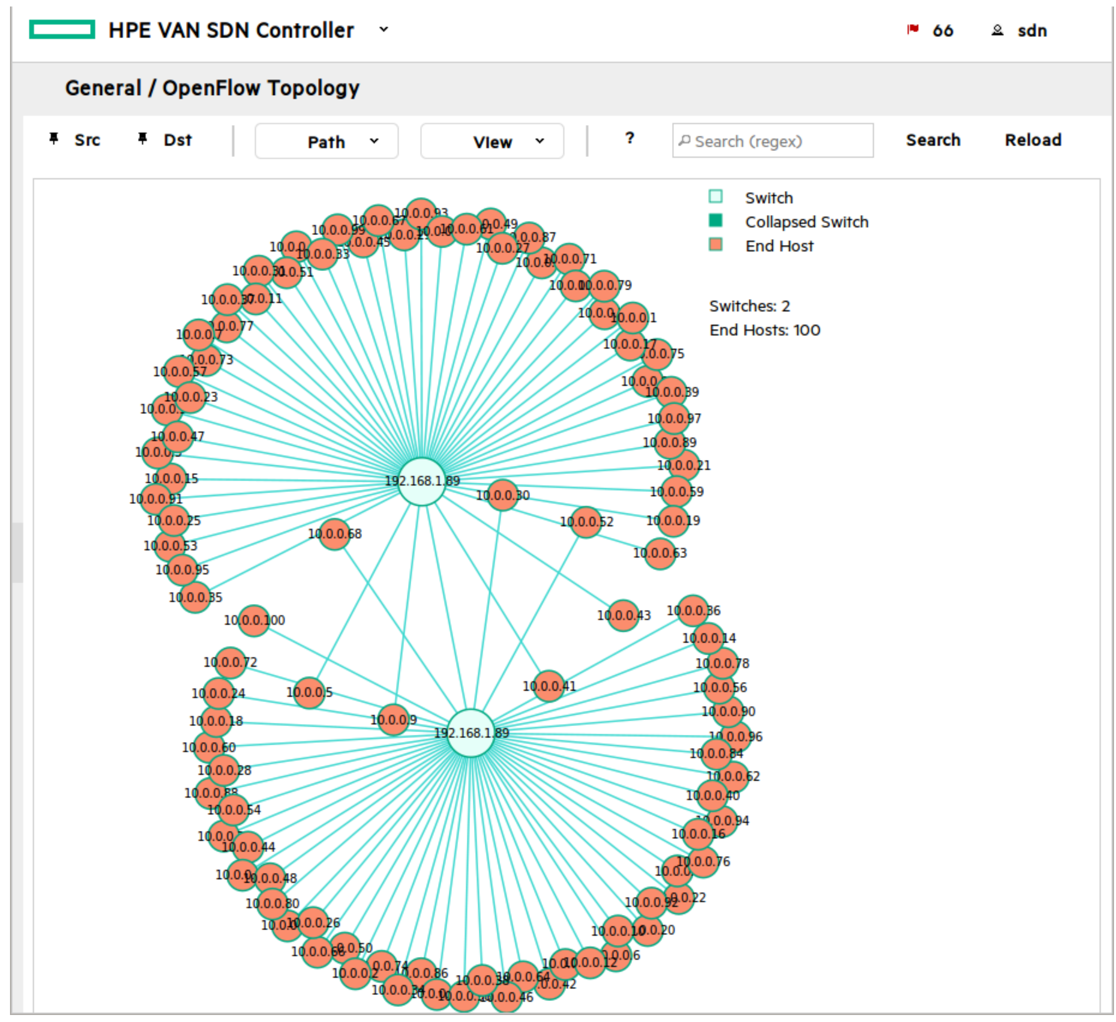


Figure 9: HPE SDN VAN Controller GUI – OpenFlow Topology.

When the controller is installed and running, there are a few useful places for management and experimentation (Hewlett Packard Enterprise, 2016d, p. 21; Hewlett Packard Enterprise, 2016e, p. 10):

- `https://controller-IP-address:8443` – graphical administration of the controller.
  - OpenFlow Monitor – an overview of all switches connected to the controller provides detailed information about the switch, its ports and defined flow entries and groups. This tool is captured in Figure 8.
  - OpenFlow Topology – a simple visualization of the connected network that displays not only the network topology, but may also provide elementary information about end-nodes and switches such as port identifiers, MAC addresses, IP addresses and other. It can also show the shortest path between two selected end-nodes computed using the Dijkstra algorithm. This tool is captured in Figure 9.
- `https://controller-IP-address:8443/api` – RSdoc, an interactive online documentation of the REST API. If authenticated, the user can access and manipulate real controller data, therefore it should be used carefully in order to not cause undesired changes. The REST API Reference (Hewlett Packard Enterprise, 2016e, p. 10–11) provides instructions on how to authenticate through RSdoc.
- `https://controller-IP-address:8443/sdn/v2.0/models` – REST API JSON data model, which presents all possible JSON objects and their attributes that can be used in the communication via the REST API.

3.3.2 REST and RESTful API

Representational State Transfer (REST) is an architectural style for application interfaces. It defines a set of constraints an interface must comply with to be called RESTful. RESTful APIs are to provide simple and clean access to resources and their manipulation (Fredrich, 2013, p. 6).

Every resource is accessible through its identifier – URI – and a client may read or manipulate the resource using messages. It is important to note that only a representation of the resource is given to the client, for example, a server does not send the whole database, but only a, for example, XML representation of particular records. When a client holds the representation of a resource, it contains enough information for manipulation of the resource on the server if the client has adequate permissions. Similarly, each message includes information describing how to process it, for example by specifying a media type (Fredrich, 2013, p. 7).

While the architecture demands no particular transfer protocol to be used, HTTP is advantageously used in most cases. Its request methods GET, POST, PUT,

and DELETE provide all the elementary operations for reading or manipulating resources (Fredrich, 2013, p. 11–14):

- GET - is used for retrieving information and its request body is empty.
- POST - is used for creating a new record, for which the server decides the new URI. The request body contains the data of the new record.
- PUT - is used for updating an existing record or creating a new record, for which the user specifies the URI. The request body contains the data of the modified or new record.
- DELETE - is used for removing an existing record based on the provided URI and its request body may therefore stay empty.

It is worth noting that different APIs may implement these HTTP methods in a slightly different manner, the corresponding documentation should therefore always be investigated.

For the representation of resources, there are as well no restrictions on the chosen format. Typically a general-purpose standard format is used such as HTML, XML, or JSON for simple processing, but any other format may be used that best suites the particular data (Fredrich, 2013, p. 7).

Simple communication over a RESTful API is captured in Figure 10 and Figure 11 in the form of extracts from Wireshark, a network protocol analyzer. In Figure 10, an HTTP request is made. It uses the GET method for reading, and approaches a resource available under URI `/get` of server `httpbin.org`. The request message has no body. In Figure 11 a response is received. Content type `application/json` declares that the resource is represented as a JSON object, and the resource representation itself can be seen in the body of the response message.

```
▾ Hypertext Transfer Protocol
  ▸ GET /get HTTP/1.1\r\n
    Host: httpbin.org\r\n
    Connection: keep-alive\r\n
    Cache-Control: max-age=0\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.71 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
    Accept-Encoding: gzip, deflate, sdch\r\n
    Accept-Language: en,cs;q=0.8\r\n
    Cookie: _ga=GA1.2.446253191.1477571124; _gat=1\r\n
    \r\n
    [Full request URI: http://httpbin.org/get]
    [HTTP request 1/1]
    [Response in frame: 149]
```

Figure 10: RESTful API – HTTP GET request.

3.3.3 Python

Python is an interpreted high-level programming language that is built around objects and rich data structures with dynamic typing and binding. Python programs

```
▾ Hypertext Transfer Protocol
  ▸ HTTP/1.1 200 OK\r\n
    Server: nginx\r\n
    Date: Thu, 27 Oct 2016 12:32:54 GMT\r\n
    Content-Type: application/json\r\n
  ▸ Content-Length: 567\r\n
    Connection: keep-alive\r\n
    Access-Control-Allow-Origin: *\r\n
    Access-Control-Allow-Credentials: true\r\n
    \r\n
    [HTTP response 1/1]
    [Time since request: 0.123735000 seconds]
    [Request in frame: 134]
▾ JavaScript Object Notation: application/json
  ▾ Object
    ▸ Member Key: "args"
    ▸ Member Key: "headers"
    ▸ Member Key: "origin"
    ▸ Member Key: "url"
```

Figure 11: RESTful API – HTTP response with a JSON object.

are slower than those written in languages like Java or C++, but the code of equivalent programs is significantly shorter. As such, it is well suited for example for rapid application development. Nonetheless, it is a full-fledged programming language with a wide palette of standard as well as third-party libraries and it can be used for a broad range of applications (Python Software Foundation, 2016b; Van Rossum, 1997).

Python was chosen as the base programming language for this thesis not only for its simplicity and high-level programming approach, but also because of the existence of HP SDN Client – a library that provides bindings for the REST API of the HPE controller. It makes the interaction with the controller simple by covering the HTTP requests and the data in readable and easy-to-use methods and data structures (Tucker, 2014). Version 3 of Python was chosen in order to build the project on the newest version, even though it is still making its way to replace the widely-spread version 2 and compatibility issues may appear.

3.3.4 Graphical user interface (GUI) framework

When choosing the GUI framework, I had only little experience with creating graphical interfaces and did not know in advance all elements that might be needed in the project. Also, I could not have guessed whether it would be feasible to implement all features in a particular framework without having sufficient prior experience with it.

Therefore, I looked for a framework that would be under active-development, multi-platform, well-established, and with sufficient documentation and learning resources. Preferably, I also wanted to gain knowledge of a framework that I could use in future projects, possibly with other programming languages. To create the first list of possible choices, I used an overview of GUI frameworks for Python (GUI Programming in Python, 2016).

From the beginning, I filtered out frameworks that are obsolete or intended for a very specific and narrow usage, and then ended up with these remaining frameworks:

- **Kivy** – is an open-source Python library that allows rapid development of applications with innovative user interfaces. The framework is mostly concerned with the graphical interface part of applications and as such provides native support of many inputs and devices, allowing creation of multi-touch applications. It can run the same program on Windows, Linux, macOS, Android, and iOS without any code modifications (Kivy, 2016a). Kivy supports Python 3 except for iOS build tools that require Python 2.7. (Kivy, 2016b).
- **PyGObject** – is a set of Python bindings for the GNOME software platform (Projects/PyGObject, 2016). The platform encompasses the GTK+ framework for user interfaces as well as components, for example, for networking, data access, multimedia, communication, and other (McCance et al., 2016). While GNOME as a whole is targeted at Linux distributions (GNOME, 2016), GTK+ is multi-platform and may be used on Linux as well as Windows and macOS (The GTK+ Team, 2016). PyGObject supports Python 3 (1. Installation, 2016).
- **PyQt** – is a library of Python bindings for the Qt application framework. Qt is mostly known as a graphical user interface (GUI) framework, but it also includes modules, for example, for networking, databases, threads, multimedia, web browsing, and more. PyQt runs on the same platforms as Qt - mainly Windows, Linux, macOS, iOS, and Android. PyQt supports Python 3 and the latest version of Qt – 5.7 (Riverbank, 2016).
- **PySide** – is similar to PyQt, but it is an officially supported Python binding for Qt. It works on Linux, Windows, macOS and Maemo. PySide supports Python 3, but it is currently available only for Qt version 4.8, which is no longer supported (Shaw, 2014), and bindings for version 5 are under development (PySide, 2016; PySide FAQ, 2016).
- **Tkinter** – is a Python interface for the Tk GUI toolkit, which is integrated with the Tcl programming language. Tkinter is considered to be the standard Python GUI package (Python Software Foundation, 2016a; Tkinter, 2014). It offers a range of commonly used elements, which can be further extended by various extensions (Roseman, 2015). As for supported platforms, Tkinter is available for most Unix platforms (including Linux and macOS) and Windows systems (Python Software Foundation, 2016a). As a standard package, Tkinter is available in Python 3.
- **wxPython** – is a Python binding for the wxWidgets library (wxPython, 2016). wxWidgets covers not only the area of graphical interfaces, but also many other parts of application development, such as file manipulation, audio and video playback, multithreading, HTML rendering, and more (wxWidgets, 2016). wxPython can be used on Window as well as Unix-like systems, be it Linux, macOS,

or any other (wxPython, 2016). wxPython does not currently support Python 3, even though a new generation of the project is under development (How to install wxPython; ProjectPhoenix, 2015).

Of this list, I removed PySide and wxPython right away – PySide for supporting only an outdated version of Qt and wxPython for not supporting Python 3, on which I wanted to build my project. Then I excluded Kivy and Tkinter because they both deal only with graphics. Also, Kivy can be used only with Python and Tkinter does not seem to offer advanced GUI elements. In the end, I chose PyQt over PyGObject because of the bias the GNOME platform has for Linux systems.

Qt, ported to Python through PyQt, is a mature and advanced application framework suitable for a wide range of applications, including very demanding ones, that can be run on all major platforms of today. Qt offers an extensive documentation and once learned through PyQt, the gained experience can then be used with the authentic Qt, which is based on C++ (The Qt Company, 2016).

3.3.5 VirtualBox

VirtualBox is an open-source virtualization platform that enables a computer to run multiple operating systems at once. Such software is called a hypervisor. VirtualBox is a hosted hypervisor, which means that it needs an operating system to run on compared to bare-metal hypervisors running directly on the server. It can run on any standard operating system – Windows, Linux, Mac, or Solaris (Oracle Corporation, 2016).

VirtualBox is a useful tool in the development process as it allows creation of separate workspaces, which run only the required programs and do not in any way collide with the hosting operating system. It also comes in handy when an environment with multiple computers is desired, be it a server and a client, or a cluster of equal machines.

Management of virtual machines is simple using the provided graphical user interface. To create a new virtual machine, the user only has to allocate the required space for a virtual disk image, configure parameters such as number of CPU cores or amount of RAM, then mount the installation media, install the operating system and use it just as a regular system would be used.

3.3.6 Mininet

Mininet is an open-source network emulator that creates a virtual network with hosts, switches, controllers, and links. It runs on Linux as it utilizes a standard Linux feature of network namespaces that allows individual processes to have separate network interfaces, routing tables, and ARP tables. The hosts can also run standard Linux network applications (Mininet Team, 2016a).

Mininet supports SDN and uses OpenFlow-enabled virtual switches. It can be conveniently used to run complex topologies with applications using real code, which can be later easily transferred to hardware with little or no modifications. It is controlled through a CLI and also offers a Python API for creating networks and experimenting. With its capability to emulate various kinds of networks, Mininet is a real-world-like test environment for networking (Mininet Team, 2016a).

Mininet can be either installed on an existing Linux operating system or an official virtual machine with the necessary packages preinstalled may be used (Mininet Team, 2016b).

Elementary Mininet commands and their usage are described in Appendix C.

4 Application design

In this chapter I conceptually describe how I designed the application, for which I will from now on use a code name *Visdan* for the purpose of clarity. Different models of interaction between Visdan and a network are introduced, followed by the application architecture, whose parts are then examined in detail.

4.1 Models of communication with the network

Visdan approaches the network through a controller (HPE VAN SDN Controller, as mentioned before), but there are several different ways how the cooperation with the controller may be accomplished. In the following sections I introduce three application models I was considering for Visdan.

4.1.1 Application residing in the controller

In this case, the application is based on the native Java API and implemented to operate directly in the controller. It registers listeners for events regarding changes in the topology and immediately reflects them in the visualization. Residing right in the center of activity, the application does not generate any traffic when querying the controller for network topology or statistics. Also, its graphical user interface is integrated straight into the administration interface of the controller. When used simultaneously by more users, their created data is stored in the same place and therefore synchronized among all of them.

On the other hand, the application is tailored for a particular controller and its native API, which makes it difficult if not even impossible to adjust it for another controller.

4.1.2 Application with an intermediate server

This approach adds an extra server in between the application and the controller. The role of the server is to periodically collect data from the controller about network topology and statistics, and to process application requests and hand them over to the controller. The server uses the REST API available for external applications, which makes it possible to add communication modules for other controllers that also provide the REST API.

The application itself then connects to the server using an appropriate communication interface and uses it to retrieve the necessary data or to push requests for changes in the network. With all information stored on the server side, multiple instances of the application can run at the same time and access the same data, including the data created by other users.

As for the disadvantages, this setting creates a significant amount of network traffic since the server needs to download all the data from the controller and then every instance of the application downloads it again from the server. At least for the server, it is necessary to always download the whole network database and then investigate it for possible changes. The server could then accommodate a service that would provide the applications only with updates. The same functionality could be possibly deployed also on the controller as a native application with an appropriate API for access from the outside.

4.1.3 Standalone application

In this scenario, the application exists independently of the controller or any other component. It exploits the REST API to retrieve the required data from the controller or to push it back, which allows it to be extended to support other controllers with the REST API. Requiring no prior preparation, the application can be instantly used in any environment where necessary.

However, this isolation leads to several shortcomings. Since every instance of the application runs completely separately, each of them has to periodically request data from the controller concerning network topology and statistics. This burdens not only the network, as was the case in the second model, but also the controller itself as all requests are directed to it. But as was mentioned previously, an application could be implemented to run in the controller and through a special interface provide information on network updates, which could decrease the amount of produced traffic. Also, any information created by users in their application remains in that particular instance and can by no means be synchronized with other individual instances.

4.1.4 The chosen model

The first model provides the best performance, but is limited for use with one specific controller. The second model can be potentially used with various controllers, but adds an extra component, an intermediate server. The third model is flexible concerning deployment, but burdens the network substantially.

I have in the end decided to implement Visdan using the third model despite all its flaws. Key arguments for the decision were openness for other controllers, independence of any additional elements, and more straightforward development. Created in such manner, Visdan should be well suited for an experimental environment, although it would not fit into a production setting because of the performance issues.

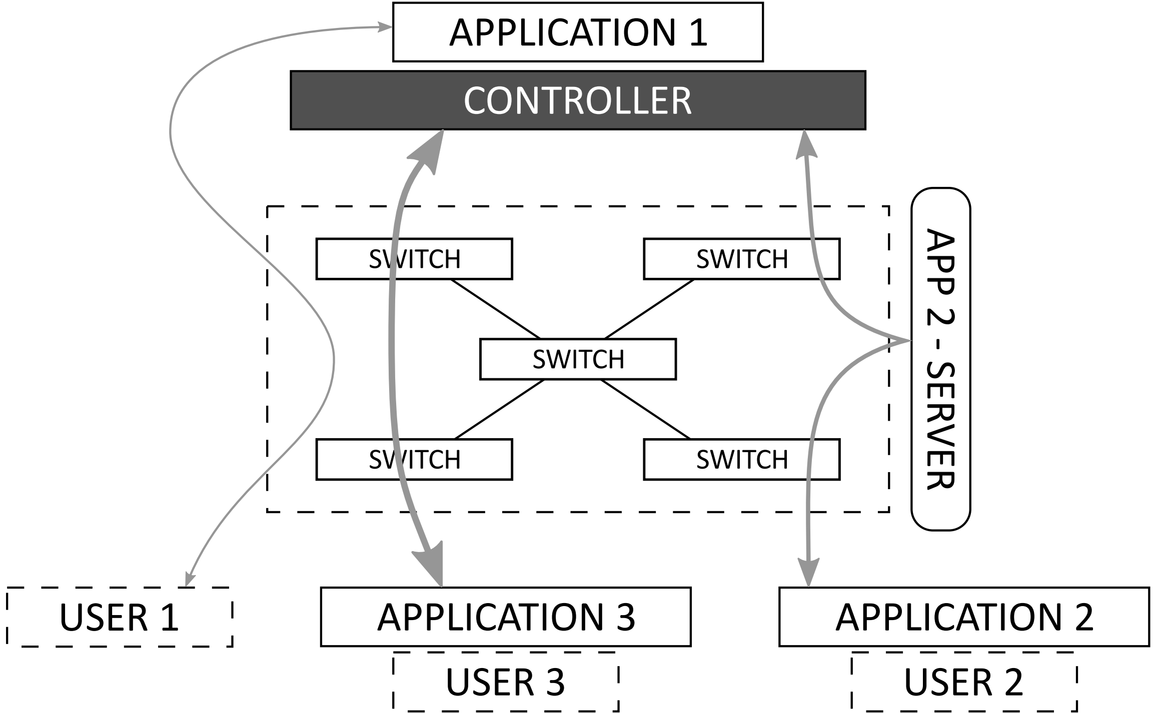


Figure 12: A scheme of the application models and how they operate relative to the controller. Application 1 stands for the application residing in the controller, Application 2 stands for the application with an intermediate server, and Application 3 stands for the standalone application. The thickness of the connections represents the traffic load each application model creates.

4.2 Application architecture

I have chosen the MVC (Model-View-Controller) architecture as a foundation for the design of Visdan. MVC separates an application into three distinct sections (or layers), each concerned with a different functionality. Of the many variations of the MVC architecture, I have decided for the one with a passive Model and a passive View. The roles of each section are following (Borini, 2016):

- **Model** – encompasses data resources, that is interfaces to databases, file systems, specific hardware, etc., and business logic that processes or creates data. A passive Model provides no notifications about changes in the data, and always has to be inquired again. The Model has no connections to the View or to the Controller.
- **View** – provides a user interface, typically graphical, which presents data and allows the user to interact with and modify the data. A passive View serves only for presentation, and has no knowledge of the data it presents or the meaning of user interactions. All logic concerned with the View should be in the Controller. The View has no connections to the Model.

- **Controller** – serves as an intermediary between the Model and the View. It requests data from the Model and uses it to fill the View. If it is necessary to keep track of changes in the data, the Model has to be inquired repeatedly. The Controller receives event notifications from the View and takes appropriate actions. It may change particular data in the Model or alter the View in reaction to a user operation.

The communication between the individual sections is depicted in Figure 13. Because Visdan uses the rich Qt framework to build the graphical user interface, some of the View logic may stay in the View instead of being moved to the Controller, for example input validation in form fields.

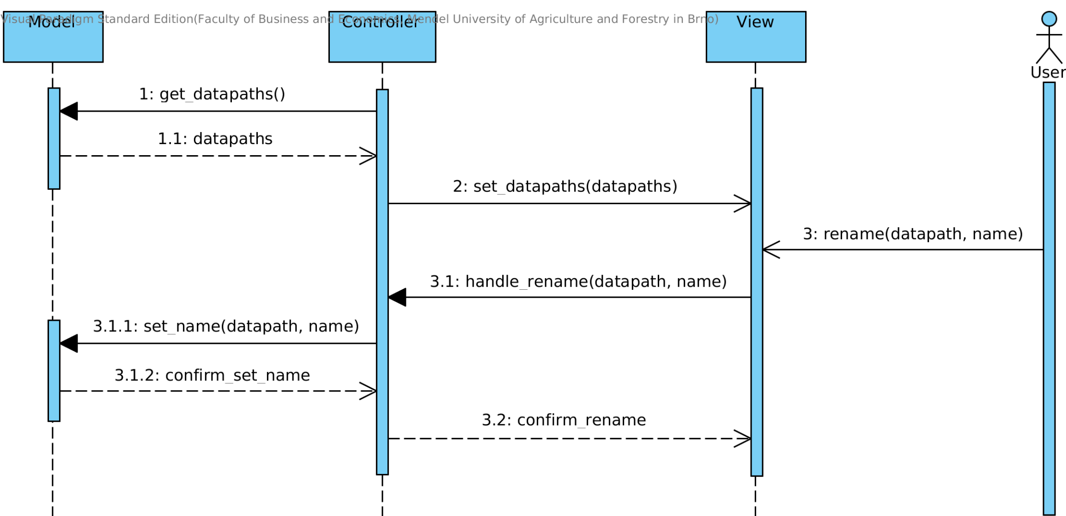


Figure 13: A sequence diagram capturing how the communication between the Model, the Controller, and the View is realized. Inspired by the illustration from section 2.3.2. Passive View from Understanding Model-View-Controller (Borini, 2016).

Class diagrams present in the following sections are rather conceptual and do not contain all methods and attributes present in the final implementation. Only the ones crucial for understanding of the functionality and cooperation between objects are mentioned. Also, classes of the same name may appear in more layers, because the layers will be put in separate namespaces in the implementation.

4.3 Model layer

As mentioned before, the Model layer provides access to data resources and functions for their processing. In Visdan, it takes care of the communication with the network controller, retrieving network data, and storing user-defined data related to the network. Its class diagram is depicted in Figure 14.

NetworkData aggregates network-related data from both the SDN controller and the user. Since it should be possible to use Visdan with various controllers, an inter-

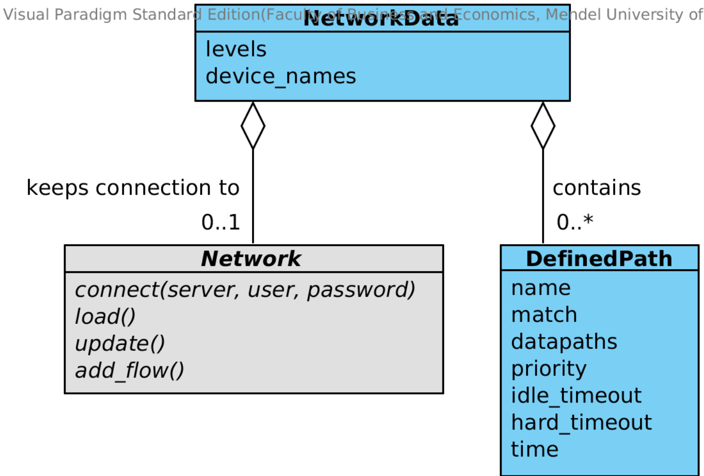


Figure 14: A class diagram of the Model layer.

face Network (described later in greater detail) was designed to allow for a unified approach to all controllers. To add support for another controller, a module implementing the interface has to be included, but no other changes have to be done in Visdan.

Other than the data retrieved from the controller, NetworkData stores user-defined information about hierarchy levels assigned to particular datapaths, names given to particular datapaths or nodes, and defined network paths.

DefinedPath is a data container that contains parameters of a defined path, which has a user-defined local name. Match fields common for all datapaths, involved in the path, are stored in a dictionary. At this point, following match fields are considered: IPv4 source and destination addresses, IP protocol (TCP or UDP), and IP protocol port numbers.

Fields specific for each datapath are stored in a dictionary. The fields include datapath DPID, negotiated OpenFlow version, and input and output ports for matched packets. These dictionaries are stored in an array, whose order reflects the order of datapaths in the path.

Flow-entry configuration parameters priority, idle timeout, and hard timeout are also stored as well as the timestamp of when the path was defined.

4.3.1 Network controller interface: data structure

The **Network** interface, as mentioned before, serves as a base for implementation of concrete interfaces for communication with various controllers. Figure 15 presents a class diagram of the abstract Network class accompanied by additional data structures it makes use of. Every interface has to implement methods regarding connection to the controller, data retrieval, and data pushing.

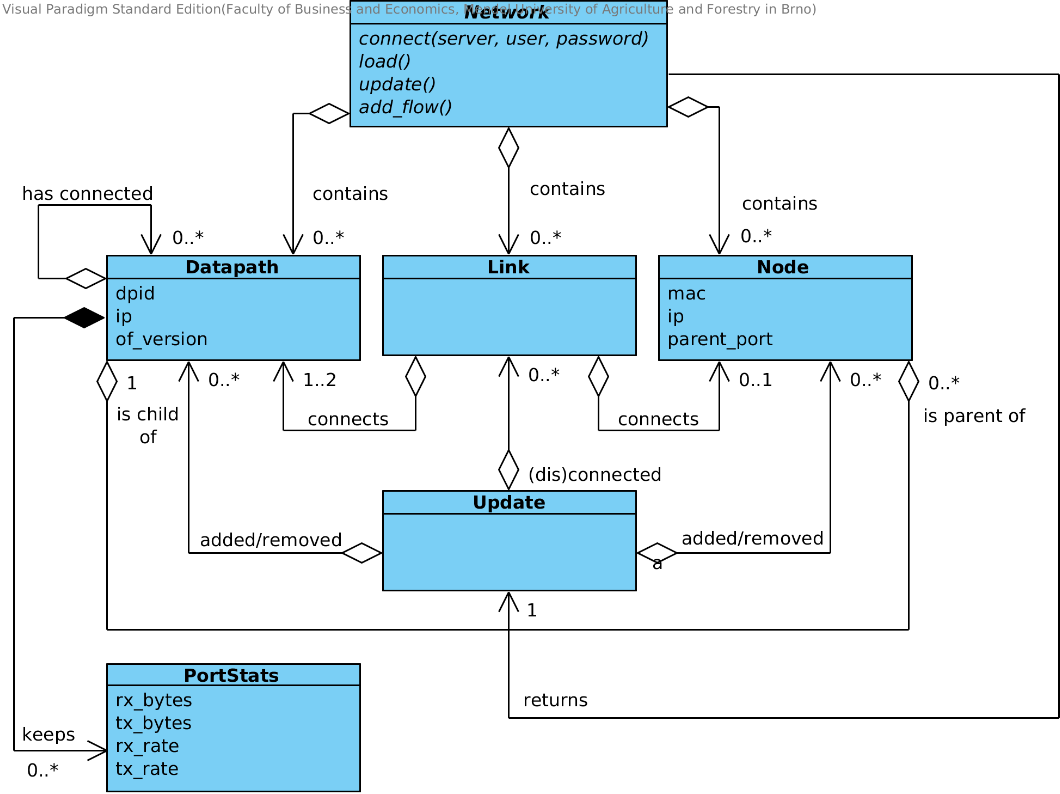


Figure 15: A class diagram of the Network interface used for communication with a controller, and additional data structures used by the interface.

In order to always provide Visdan with network data in the same format regardless of the actual connected controller, a set of data structures was designed that are to be used by the interfaces without any changes.

Datapath contains information about a datapath, and has two dictionaries for its active ports – one stores references to connected datapaths and nodes, and the other keeps traffic statistics in the **PortStats** containers. **Node** represents an end-user device, and contains a reference to its parent datapath. **Link** stores references to two interconnected devices – be it two datapaths, or a datapath and a node. **Update** serves as a return object given by Network when inquired for network changes. It contains references to added or removed datapaths or nodes, and connected or disconnected links. It describes the differences between the actual and the last saved state of the network.

Operation of algorithms for loading and updating data from the network through the controller is described in Appendix E.

Data structures used from the REST API of the HPE VAN SDN Controller are described in Appendix D.

4.4 View layer

The View Layer is concerned with presenting various data and control elements to the user. As for Visdan, this is realized through a graphical interface. A design for the layout of the application window is depicted in Figure 16, while its class diagram is depicted in Figure 17. The structure of the classes closely corresponds to the structure of the graphical interface and how the elements are put together. All the classes subclass classes of the Qt framework and are configured and customized for the needs of Visdan.

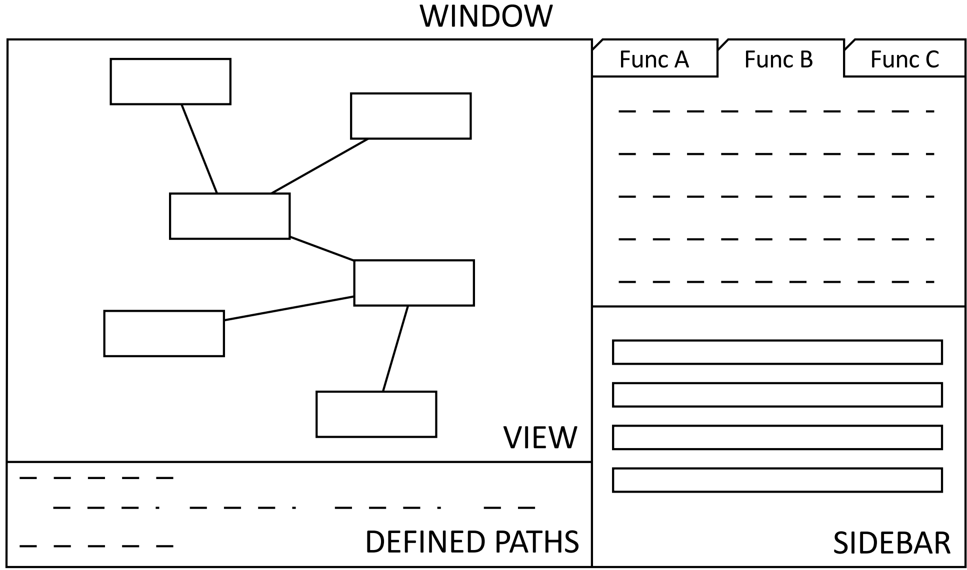


Figure 16: A layout of the graphical user interface of Visdan.

Window represents the application window, and is the core element that contains and positions the key components.

In the center, there is **View** that provides network visualization, the main functionality of Visdan. The view itself does not contain any data, but needs to have a particular scene attached, which it then displays. **Scene** is independent of View, and works like a canvas on which various elements may be drawn. These elements are represented by classes **Datapath**, **Node**, **Link**, and **Stats**. When a change occurs in the scene, the associated view is automatically refreshed. The view also captures mouse events from the user, which allows for the canvas elements to be, for example, arbitrarily moved by dragging or selected by clicking.

On the right side, there is **Sidebar**. The sidebar serves as a home for different panels, which can be switched using a tab bar on the top. As for now, there are four panels. **Connection** provides a form to choose the desired controller interface, fill the necessary information, and connect to the controller. **Detail** presents extended information about a datapath or a node. **PathDefinition** presents a tabular overview of the elements already chosen for the path, and then provides a form to configure

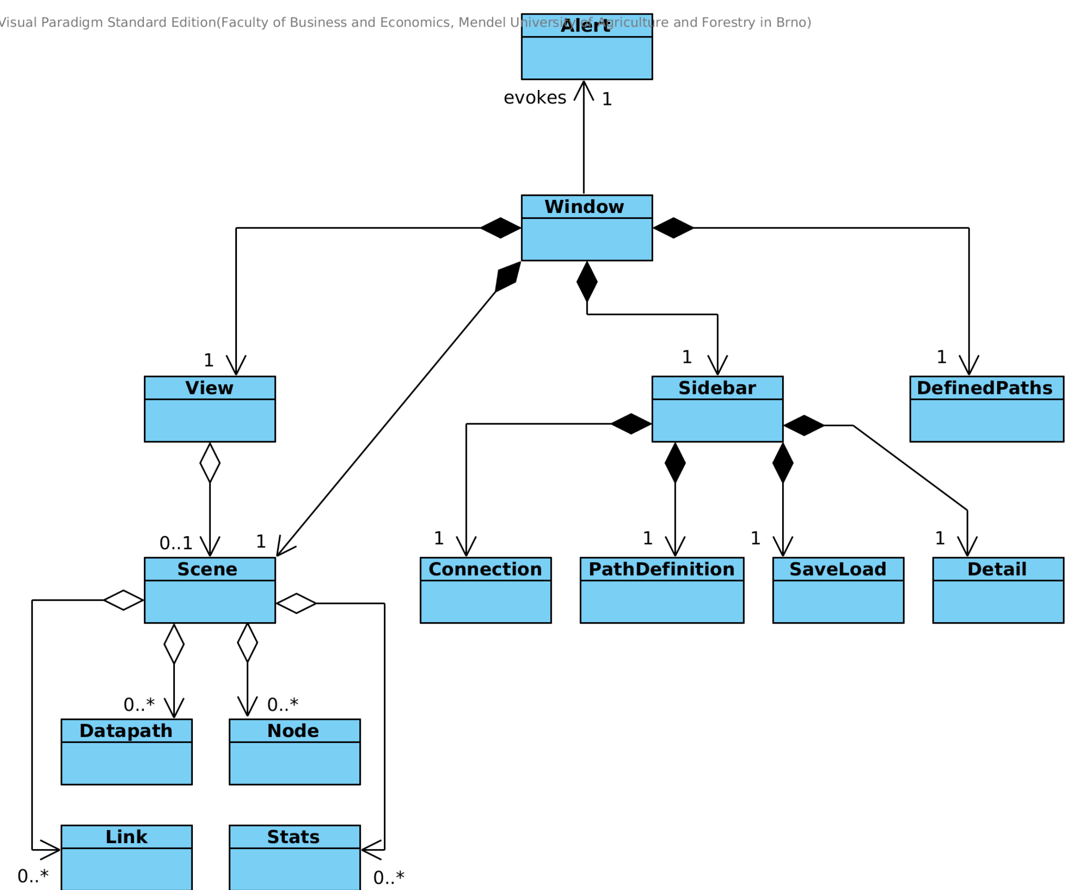


Figure 17: A class diagram of the View Layer.

the details of the path definition once the choosing is complete. **SaveLoad** allows the user to export or import custom-defined data to or from an external file.

Below the view, there are defined paths. **DefinedPaths** offers a tree-structured list of user-defined paths, in which each item may be unpacked to show the particular elements involved in the path. The path may also be deleted from the list.

Alert is an auxiliary class that is used when it is necessary to show warning or question dialogs to the user.

4.5 Controller layer

The Controller layer is to ensure the connection between the Model layer and the View layer, and to provide appropriate reactions to user inputs. In Visdan, neither the Model layer nor the View layer keep direct references to the Controller layer, but the Controller layer keeps direct references to both of them. However, to inform the Controller layer about any occurring user events, the View layer has to be connected

to it. Visdan utilizes a feature called *signals and slots* of the Qt framework for this purpose.

Signals and slots can be well exploited to realize the communication between the View layer and the Controller layer, which is schematically depicted in Figure 18. Signals are emitted by various elements in the View layer in reaction to a change or a user action. They may be plain or carry some values. For example, a text form field (Qt class `QTextEdit`) emits a signal every time its content changes (signal `textChanged()`). Slots are functions or methods that may be evoked by signals connected to them. For example, the previously mentioned text field has a slot that clears its content (slot `clear()`), which might, for example, be evoked in reaction to a user clicking the Cancel button.

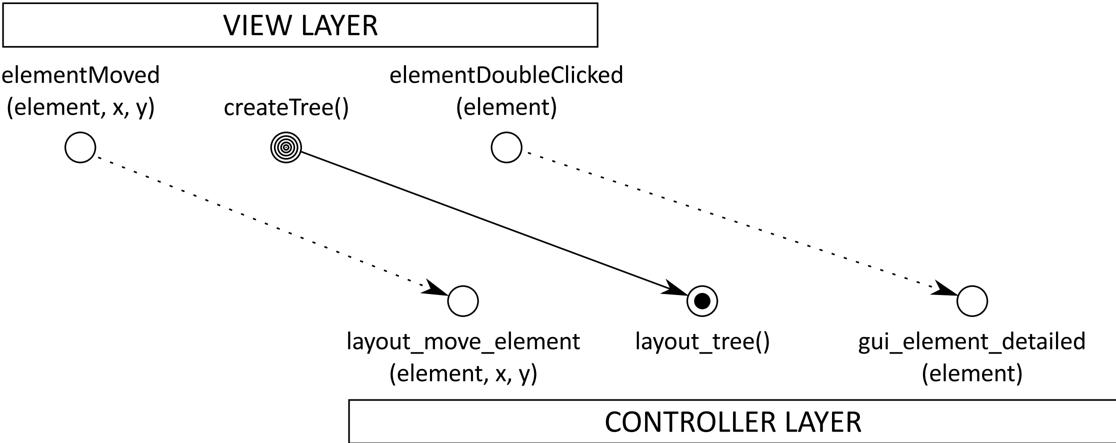


Figure 18: A schema of the communication between the View layer and Controller layer as realized through signals and slots. The signal `createTree()` has been emitted, and the slot `layout_tree()` is evoked and changes the visualization to a tree layout.

As for Visdan, there is a class **Controller** (as depicted in the class diagram in Figure 19), which provides slots for all concerned signals emitted by objects in the View layer and takes care of having the corresponding objects and methods process the events and react on them. It keeps a reference to Window, the main window of the graphical interface, to NetworkData, the aggregate class of the Model layer, and to Path and Layout, which are described below.

UpdateWorker is an auxiliary class that is used to run network updates in a separate thread. This is to prevent the graphical interface being stuck while processing changes in the network.

Path keeps track of the current path definition. It records the datapaths and nodes selected by the user for the path, and manages the table in the PathDefinition panel, in which the so far selected elements are listed.

Layout serves to transform the data retrieved from a network controller into a visualization useful to and easily readable by a user. Once processed, graphical elements

corresponding to the network elements are added to the scene in the View layer. By default, the graphical elements are arranged in a randomly generated graph structure. If the user defines a hierarchy level distribution or core for at least one datapath, the visualization may be transformed into a tree-like structure. Because the information provided by the user may be concerned only with a part of the connected network, elements that are not included by the tree are positioned using the graph algorithm. This may happen, for example, when there are two separate networks connected to the controller, but the user designates the hierarchy only for one of them.

Auxiliary classes **TreeDatapath** and **TreeNode** are used to create a tree-like data structure necessary for the drawing itself. This data structure is not actually a tree, even though it takes some of it characteristics. There are two facts in which it mainly differs. Firstly, it may have more than one root element (for example a core switch doubled for high availability), and secondly, it keeps only one predecessor for each datapath, although in reality it may be connected to more elements superior in the hierarchy. This is because every element has to belong to one specific sub-tree of a particular datapath. The second difference relates only to the data structure, in the visualization all links are depicted.

The **Translation** class is a data container with several dictionaries that store the relations between network elements and their graphical counterparts.

Algorithms that are used to create the tree-like data structure for visualization and draw it are described in Appendix F.

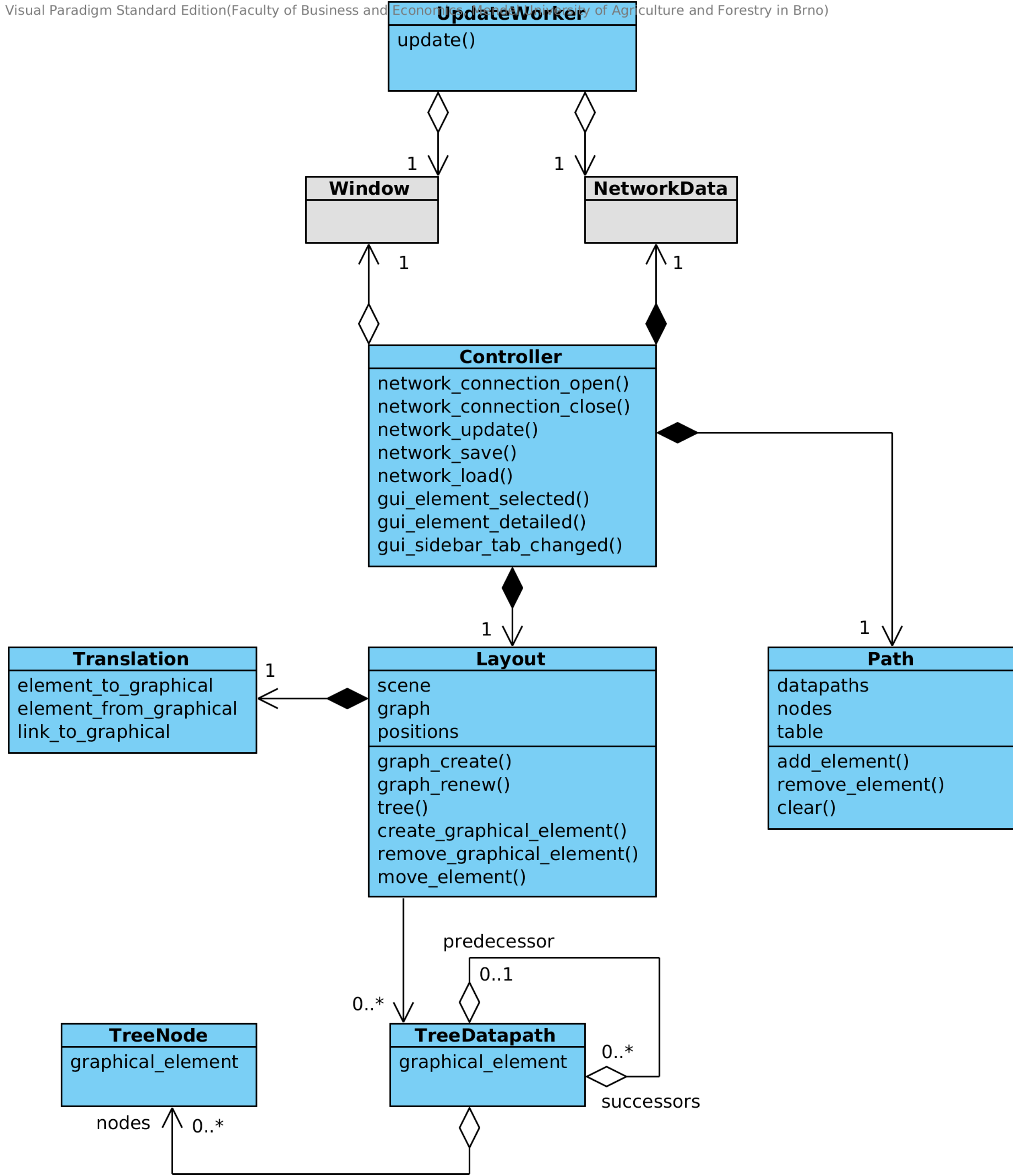


Figure 19: A class diagram of the Controller layer. Not all attributes and methods are shown, but only an excerpt for elementary understanding of the purpose of the classes.

5 Implementation

This chapter describes how key features of Visdan are implemented. Relevant excerpts from the source code are explained, the whole application code is then present in the digital appendices.

5.1 Data retrieval

5.1.1 Connecting to the controller

Since Visdan uses the REST API, it does not keep a permanent connection to the controller. Instead, the connection is based on requests. In the beginning, it is necessary to obtain an authentication token, which is checked by the controller every time a request is made. The controller keeps information about issued tokens and allows access to its resources only if the token is valid.

All communication with the controller is carried out by the network interface module `interfaces.hpe` through its class `Network`. To approach the controller, the module *hp-sdn-client* is used. In the code it is imported as `hp`, which is then used as a prefix when objects and methods of this module are called.

Soure code 1: Connecting to the controller.

```
class Network(base.BaseNetwork):
    ...
    def connect(self, server, user, password):
        # Obtain an authentication token
        auth = hp.XAuthToken(server=server, user=user, password=password)
        # Keep a reference to the API accessed with the token
        self._api = hp.Api(controller=server, auth=auth)
```

5.1.2 Loading data

To load network data from the controller into the application, it is necessary to retrieve the whole section of network data and then process each of its elements in a cycle. As for datapaths, the required information is saved in a `Datapath` object, which is then stored in its superior `Network` object.

Soure code 2: Loading datapaths (method load – part 1).

```
class Network(base.BaseNetwork):
    ...
    def load(self):
        # Retrieve datapaths from the controller
        datapaths = self._api.get_datapaths()

        for dp in datapaths:
            # Save the datapath
```

```
            self._datapaths[dp.dpid] = base.Datapath(dp.dpid, dp.device_ip,
                dp.negotiated_version)
        ...
```

As for nodes, the required information is stored in a `Node` object, but also the connection between the node and its parent datapath has to be stored. In `Node`, a reference to the parent `Datapath` object is kept together with the port number. As well a reference to `Node` is kept in `Datapath`. Information about the connection is also stored in a `Link` object. Both `Node` and `Link` then have to be saved in `Network`.

Soure code 3: Loading nodes (method load – part 2).

```
        ...
        # Retrieve nodes from the controller
        nodes = self._api.get_nodes()

        for n in nodes:
            # Obtain the Datapath object for the parent datapath
            parent = self._datapaths[n.dpid]
            # Save the node
            node = base.Node(n.mac, n.ip, parent, n.port)
            self._nodes[n.mac] = node
            # Connect the node to its parent datapath
            parent.connect(n.port, node)
            # Save the link
            link = base.Link(parent, node)
            self._links.add(link)
        ...
```

As for links, information about the connection is stored in a `Link` object as well as in the involved datapaths, which have to be mutually connected. `Link` is then saved in `Network` as mentioned before.

Soure code 4: Loading links (method load – part 3).

```
        ...
        # Retrieve links from the controller
        links = self._api.get_links()

        for l in links:
            # Save the link
            link = base.Link(self._datapaths[l.src_dpid],
                             self._datapaths[l.dst_dpid])
            self._links.add(link)
            # Mutually connect the involved datapaths
            self._datapaths[l.src_dpid].connect(l.src_port,
                                                self._datapaths[l.dst_dpid])
            self._datapaths[l.dst_dpid].connect(l.dst_port,
                                                self._datapaths[l.src_dpid])
```

5.1.3 Updating data

Data updates make use of the functionality used for loading data. All network data is downloaded again and then compared with the data saved in the `Network` object. This is done periodically, and the involved processes have to run in separate threads to not block the graphical interface.

In the beginning of the algorithm, datapaths and nodes are retrieved, and several sets are initialized. All new elements (devices or links) are added to these sets, which are at the end compared with the original content of Network to find and remove elements that are not present in the network anymore.

The `Update` object stores information about changes in the network. It has four lists – add, remove, connect, and disconnect. Whenever a datapath or a node is added or removed, its object is added to the corresponding list in order to to be added to or removed from the visualization. In the same manner, links are added to lists connect and disconnect.

Soure code 5: Preparing network update (method update – part 1).

```
class Network(base.BaseNetwork):
    ...
    def update(self, interval):
        # Initialize the container for updates
        update = base.Update()

        # Retrieve datapaths from the controller
        datapaths = self._api.get_datapaths()
        # Retrieve nodes from the controller
        nodes = self._api.get_nodes()

        # Prepare sets that will serve to compare the new network state to the
        # original state
        new_dpids = set()
        new_macs = set()
        new_links = set()
        ...
```

In the next step, retrieved datapaths are iterated, and a `Datapath` object is created for each one. If a datapath with the same DPID already exists in `Network`, the old and the new object are compared. In case there are any differences, that is any parameter has changed, the old `Datapath` object is removed. In the same way the new object is then added to `Network`. DPIDs stored in `Network` are then compared with the set of new DPIDs, and any leftovers are removed.

Soure code 6: Updating datapaths (method update – part 2).

```
        ...
        # Iterate retrieved datapaths
        for dp in datapaths:
            new = base.Datapath(dp.dpid, dp.device_ip, dp.negotiated_version)
```

```
        new_dpids.add(dp.dpid)

        # Is this datapath (with the same DPID) already stored?
        # Does the new object differ from the original object?
        # Remove the original datapath if it differs from the new one.
        if dp.dpid in self._datapaths and self._datapaths[dp.dpid] != new:
            update.remove.append(self._datapaths[dp.dpid])
            del self._datapaths[dp.dpid]

        # Save the datapath in case it is not stored
        # Also applies if it was removed in the previous step
        if dp.dpid not in self._datapaths:
            update.add.append(new)
            self._datapaths[dp.dpid] = new

    # Remove datapaths that were in the original network state, but are not
    # in the new state
    leftover_dp = self._datapaths.keys() - new_dpids
    for i in leftover_dp:
        update.remove.append(self._datapaths[i])
        del self._datapaths[i]
    ...
```

Nodes are compared in exactly the same way, there is only one difference. Since nodes are connected to datapaths, they have to be disconnected to free the port before they are removed. When a new `Node` is added, it has to be connected to its parent `Datapath` and a `Link` object has to be created, as it was when loading nodes.

Soure code 7: Updating nodes (method update – part 3).

```
    ...
    # Iterate retrieved nodes
    for n in nodes:
        new = base.Node(n.mac, n.ip, self._datapaths[n.dpid], n.port)
        new_macs.add(n.mac)

        # Is this node (with the same MAC address) already stored?
        # Does the new object differ from the original object?
        # Remove the original node if it differs from the new one.
        if n.mac in self._nodes and self._nodes[n.mac] != new:
            self._nodes[n.mac].parent.disconnect_device(self._nodes[n.mac])
            update.remove.append(self._nodes[n.mac])
            del self._nodes[n.mac]

        # Save and connect the node in case it is not stored
        # Also applies if it was removed in the previous step
        if n.mac not in self._nodes:
            self._nodes[n.mac] = new
            update.add.append(new)
            new.parent.connect(n.port, new)
```

```
        link = base.Link(new.parent, new)
        new_links.add(link)
        # Save the link in case it is not already stored
        if link not in self._links:
            self._links.add(link)
            update.connect.append(link)

    # Remove and disconnect nodes that were in the original network state,
    # but are not in the new state
    leftover_n = self._nodes.keys() - new_macs
    for i in leftover_n:
        self._nodes[i].parent.disconnect_device(self._nodes[i])
        update.remove.append(self._nodes[i])
        del self._nodes[i]
    ...
```

Next, datapaths in the new network state are iterated in order to update their links with other datapaths and to update their port statistics. For every link it is checked whether the other datapath is connected to the checked datapath and whether the port is the same as it was before. If the connection is the same, the other datapath cannot be connected to the same port again because it would erase statistics of this port.

Ports are then checked for leftovers, and any port connected to a `Datapath` object is cleared because it is not connected anymore. In case the port was connected to a `Node` object, the port would be preserved. Connections between datapaths and nodes are not included in links downloaded from the controller, and were therefore handled separately in the previous section of the algorithm. For every active (not cleared) port of the datapath, port statistics are then downloaded and updated.

Soure code 8: Updating datapath links and port statistics (method update – part 4).

```
    ...
    # Iterate datapaths in the new state
    for dpid in self._datapaths:
        # Retrieve links for this datapath from the controller
        links = self._api.get_links(dpid)
        # Prepare a set that will serve to compare the connected ports on
        # this datapath from the new network state to the ones from
        # the original state
        new_ports = set()
        for li in links:
            # Find out whether this datapath is in the link as source
            # or as destination
            if dpid == li.src_dpid:
                this = li.src_dpid
                this_port = li.src_port
                other = li.dst_dpid
                other_port = li.dst_port
            else:
                this = li.dst_dpid
```

```
                this_port = li.dst_port
                other = li.src_dpid
                other_port = li.src_port

            link = base.Link(self._datapaths[li.src_dpid],
                             self._datapaths[li.dst_dpid])
            new_links.add(link)
            new_ports.add(this_port)

            # Connect the datapaths
            # In case the other datapath has been already connected and
            # its port has not changed, do not connect the datapaths again
            if (self._datapaths[this].port_of_device(
                    self._datapaths[other]) != this_port):
                self._datapaths[li.src_dpid].connect(
                    li.src_port, self._datapaths[li.dst_dpid])
                self._datapaths[li.dst_dpid].connect(
                    li.dst_port, self._datapaths[li.src_dpid])

            # Save the link in case it is not already stored
            if link not in self._links:
                self._links.add(link)
                update.connect.append(link)

        # Clear ports that were connected to a datapath in the old state,
        # but are not anymore in the new state
        leftover_p = self._datapaths[dpid].ports_devices.keys() - new_ports
        for p in leftover_p:
            if isinstance(self._datapaths[dpid].ports_devices[p],
                          base.Datapath):
                self._datapaths[dpid].disconnect_port(p)

        # Retrieve port statistics from the controller
        stats = self._api.get_port_stats(dpid)
        for s in stats:
            # Update port statistics for each active port.
            # The check is necessary because a local port for out-of-band
            # switch-controller communication would not be included in
            # the ports, but would have its statistics.
            if s.port_id in self._datapaths[dpid].ports_stats:
                self._datapaths[dpid].update_port_stats(s.port_id,
                    interval, s.rx_bytes, s.tx_bytes)
    ...
```

For the last step, new and old Link objects are compared and leftovers removed. At this point, the updating process is finished and the `Update` object containing all changes is returned to the calling process.

Soure code 9: Updating datapath links and port statistics (method update – part 5).

```
    ...
    # Remove links that were in the original network state, but are not
```

```
        # in the new state
        leftover_li = self._links - new_links
        for i in leftover_li:
            update.disconnect.append(i)
        self._links = new_links

        return update
```

5.2 Data visualization

5.2.1 Graph

Visualization of network data is realized by class `Layout`. It employs a library *networkX*, which provides graph data structures and algorithms that operate on them. It is necessary to transform the retrieved network data into a graph in order to make it possible to run a positioning algorithm on this data. At first, nodes and edges have to be added to the graph.

Soure code 10: Adding nodes and edges to the graph (method graph_create – part 1).

```
class Layout:
    ...
    def graph_create(self, net_elements, net_links):
        # Iterate datapaths and nodes
        for net_element in net_elements:
            # Create a graphical element for the network element
            grph_element = self.create_graphical_element(net_element)
            # Add the graphical element to the graph
            self.graph.add_node(grph_element)

        # Initialize a~list for graphical links
        grph_links = []
        # Iterate links
        for net_link in net_links:
            # Create a graphical link for the network link
            grph_link = self.create_graphical_link(net_link)
            # Add to the graph an edge connecting graphical counterparts of the
            # connected network elements
            grph_element1 = grph_link.source
            grph_element2 = grph_link.destination
            self.graph.add_edge(grph_element1, grph_element2)
            # Add the link to a~list of graphical links
            grph_links.append(grph_link)
        ...
```

Once the graph data structure is established, it is possible to generate positions for every vertex. A Fruchterman-Reingold force-directed algorithm is used to distribute the nodes in a manner that respects their mutual connections and positions them accordingly. By default, the nodes are positioned in a box of size [0, 1] x [0, 1],

which is not useful when the scene, on which the graphical elements are placed, uses real pixel dimensions. Therefore, a scale factor is used, which was determined experimentally and is based on the number of nodes in the graph.

Soure code 11: Calculating positions of graph nodes (method graph_create – part 2).

```
...
scale = 100 * math.pow(1.15, self.graph.number_of_nodes())
self.positions = nx.spring_layout(self.graph,
   scale=scale)
...
```

With the positions calculated, it only remains to place the graphical elements on the scene at these positions. The positioning algorithm returns a dictionary where nodes serve as keys, and for values are used arrays with 2 values representing x and y coordinates. When all graphical elements are settled, graphical links may be added to connect them.

Soure code 12: Calculating positions of graph nodes (method graph_create – part 3).

```
...
for grph_element in self.positions:
   self.scene.add_element(grph_element, self.positions[grph_element])

for grph_link in grph_links:
   self.scene.add_link(grph_link)
```

5.2.2 Tree

If the user declares particular datapaths as distribution or core, these may serve as root elements for a tree-like structure, into which the visualization may be transformed. A breadth-first search algorithm is used to create a supplementary data structure using `TreeDatapath` and `TreeNode` objects based on the specified roots.

Soure code 13: Creating a tree-like data structure.

```
class Layout:
   ...
   def tree(self, net_treetop):
      top = True
      row = set(net_treetop)
      passed = set()
      next_row = set()
      predecessors = {}
      tree_roots = []
      while len(row) != 0:
         # Get next Datapath object in the row
         net_datapath = row.pop()
         # Mark it is as processed
```

```
        passed.add(net_datapath)

        # Obtain a graphical element for the given network datapath
        grph_datapath = self.translation.element_to_graphical[net_datapath]
        # Create a tree representation of the datapath
        tree_datapath = TreeDatapath(grph_datapath)
        # If the first layer is being processed, save the tree object into
        # a list of root tree objects
        if top:
            tree_roots.append(tree_datapath)

        # Iterate devices connected to the datapath
        for device in net_datapath.ports_devices.values():
            # If the device is a datapath, add it to the next row unless it
            # is already there or unless it has been already processed or is
            # about to be processed in this row.
            if isinstance(device, data.interfaces.Datapath):
                if device not in (passed | row | next_row):
                    next_row.add(device)
                    # A tree object for the datapath will be created once it is
                    # processed by the main cycle. The relationship between
                    # these datapaths will have to be stored
                    # when both of them have a tree object. For now, it is
                    # temporarily stored in predecessors dictionary.
                    predecessors[device] = tree_datapath
            # If the device is a node, a TreeNode object is created
            # immediately and connected to parent TreeDatapath.
            elif isinstance(device, data.interfaces.Node):
                # Obtain a graphical element for the given network node
                grph_node = self.translation.element_to_graphical[device]
                # Create a tree representation of the node
                tree_node = TreeNode(grph_node)
                # Connect it to its parent datapath
                tree_datapath.nodes.append(tree_node)

        # At this point, the relationships between datapaths are finalized.
        # Parent TreeDatapath is connected to this Tree Datapath as.
        # a predecessor. This TreeDatapath is added to successors of parent
        # TreeDatapath.
        if net_datapath in predecessors:
            tree_datapath.predecessor = predecessors[net_datapath]
            predecessors[net_datapath].successors.append(tree_datapath)

        # If the row has been processed
        if len(row) == 0:
            # Every row except for the first one is not considered a root row
            top = False
            # The next row is taken to be processed
            row = next_row
            next_row = set()
...
```

Now, when the tree data structure is prepared, it can be processed through a depth-first search algorithm, which positions the graphical elements in such a way, that the result looks like a tree. In the algorithm, positions are calculated in matrix fields, which are an the end converted to real coordinates.

Soure code 14: Drawing the tree-like data structure.

```
class Layout:
    ...
    def tree(self, net_treetop):
        ...
        # Set initial coordinates
        pos = Coords(0, 0)
        # Iterate root elements
        for root in tree_roots:
            # Process every root through the depth-first search algorithm
            # The final position is used as the initial position for the next
            # neighboring root
            pos = self.tree_draw(root, pos)

    def tree_draw(self, root, start_pos):
        # If the element is a datapath
        if isinstance(root, TreeDatapath):
            # Obtain the element dimensions in matrix units
            width, height = gui.scene.SceneMatrix.element_matrix_size(
                root.grph_element)

            # Calculate the initial position for the next row based on the height
            # of the element plus spacing
            succ_y = start_pos.y + height + 7
            succ_pos = Coords(start_pos.x, succ_y)
            # Define horizontal end position in case there are no successors
            end_x = start_pos.x + width
            # Iterate successors of this datapath
            for succ in root.successors:
                # The end position is used as the initial position for the
                # next successor
                succ_pos = self.tree_draw(succ, succ_pos)
                # Keep the farthest end x coordinate
                end_x = max(end_x, succ_pos.x)
            # Compensation for spacing after the last element of the sub-tree
            end_x = end_x - 5 if end_x > (start_pos.x + width) else end_x

            # Keep the position of the first node
            first = succ_pos
            # Iterate nodes connected to this datapath
            for i, node in enumerate(root.nodes, start=1):
                succ_pos = self.tree_draw(node, succ_pos)
                end_x = max(end_x, succ_pos.x)
                # Nodes are positioned in a~grid with tree nodes on a~row
                if i % 3 == 0:
                    node_width, node_height = (
```

```
                gui.scene.SceneMatrix.element_matrix_size(
                node.grph_element))
            new_y = succ_pos.y + node_height + 3
            succ_pos = Coords(first.x, new_y)

        # When all successors and nodes are processed, the sub-tree of
        # this datapath is completely drawn. The datapath is then centered
        # above its sub-tree.
        offset_x = int(math.floor((end_x - start_pos.x - width)/2))
        position = Coords(start_pos.x + offset_x, start_pos.y)

        # Convert matrix coordinates to real coordinates
        new_scene_pos = gui.scene.SceneMatrix.pos_to_scene(position)
        # Position the graphical element on the scene
        root.grph_element.setPos(new_scene_pos.x, new_scene_pos.y)
        self.positions[root.grph_element] = (new_scene_pos.x,
            new_scene_pos.y)

        # Return the end coordinates with spacing included
        end_pos = Coords(end_x + 5, start_pos.y)
        return end_pos
    # If the element is a~node, it is only placed on the scene using the
    # initial coordinates and its end coordinates are returned.
    elif isinstance(root, TreeNode):
        new_scene_pos = gui.scene.SceneMatrix.pos_to_scene(start_pos)
        root.grph_element.setPos(new_scene_pos.x, new_scene_pos.y)
        self.positions[root.grph_element] = (new_scene_pos.x,
            new_scene_pos.y)

        width, height = gui.scene.SceneMatrix.element_matrix_size(
            root.grph_element)

        end_pos = Coords(start_pos.x + width + 1, start_pos.y)
        return end_pos
```

5.3 Path definition

When a traffic path is selected and configured in the graphical interface, it has to be decomposed into individual flow entries that are then pushed to particular datapaths. This decomposition takes place in class `NetworkData`. `Controller` takes data from the path definition form and hands it over to `NetworkData`.

Mandatory fields are priority, idle timeout, and hard timeout. Optional fields are IP protocol and TCP or UDP source and destination port numbers.

Soure code 15: Decomposing a path.

```
class NetworkData:
    ...
    def add_path(self, src, dst, datapaths, mandatory, optional):
```

```
    ...
    # Count datapaths
    dp_count = len(datapaths)
    # Set the source end node as the preceding device for the first datapath
    pred = src
    # Iterate datapaths involved in the path
    for index in range(dp_count):
        # Set current datapath
        curr = datapaths[index]
        # Check if this datapaths is the last one
        # If yes than the succeeding device is the destination end node
        if index == dp_count-1:
            succ = dst
        else:
            succ = datapaths[index+1]

        # Input port = port where the preceding device is connected
        in_port = curr.port_of_device(pred)
        # Output port = port where the succeeding device is connected
        out_port = curr.port_of_device(succ)

        # Send the flow entry definition to the network interface
        self.net_controller.add_flow(curr, src.ip, dst.ip,
            in_port, out_port, mandatory['priority'],
            mandatory['idle_timeout'], mandatory['hard_timeout'],
            **optional)


        # Set the current datapath as preceding
        pred = curr
    ...
```

In the network interface, the flow entry data is converted into a structure required by the API, which is then pushed to the specified datapath.

Soure code 16: Pushing a flow entry into a datapath.

```
class Network(base.BaseNetwork):
    ...
    def add_flow(self, datapath, src_ip, dst_ip, in_port, out_port,
            priority, idle_timeout, hard_timeout, **kwargs):
        # Prepare match fields of the flow entry
        match_args = {}
        match_args['ipv4_src'] = src_ip
        match_args['ipv4_dst'] = dst_ip
        match_args['eth_type'] = 'ipv4'
        match_args['in_port'] = in_port

        # Add IP protocol field if it equals to TCP or UDP
        if kwargs['ip_protocol'] is enums.IpProtocol.tcp:
            match_args['ip_proto'] = 'tcp'
            # Add TCP source port if given
```

```
        if kwargs['tcp_src']:
            match_args['tcp_src'] = kwargs['tcp_src']
        # Add TCP destination port if given
        if kwargs['tcp_dst']:
            match_args['tcp_dst'] = kwargs['tcp_dst']
    elif kwargs['ip_protocol'] is enums.IpProtocol.udp:
        match_args['ip_proto'] = 'udp'
        # Add UDP source port if given
        if kwargs['udp_src']:
            match_args['udp_src'] = kwargs['udp_src']
        # Add UDP destination port if given
        if kwargs['udp_dst']:
            match_args['udp_dst'] = kwargs['udp_dst']

    # Convert match fields to the Match datatype of hp-sdn-client
    match = hp.datatypes.Match(**(match_args))
    # Create an action using the Action datatype of hp-sdn-client
    action = hp.datatypes.Action(output=out_port)

    # Prepare configuration parameters of the flow entry
    flow_args = {}
    flow_args['priority'] = priority
    flow_args['idle_timeout'] = int(idle_timeout)
    flow_args['hard_timeout'] = int(hard_timeout)

    # Construct messages using the Flow datatype of hp-sdn-client
    # Construct a message for OpenFlow v. 1.0
    if datapath.of_version == "1.0.0":
        # Version 1.0.0 uses Actions
        flow = hp.datatypes.Flow(match=match, actions=action, **flow_args)
    # Construct a message for OpenFlow v. 1.3
    else:
        # Version 1.3.0 uses Instructions with Actions inside
        instruction = hp.datatypes.Instruction(apply_actions=action)
        flow = hp.datatypes.Flow(match=match, instructions=[instruction],
                                 **flow_args)

    # Push the flow entry to the datapath
    self._api.add_flows(datapath.dpid, flow)
```

5.4 Graphical user interface realization

Concerning the graphical user interface, it was realized according to the design concept. It uses standard window and form elements for most of the functions. As for network elements drawn on the canvas, their graphics is simple yet functional. A few illustration follow.

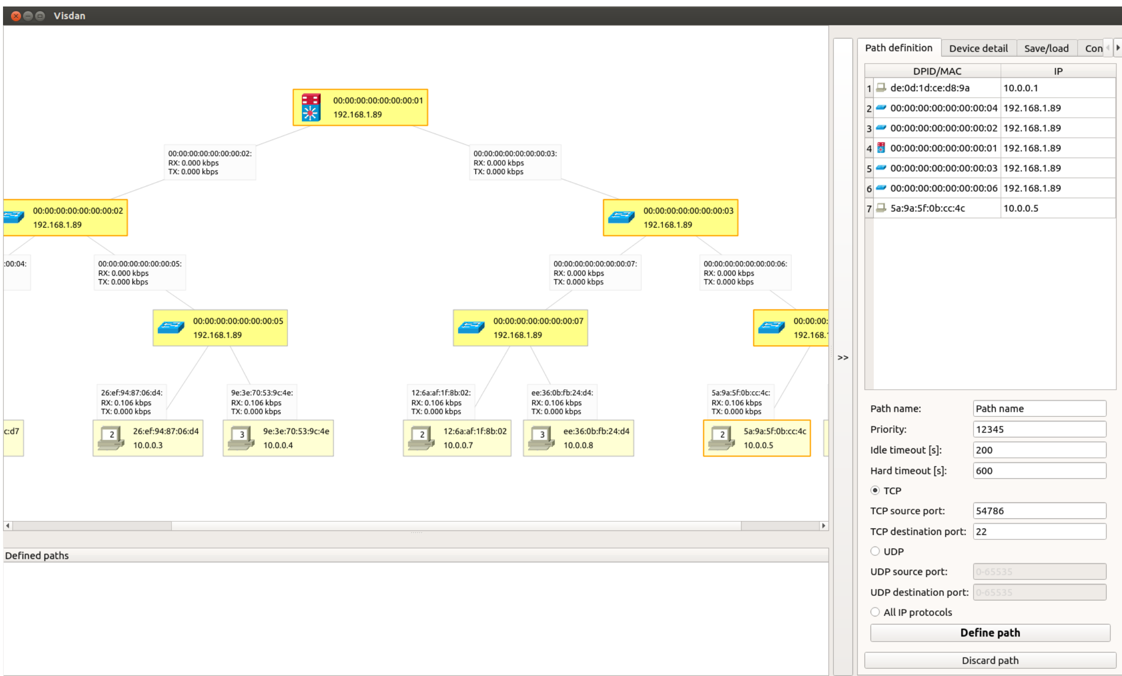


Figure 20: Vidan application window. The complete screenshot is attached in digital appendices.

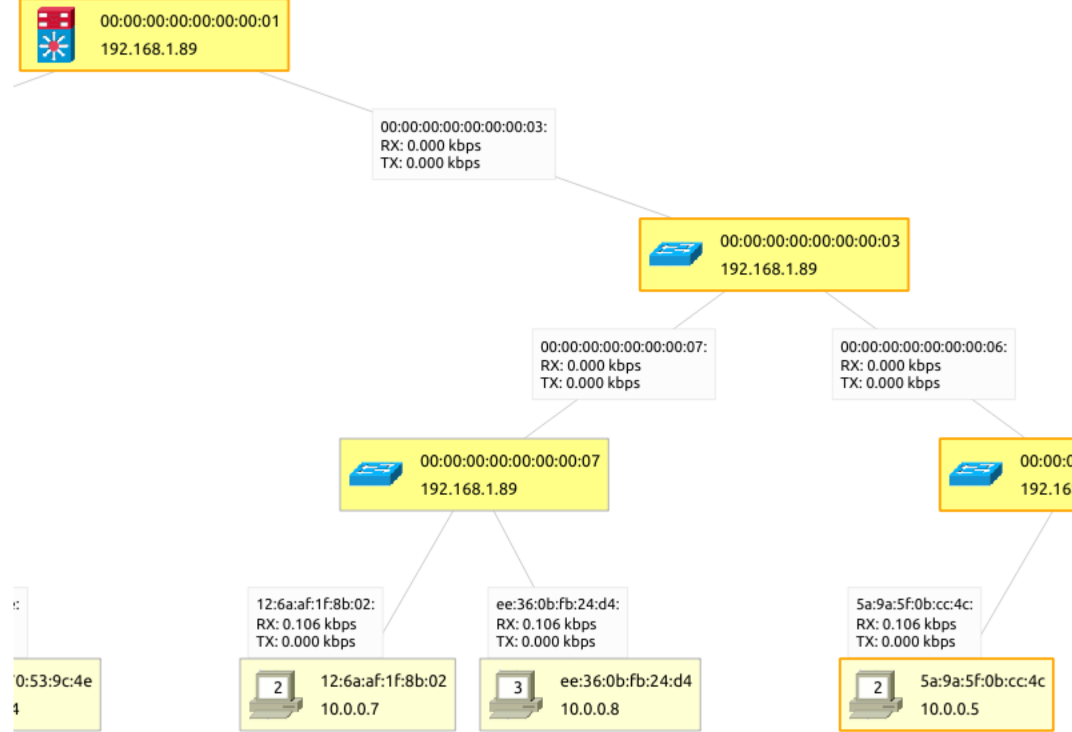


Figure 21: Detail of network visualization. Elements with an extra frame are selected in the defined path. The complete screenshot is attached in digital appendices.

Path definition | Device detail | Save/load | Con

	DPID/MAC	IP
1	de:0d:1d:ce:d8:9a	10.0.0.1
2	00:00:00:00:00:00:00:04	192.168.1.89
3	00:00:00:00:00:00:00:02	192.168.1.89
4	00:00:00:00:00:00:00:01	192.168.1.89
5	00:00:00:00:00:00:00:03	192.168.1.89
6	00:00:00:00:00:00:00:06	192.168.1.89
7	5a:9a:5f:0b:cc:4c	10.0.0.5

Figure 22: Detail of path definition. The table provides an overview of network elements that have been selected so far. The complete screenshot is attached in digital appendices.

Path name: Path name
Priority: 12345
Idle timeout [s]: 200
Hard timeout [s]: 600
TCP
TCP source port: 54786
TCP destination port: 22
UDP
UDP source port: 0-65535
UDP destination port: 0-65535
All IP protocols
Define path
Discard path

Figure 23: Detail of path definiton. The form serves for configuration of the path. The complete screenshot is attached in digital appendices.

Defined paths

- Path name
  - Match fields
    - Source IP address: 10.0.0.1
    - Destination IP address: 10.0.0.5
    - IP protocol: TCP
    - TCP source port: 54786
    - TCP destination port: 22
  - Datapaths
    - 00:00:00:00:00:00:00:04
      - OpenFlow version: 1.0.0
      - Input port: 2
      - Output port: 1
    - 00:00:00:00:00:00:00:02
      - OpenFlow version: 1.0.0
      - Input port: 2
      - Output port: 1
    - 00:00:00:00:00:00:00:01
      - OpenFlow version: 1.0.0
      - Input port: 1
      - Output port: 2
    - 00:00:00:00:00:00:00:03
      - OpenFlow version: 1.0.0
      - Input port: 1
      - Output port: 2
    - 00:00:00:00:00:00:00:06
      - OpenFlow version: 1.0.0
      - Input port: 1
      - Output port: 2
  - Priority: 12345
  - Idle timeout: 200
  - Hard timeout: 600
  - Date of definition: 2016/12/28 17:24:28

Figure 24: Detail of the overview of defined paths, which is presented using a tree structure. The complete screenshot is attached in digital appendices.

6 Testing

In this chapter I describe how I tested Visdan, the developed application. The main purpose of the testing is to find out whether Visdan fulfills stated goals and performs the tasks as expected:

- visualize the connected network
- measure link utilization
- allow traffic path definition

Scenarios described in following sections were tested with this background:

- Client
  - Ubuntu 14.04 desktop
  - Python 3.4.3
- Server
  - Ubuntu 14.04 server
  - HPE VAN SDN Controller 2.7.10
  - Mininet 2.2.1
  - Open vSwitch 2.0.2

6.1 Virtual network infrastructure

A network is necessary to test the functionality of Visdan. I used Mininet to create a virtual network for this purpose. The topology is depicted in Figure 25 – it is a standard binary tree topology. It does not contain any loops so it could be used without the Spanning Tree Protocol running on the switches. Nevertheless, it can be well used to test all the required functionality.

Two ways for creating this topology in Mininet are described in Appendix G.

6.2 Scenario 1: network visualization

For this functionality, it is required that all devices present in the network are visualized on the canvas in Visdan. While switches are directly connected to the controller and managed by it, end hosts have to be discovered based on their activity. By default, Mininet hosts do not generate any activity, so for example a `pingall` command has to be executed to make the hosts send ARP requests and the controller register the hosts. Node discovery is a responsibility of the controller, so the controller should have a module that equips switches with flow entries to handle ARP packets.

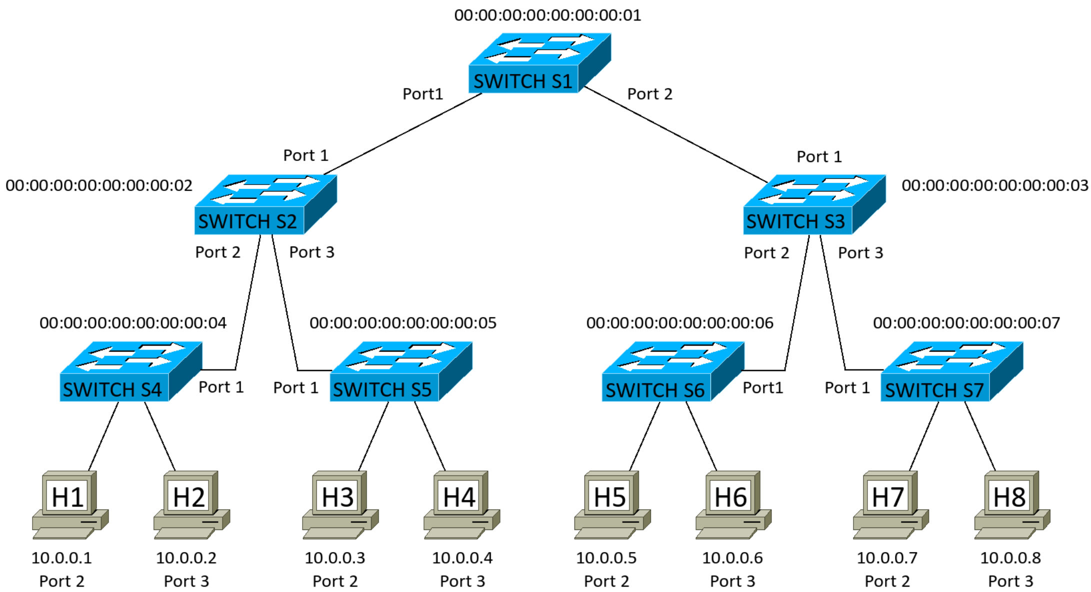


Figure 25: The topology of the virtual testing network. It is a standard binary tree topology comprised of 7 switches (1 core switch S1, 2 distribution switches S2–S3, and 4 access switches S4–S7) and 8 end hosts (H1–H8). There is a single link connecting each pair of devices.

Once the network is alive and connected to the controller with all end hosts discovered, the application is started and connected to the controller. A random graph as captured in Figure 26 is presented. All network devices defined in the topology are present. When the switch S1 is designated as core and switches S2 and S3 are designated as distribution, their icons change and it is possible to let Visdan transform the visualized network into a tree structure. The transformed visualization is captured in Figure 27. It corresponds with the defined topology of the connected network.

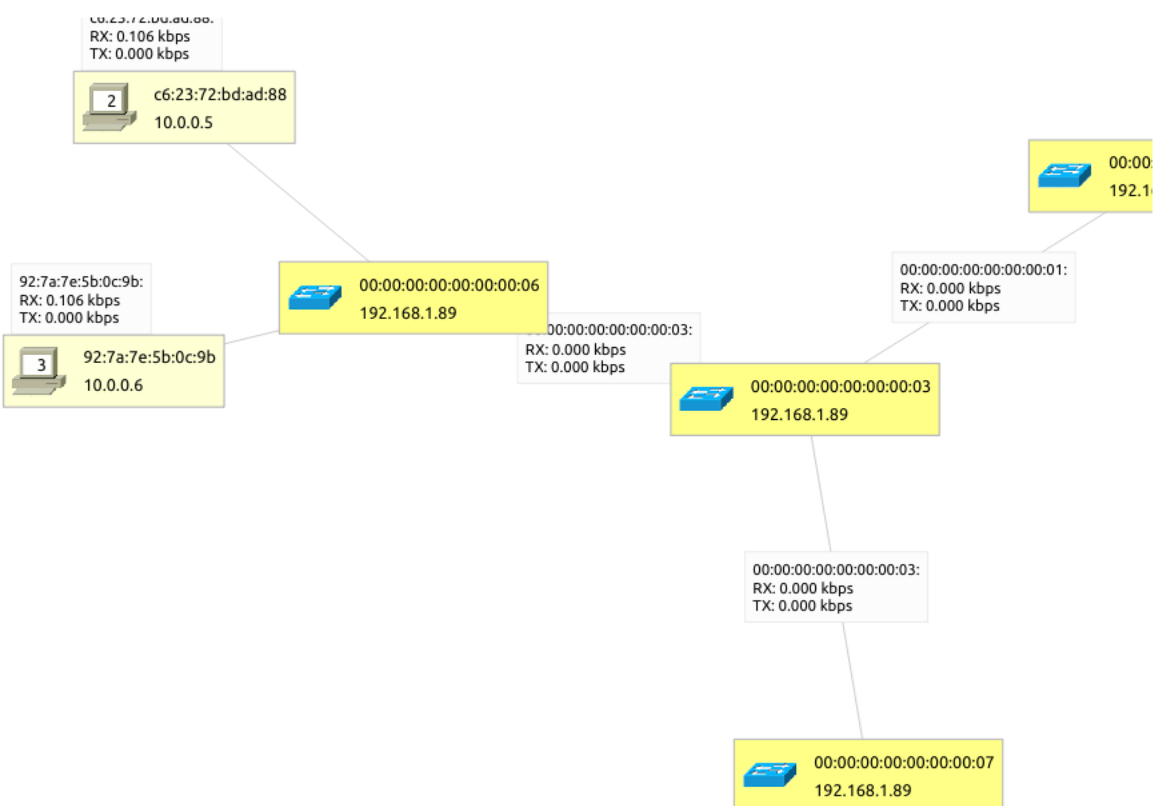


Figure 26: The connected network visualized as a random graph. The complete screenshot is attached in digital appendices.

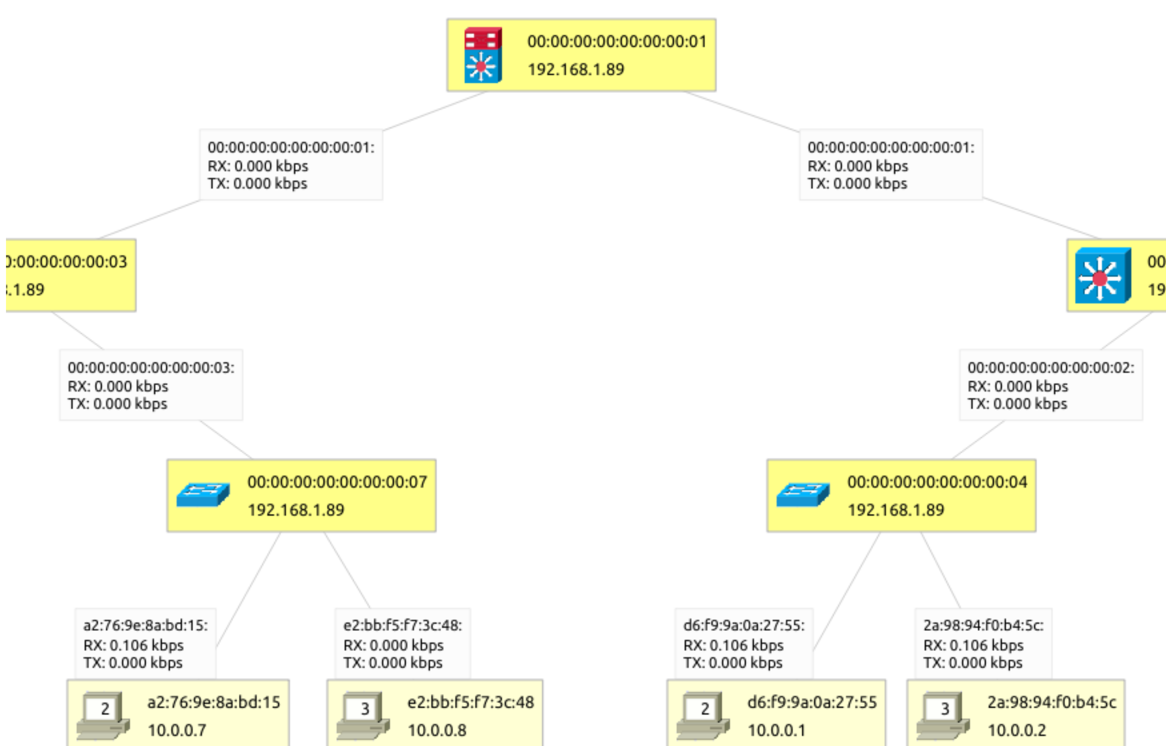


Figure 27: The connected network with defined hierarchy levels, visualized as a binary tree. The complete screenshot is attached in digital appendices.

6.3 Scenario 2: link utilization

To measure the utilization of links, it is necessary to generate traffic between hosts that should be of a known rate and therefore comparable with the values presented by Visdan. For this purpose, the following command was executed in Mininet:

```
h6 ping -s 10000 h2
```

It means that every second a packet of 10 000 bytes should be sent from host H6 (IP address 10.0.0.6) to host H2 (IP address 10.0.0.2) and then back as as a reply. This equals to the rate of 80 000 bits per second (bps) or 80 kilobits per second (kbps).

The network with the ping command running is captured in Figure 28. The flow rate of data transmitted from hosts (TX) and between switches was 81.968 kbps. The flow rate of data received at hosts (RX) is 82.074 kbps. Considering the overhead and inaccuracy caused by variable delays in data transfer, the utilization of links is measured correctly.

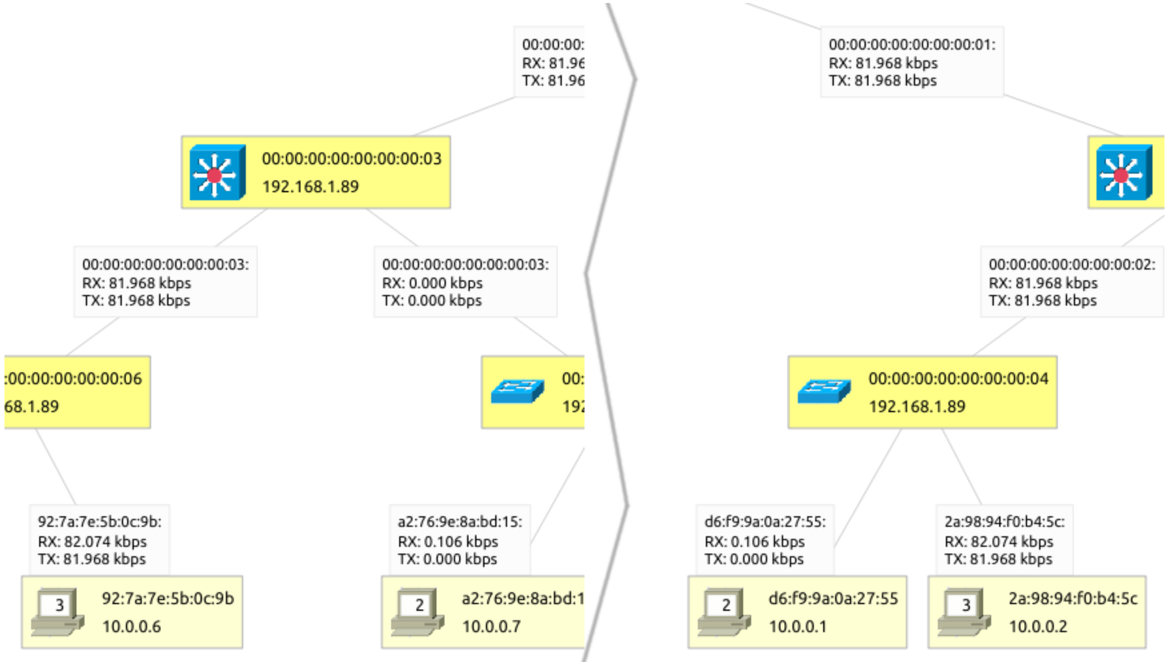


Figure 28: Utilization of the links when a ping command is running between two end hosts. Host H6 is captured in the left part, host H2 is captured in the right part. The complete screenshot is attached in digital appendices.

6.4 Scenario 3: path definition via OpenFlow v. 1.0

The testing network topology provides only one way to each switch and the employed virtual switches are capable of working in the hybrid mode, i.e. capable of forwarding packets in the traditional manner. In this situation, packets can be forwarded between any two hosts because of the flow entries installed in the switches by default by the controller. But the path definition functionality can still be tested. The default flow entry that ensures normal (legacy) packet processing has priority 0, so if a specific path was defined with a higher priority, matched packets would be processed by its created flow entries.

For this scenario, a path from host H5 to host H8 was defined as described in Table 5 as well as the same path the other way, that is from host H8 to host H5. All switches in the network were communicating with the controller through OpenFlow v. 1.0.

Table 5: Parameters of the defined testing path for Scenario 3.

Source end host:	H5 (IP address 10.0.0.5)
Destination end host:	H8 (IP address 10.0.0.8)
Priority:	20 000
Idle timeout:	0
Hard timeout:	1 800
Involved switches:	S3 (DPID 00:00:00:00:00:00:00:03)
	S6 (DPID 00:00:00:00:00:00:00:06)
	S7 (DPID 00:00:00:00:00:00:00:07)

To generate traffic on this path, the following command was executed in Mininet:

```
h5 ping -s 1024 -c 50 h8
```

This `ping` sends 50 packets in each direction with the total size of 51 200 bytes plus the overhead. In Figure 29 and Figure 30, the affected flow entries of switch S3 are captured as presented in the graphical administration of the HPE VAN SDN Controller in section OpenFlow Monitor. Figure 29 captures the state before the `ping` was executed, and Figure 30 captures the state after its execution. The amount of transmitted packets did not change for the flow entry with priority 0 and output action NORMAL, which provides the legacy forwarding in case there are no matching flow entries. The counters of the two flow entries of the defined paths correspond to the number of packets generated by the ping command and their total expected size.

20000	0	0	in_port: 2 eth_type: ipv4 ipv4_src: 10.0.0.5, mask: 255.255.255.255 ipv4_dst: 10.0.0.8, mask: 255.255.255.255	output: 3
20000	0	0	in_port: 3 eth_type: ipv4 ipv4_src: 10.0.0.8, mask: 255.255.255.255 ipv4_dst: 10.0.0.5, mask: 255.255.255.255	output: 2
0	1689	2170566		output: NORMAL

Figure 29: An extract from the flow table on switch S3 before executing the `ping` command. The columns are following: priority, transmitted packets, transmitted bytes, match fields, and action. Taken from OpenFlow Monitor from the graphical administration of the HPE VAN SDN Controller. The complete screenshot is attached in digital appendices.

20000	50	53300	in_port: 2 eth_type: ipv4 ipv4_src: 10.0.0.5, mask: 255.255.255.255 ipv4_dst: 10.0.0.8, mask: 255.255.255.255	output: 3
20000	50	53300	in_port: 3 eth_type: ipv4 ipv4_src: 10.0.0.8, mask: 255.255.255.255 ipv4_dst: 10.0.0.5, mask: 255.255.255.255	output: 2
0	1689	2170566		output: NORMAL

Figure 30: An extract from the flow table on switch S3 after executing the `ping` command. The columns are following: priority, transmitted packets, transmitted bytes, match fields, and action. Taken from OpenFlow Monitor from the graphical administration of the HPE VAN SDN Controller. The complete screenshot is attached in digital appendices.

6.5 Scenario 4: path definition via OpenFlow v. 1.3

This scenario is similar to Scenario 3. The main difference is that all switches in the network were communicating with the controller through OpenFlow v. 1.3. In this version of OpenFlow, the message to define a flow entry in a switch has a different structure compared to OpenFlow v. 1.0.

A path from host H1 to host H7 was defined as described in Table 6 as well as the same path the other way, that is from host H7 to host H1.

To generate traffic on this path, the following command was executed in Mininet:

```
h7 ping -s 1024 -c 10 h1
```

This `ping` sends 10 packets in each direction with the total size of 10 240 bytes plus the overhead. In Figure 31 and Figure 32, the affected flow entries of switch S1 are captured. Figure 31 captures the state before the ping was executed, and Figure 32 captures the state after its execution. The amount of transmitted packets did not change for the flow entry with priority 0 and output action NORMAL, which provides the legacy forwarding in case there are no matching flow entries. The

counters of the two flow entries of the defined paths correspond to the number of packets generated by the ping command and their total expected size.

Table 6: Parameters of the defined testing path for Scenario 4.

Source end host:	H1 (IP address 10.0.0.1)
Destination end host:	H7 (IP address 10.0.0.7)
Priority:	12 000
Idle timeout:	0
Hard timeout:	800
Involved switches:	S1 (DPID 00:00:00:00:00:00:00:01)
	S2 (DPID 00:00:00:00:00:00:00:02)
	S3 (DPID 00:00:00:00:00:00:00:03)
	S4 (DPID 00:00:00:00:00:00:00:04)
	S7 (DPID 00:00:00:00:00:00:00:07)

12000	0	0	in_port: 2 eth_type: ipv4 ipv4_src: 10.0.0.7 ipv4_dst: 10.0.0.1	apply_actions: output: 1
12000	0	0	in_port: 1 eth_type: ipv4 ipv4_src: 10.0.0.1 ipv4_dst: 10.0.0.7	apply_actions: output: 2
0	202	17240		apply_actions: output: NORMAL

Figure 31: An extract from the flow table on switch S1 before executing the ping command. The columns are following: priority, transmitted packets, transmitted bytes, match fields, and instructions with actions. Taken from OpenFlow Monitor from the graphical administration of the HPE VAN SDN Controller. The complete screenshot is attached in digital appendices.

12000	10	10660	in_port: 2 eth_type: ipv4 ipv4_src: 10.0.0.7 ipv4_dst: 10.0.0.1	apply_actions: output: 1
12000	10	10660	in_port: 1 eth_type: ipv4 ipv4_src: 10.0.0.1 ipv4_dst: 10.0.0.7	apply_actions: output: 2
0	202	17240		apply_actions: output: NORMAL

Figure 32: An extract from the flow table on switch S1 after executing the ping command. The columns are following: priority, transmitted packets, transmitted bytes, match fields, and instructions with actions. Taken from OpenFlow Monitor from the graphical administration of the HPE VAN SDN Controller. The complete screenshot is attached in digital appendices.

7 Evaluation

Visdan is a software application that is to facilitate a certain area of network management. It is meant to actively watch the network and present its state to the user. At this point, when the application has been developed and tested, I conclude that the chosen application model does not fit the needs very well.

Visdan is developed as a proactive application that queries the controller, but to which the controller does not send anything on its own. While it is possible to periodically download the complete state of the network as it is done in Visdan, it not only creates extra traffic, but also brings synchronization problems. It may happen that a network device disconnects in the middle of the updating process, which causes trouble. In case the application was built as reactive with event listeners registered in the controller, there would be no synchronization problems because the application would react directly to received event notifications.

Also, when it comes to the management of user-defined traffic paths, it is not easily possible to keep track of defined flow entries and their state. Using the REST API of the HPE VAN SDN Controller, it is only possible to download all configured flow entries of a particular switch, but there is no way to inquire the controller for the state of one specific flow entry. The processing of the great amount of downloaded flow entries from several switches might consume network and computing resources to a considerable extent, and therefore does not seem to be a fitting solution. Rather it would again be reasonable to listen to events about flow entries and react upon them.

Concerning the network visualization, the employed graph positioning algorithm provides a decent service, but it usually still requires the user to customize the visualization. It is probably rather meant for structures where the visualization does not have to be perfect (nodes can overlap and such), but has to provide an information as a whole. An appropriate positioning algorithm should be specifically tailored for this application and provide a complete and usable visualization of the network without any user inputs.

As for the implementation itself, Visdan utilizes a hp-sdn-client library, which makes many things easier. But it does not seem to be under development anymore unlike the REST API of the HPE VAN SDN Controller, which undergoes changes as the controller evolves. It therefore does not appear as something to rely on in applications using one of the latest versions of the controller. Also, when the support for a newer version of OpenFlow is integrated into the controller, there may be new features the library would not cover at all.

But despite the fact that the way Visdan is created may not be the best, it is not just an experimental application developed to test some features. It does function and is therefore a tool that might be well used when experimenting with software-defined networks, even though only through the HPE VAN SDN Controller at this point.

To be used in a production environment, its functionality would likely have to be improved and extended to cover more areas of network management.

8 Conclusion

The primary goal of this work was to develop an application that would provide a readable graphical visualization of a connected OpenFlow-based software-defined network, show the utilization of links between devices, and allow the user to conveniently define a custom traffic path between two end nodes in the network using a user-friendly graphical interface. Based on further analyzed user requirements, such application, called Visdan, was developed.

Visdan is able to visualize an OpenFlow-based network managed by the HPE VAN SDN Controller. There is a random visualization, but if the user provides information about hierarchy levels in the network, the visualization may be transformed into a tree structure respecting the network hierarchy. Elementary information about each network devices, such as unique identifier or IP address, is also presented.

On every link, that is between every two devices, the utilization of the link is displayed. It is presented as the rate of received and transmitted kilobits per second for one device of the link. These statistics are regularly updated together with the whole network visualization.

As for the path definition functionality, it allows the user to choose the path by clicking on desired network devices in the visualization. Once the path is complete, the user can set certain parameters. Visdan then automatically distributes adequate flow entries over involved switches.

While not a primary goal, also important were the literature review and the technological overview. This is because of the cooperation with the networking work group of Department of Informatics of Faculty of Business and Economics at Mendel University in Brno, whom this thesis should help to get acquainted with software-defined networking. Therefore these chapters have been worked out extensively.

This thesis was meant as the first swallow and its outcomes might beneficially serve someone in his or her learning of and experimenting with software-defined networking. The developed application might serve as an inspiration for similar projects because a software tool of this kind has a potential to find its place among network administrators.

9 Bibliography

Sources mentioned only in chapter *Reviews* are listed below in section *Literature review*. Sources from which actual information was cited are listed in section *Cited sources*.

9.1 Literature review

ANTOLÍK, Dávid. *A Network Control Language for OpenFlow Networks.* Brno, 2013. Bachelor thesis. Brno University of Technology, Faculty of Information Technology, Department of Information Systems. Available at: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=118755.

AZODOLMOLKY, Siamak. *Software Defined Networking with OpenFlow: Get hands-on with the platforms and development tools used to build OpenFlow network applications.* Birmingham, UK: Packt Publishing, 2013. ISBN 978-1-84969-872-6.

Brocade Communications Systems. Brocade Flow Manager *Brocade* [online]. Brocade Communications Systems, © 2016a [viewed 2016-11-15]. Available at: http://www.brocade.com/en/products-services/software-networking/sdn-controllers-applications/flow-manager.html.

CODECADEMY. Python *Codecademy* [online]. Codecademy, © 2016 [viewed 2016-11-15]. Available at: https://www.codecademy.com/learn/python.

CONNOLY, James J. *SDN: Defining a Strategic, Business-Focussed Architecture.* Edited by Sonja RUILE. North Charleston: CreateSpace, 2015. ISBN 978-1-5085-4283-4.

DOHERTY, Jim. *SDN and NFV Simplified: A Visual Guide to Understanding Software Defined Networks and Network Function Virtualization.* Upper Saddle River, NJ: Pearson Education, Inc., 2016. ISBN 978-0-13-430640-7.

HERBERT, Thomas F. *SDN, Openflow, and Open vSwitch.* Herndon, VA: Mercury Learning Information, 2014. ISBN 978-1-937585-45-7.

HERINCKX, Tim. *Dynamic and performance driven control for OpenFlow networks.* Ghent, 2013. Master thesis. Ghent University, Faculty of Engineering and Architecture, Department of Information Technology. Available at: http://lib.ugent.be/fulltxt/RUG01/002/033/156/RUG01-002033156_2013_0001_AC.pdf.

HEWLETT PACKARD ENTERPRISE. HPE Network Visualizer: Free Trial. *SDN App Store* [online]. Hewlett Packard Enterprise, © 2016a [viewed 2016-11-15]. Available at: https://marketplace.saas.hpe.com/sdn/content/net-visualizer-trial.

HEWLETT PACKARD ENTERPRISE. SDN App Store. *HPE* [online]. Hewlett Packard Enterprise, © 2016b [viewed 2016-11-15]. Available at: https://marketplace.saas.hpe.com/sdn.

HEWLETT PACKARD ENTERPRISE. HPE Networking Information Library *Hewlett Packard Enterprise* [online]. Hewlett Packard Enterprise, © 2016c [viewed 2016-11-15]. Available at: http://www.hpe.com/info/sdn/infolib.

HU, Fei. *Network Innovation through OpenFlow and SDN: Principles and Design.* Boca Raton, FL: Taylor Francis, 2014. ISBN 978-1-4665-7209-6.

HUANG, Tim. *Path Computation Enhancement in SDN Networks.* Ontario, 2015. Master thesis. Ryerson University. Available at: http://digital.library.ryerson.ca/islandora/object/RULA%3A4465/datastream/OBJ/download/Path_computation_enhancement_in_SDN_networks.pdf.

HYPERGLANCE. *Hyperglance: Visual IT Simplicity via Interactive 3D Topology* [online]. Hyperglance, © 2016 [viewed 2016-11-15]. Available at: https://www.hyperglance.com/.

INFINITE COMPUTER SOLUTIONS. Flow Manager. *GitHub* [online]. Latest commit 2016-08-25 [viewed 2016-11-15]. Available at: https://github.com/InfiniteCS/flowmanager.

ISOLANI, Pedro Heleno. *Interactive Monitoring, Visualization, and Configuration of OpenFlow-Based SDN.* Porto Alegre, 2015. Master thesis. Federal University of Rio Grande do Sul, Institute of Informatics. Available at: http://www.lume.ufrgs.br/bitstream/handle/10183/127452/000974184.pdf.

ISOLANI, Pedro Heleno and Juliano Araujo WICKBOLDT. SDN Interactive Manager. *GitHub* [online]. Latest commit 2015-11-26 [viewed 2016-11-15]. Available at: https://github.com/ComputerNetworks-UFRGS/AuroraSDN.

LARA, Adrian. *Using Software-Defined Networking to Improve Campus, Transport and Future Internet Architectures.* Lincoln, 2015. Dissertation. University of Nebraska-Lincoln, The Graduate College. Available at: http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1109&context=computerscidiss.

LEITNER, Matej. *Review of Available Tools for Control Plane of Software Defined Networks* . Brno, 2015. Bachelor thesis. Masaryk University, Faculty of Informatics. Available at: http://is.muni.cz/th/396543/fi_b/bc.pdf.

MARCINIAK, Petr. *Load Balancing in OpenFlow Networks.* Brno, 2013. Master thesis. Brno University of Technology, Faculty of Information Technology,

Department of Information Systems. Available at:
`https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=118981`.

MARIST SDN LAB. Avior 2.0. *GitHub* [online]. Latest commit 2016-04-18 [viewed 2016-11-15]. Available at:
`https://github.com/1PhoenixM/avior-service`.

MARSCHKE, Doug, Jeff DOYLE and Pete MOYER. *Software Defined Networking (SDN): Anatomy of OpenFlow.* Raleigh, NC: Lulu, 2015. ISBN 978-1-4834-2723-2.

MININET. Documentation – mininet/mininet Wiki. *GitHub* [online]. Last reviewed 2016-08-20. Available at:
`https://github.com/mininet/mininet/wiki/Documentation`.

MORREALE, Patricia A. and James M. ANDERSON. *Software Defined Networking: Design and Deployment.* Boca Raton, FL: CRC Press, 2014. ISBN 978-1-4822-3863-1.

Ministerstvo školství, mládeže a tělovýchovy. Přehled vysokých škol v ČR. *MŠMT ČR* [online]. Ministerstvo školství, mládeže a tělovýchovy, © 2013–2016 [viewed 2016-11-09]. Available at:
`http://www.msmt.cz/vzdelavani/vysoke-skolstvi/prehled-vysokych-skol-v-cr-3`.

NADEAU, Thomas D. and Kenneth GRAY. *SDN: Software Defined Networks.* Sebastopol, CA: O'Reilly Media, 2013. ISBN 978-1-4493-4230-2.

NAVARRO, Martí Boada. *Dynamic Load Balancing in Software-Defined Networks.* Aalborg, 2014. Master thesis. Aalborg University, Department of Electronic Systems. Available at:
`http://projekter.aau.dk/projekter/files/198529981/Marti_Boada_Master_Thesis.pdf`.

OPEN NETWORKING FOUNDATION. Technical Library. *Open Networking Foundation* [online]. Open Networking Foundation, © 2016a [viewed 2016-11-15]. Available at:
`https://www.opennetworking.org/sdn-resources/technical-library`.

PALATINUS, Michal. *Extension of SDN platform available at FIIT STU.* Bratislava, 2015. Bachelor thesis. Slovak University of Technology, Faculty of Informatics and Information Technologies. Available at:
`http://www.crzp.sk/crzpopacxe/openURL?crzpID=53621&crzpSigla=stubratislava`.

PHINJIRAPONG, Pattanapoom. *CAFFEINE: Congestion Avoidance For Fairness & Efficiency In Network Entities.* State College, 2015. Master thesis. The

Pennsylvania State University, The Graduate School, School of Science, Engineering, and Technology. Available at: https://etda.libraries.psu.edu/files/final_submissions/11208.

PILGRIM, Mark. *Dive Into Python 3* [online]. © 2009 [viewed 2016-11-15]. Available at: http://www.diveintopython3.net.

PYTHON SOFTWARE FOUNDATION. Overview. *Python 3.5.2 documentation* [online]. Python Software Foundation, © 2001–2016 [viewed 2016-11-15]. Available at: https://docs.python.org/3/.

RIVERBANK COMPUTING PyQt5.7 Reference Guide. *PyQt5 Reference Guide* [online]. Riverbank Computing, © 2015 [viewed 2016-11-15]. Available at: http://pyqt.sourceforge.net/Docs/PyQt5.

THE QT COMPANY Qt 5.7 *Qt Documentation* [online]. The Qt Company, © 2016a [viewed 2016-11-15]. Available at: http://doc.qt.io/qt-5.

THE QT COMPANY Widget-based User Interfaces *Qt Documentation* [online]. The Qt Company, © 2016b [viewed 2016-11-15]. Available at: http://doc.qt.io/qt-5/topics-ui.html#widget-based-user-interfaces.

RAMACHANDRA, Prerna. *ViewNet: A Visualization Tool for Software Defined Networks.* Princeton, 2014. Undergraduate senior thesis. Princeton University, Department of Computer Science. Available at: http://arks.princeton.edu/ark:/88435/dsp01k643b1344.

REITZ, Kenneth. *PEP 8: the Style Guide for Python Code* [online]. [viewed 2016-11-15]. Available at: http://pep8.org.

SEMEDO, Gonçalo Miguel Alves. *Load Balancing in Real Software Defined Networks.* Lisbon, 2014. Master thesis. University of Lisbon, Faculty of Sciences, Department of Informatics. Available at: http://www.di.fc.ul.pt/ñuno/THESIS/GoncaloSemedo_master14.pdf.

SCHMIDT, Andreas. *Interactive Visualization of Software Defined Networks.* Saarbruecken, 2013. Bachelor thesis. Saarland University, Faculty of Natural Sciences and Technology I, Department of Computer Science. Available at: https://www.on.uni-saarland.de/publications/IVOSDN_Andreas_Schmidt.pdf.

SCHMIDT, A., P. S. TENNIGKEIT, and M. KARL. SDN-Visualization. *GitHub* [online]. Latest release 2015-08-18 [viewed 2016-11-15]. Available at: https://github.com/UdS-TelecommunicationsLab/SDN-Visualization.

SDNCENTRAL. *SDxCentral* [online]. SDNCentral, © 2016 [viewed 2016-11-15]. Available at: https://www.sdxcentral.com.

SHUKLA, Vishal. *Introduction to Software Defined Networking - OpenFlow & VxLAN.* North Charleston: CreateSpace, 2013. ISBN 978-1-48267-813-0.

SMILER. S, Kingston. *OpenFlow Cookbook: over 110 recipes to design and develop your own OpenFlow switch and OpenFlow controller.* Birmigham, UK: Packt publishing, 2015. ISBN 978-1-78398-794-8.

STACK EXCHANGE. *Stack Overflow* [online]. Stack Exchange, © 2016 [viewed 2016-11-15]. Available at: https://www.stackoverflow.com.

STALLINGS, William. *Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud.* Indianapolis, IN: Addison-Wesley Professional, 2015. ISBN 978-0-13-417539-3.

SUNDARARAJAN, Rajesh K. *Software Defined Networking (SDN) - a definitive guide.* Amazon Digital Services LLC, 2013. Kindle Edition. ASIN B00D5V02E0.

TECHTARGET. Global Network of Information Technology Websites and Contributors *TechTarget* [online]. TechTarget, © 2016 [viewed 2016-11-15]. Available at: http://www.techtarget.com/network.

TIWARI, Vivek. *SDN and OpenFlow for beginners with hands on labs.* Amazon Digital Services LLC, 2013. Kindle Edition. ASIN B00EZE46D4.

TUCKER, Dave. HP SDN Client 1.1.1 documentation. *HP SDN Client* [online]. Hewlett-Packard Development Company, © 2014 [viewed 2016-11-15]. Available at: http://hp-sdn-client.readthedocs.io.

WALLASCHEK, Felix. JSFlowViz – A simple OpenFlow Visualization for Beacon. *GitHub* [online]. Latest commit 2014-12-10 [viewed 2016-11-15]. Available at: https://github.com/wallaschek/JSFlowViz.

9.2 Cited sources

1. Installation. *Python GTK+ 3 Tutorial 3.4 documentation* [online]. [viewed 2016-11-20]. Available at: http://python-gtk-3-tutorial.readthedocs.io/en/latest/install.html.

BORINI, Stefano. Introduction. *Understanding Model-View-Controller* [online]. [viewed 2016-12-05]. Available at: https://www.gitbook.com/book/stefanoborini/modelviewcontroller/details.

CISCO. Cisco Express Forwarding Overview *Cisco* [online]. Cisco, 2014 [viewed 2016-11-18]. Available at:

http://www.cisco.com/c/en/us/td/docs/ios/12_2/switch/configuration/guide/fswtch_c/xcfcef.html.

Clean Slate Design for the Internet [online]. © 2016 [viewed 2016-08-05]. Available at: http://cleanslate.stanford.edu.

FREDRICH, Todd. *RESTful Service Best Practices: Recommendations for Creating Web Services.* [online]. August 2, 2013 [viewed 2016-11-20]. Available at: https://github.com/tfredrich/RestApiTutorial.com/raw/master/media/RESTful%20Best%20Practices-v1_2.pdf.

GIJARE, Nandan. What is the difference between a router and a Layer 3 switch? *SearchNetworking* [online]. TechTarget, 2004 [viewed 2016-08-05]. Available at: http://searchnetworking.techtarget.com/answer/What-is-the-difference-between-a-router-and-a-Layer-3-switch.

GNOME. Getting GNOME. *GNOME* [online]. The GNOME Project, © 2005–2016 [viewed 2016-11-20]. Available at: https://www.gnome.org/getting-gnome.

GÖRANSSON, Paul and Chuck BLACK. *Software Defined Networks: A Comprehensive Approach.* Amsterdam: Morgan Kaufmann, 2014. ISBN 978-0-12-416675-2.

GUI Programming in Python. *Python Wiki* [online]. Last reviewed 2016-11-05 [viewed 2016-08-05]. Available at: https://wiki.python.org/moin/GuiProgramming.

HEWLETT PACKARD ENTERPRISE. *HPE VAN SDN Controller Software* [online]. Revision 3. Hewlett Packard Enterprise, March 2016a. [viewed 2016-11-20]. Available at: http://h20195.www2.hpe.com/v2/getpdf.aspx/4AA4-9827ENW.pdf.

HEWLETT PACKARD ENTERPRISE. *HPE VAN SDN Controller and Applications Support Matrix* [online]. Hewlett Packard Enterprise, September 2016b. [viewed 2016-11-20]. Available at: http://h20566.www2.hpe.com/hpsc/doc/public/display?docId=c05040231.

HEWLETT PACKARD ENTERPRISE. *HPE VAN SDN Controller 2.7 Installation Guide* [online]. Edition 2. Hewlett Packard Enterprise, March 2016c. [viewed 2016-11-20]. Available at: http://h20566.www2.hpe.com/hpsc/doc/public/display?docId=c05028139.

HEWLETT PACKARD ENTERPRISE. *HPE VAN SDN Controller 2.7 Administrator Guide* [online]. Hewlett Packard Enterprise, March 2016d. [viewed 2016-11-20]. Available at: http://h20566.www2.hpe.com/hpsc/doc/public/display?docId=c05028095.

HEWLETT PACKARD ENTERPRISE. *HPE VAN SDN Controller 2.7 REST API Reference* [online]. Hewlett Packard Enterprise, March 2016e. [viewed 2016-11-20]. Available at: http://h20566.www2.hpe.com/hpsc/doc/public/display?docId=c05040230.

How to install wxPython. *wxPyWiki* [online]. Last reviewed 2015-08-11 [viewed 2016-11-20]. Available at: https://wiki.wxpython.org/How%20to%20install%20wxPython.

KERNER, Sean Michael. OpenFlow Protocol 1.3.0 Approved *Enterprise Networking Planet* [online]. QuinStreet, May 17, 2012 [viewed 2016-10-12]. Available at: http://www.enterprisenetworkingplanet.com/nethub/openflow-protocol-1.3.0-approved.html.

KIM, Hyojoon and Nick FEAMSTER. Improving Network Management with Software Defined Networking. *IEEE Communications Magazine* [online]. 2013, vol. 51, no. 2, p. 114-119 [viewed 2016-10-12]. DOI 10.1109/MCOM.2013.6461195. ISSN 0163-6804. Available at: http://ieeexplore.ieee.org/document/6461195/.

KIVY. *Kivy: Cross-platform Python Framework for NUI Development* [online]. [viewed 2016a-11-20]. Available at: https://kivy.org.

KIVY. FAQ. *Kivy 1.9.2-dev0 documentation* [online]. [viewed 2016b-11-20]. Available at: https://kivy.org/docs/faq.html.

KOZIEROK, Charles M. *The TCP/IP Guide* [online]. © 2001-2005a [viewed 2016-08-05]. Available at: http://www.tcpipguide.com/free.

KOZIEROK, Charles M. The Open System Interconnection (OSI) Reference Model. *The TCP/IP Guide* [online]. © 2001-2005b [viewed 2016-08-05]. Available at: http://www.tcpipguide.com/free/t_TheOpenSystemInterconnectionOSIReferenceModel.htm.

MCCANCE Shaun et al. GNOME application development overview. *GNOME Developer Center* [online]. GNOME Foundation [viewed 2016-11-20]. Available at: https://developer.gnome.org/platform-overview/unstable.

MININET TEAM. Mininet Overview. *Mininet* [online]. Mininet Team, © 2016a [viewed 2016-11-20]. Available at: http://mininet.org/overview.

MININET TEAM. Download/Get Started With Mininet. *Mininet* [online]. Mininet Team, © 2016b [viewed 2016-11-20]. Available at: http://mininet.org/download.

MININET TEAM. Mininet Walkthrough. *Mininet* [online]. Mininet Team, © 2016c [viewed 2016-11-20]. Available at: http://mininet.org/walkthrough.

MININET. mininet/node.py at master mininet/mininet *GitHub* [online]. Lines 34–37, 1541–1554. Last reviewed 2016a-06-04 [viewed 2016-11-20]. Available at: https://github.com/mininet/mininet/blob/master/mininet/node.py.

MININET. Documentation – mininet/mininet Wiki. *GitHub* [online]. Last reviewed 2016b-08-20 [viewed 2016-11-20]. Available at: https://github.com/mininet/mininet/wiki/Documentation.

MININET. mn – create a Mininet network. *Ubuntu Manpage* [online]. [viewed 2016c-11-20]. Available at: http://manpages.ubuntu.com/manpages/xenial/en/man1/mn.1.html.

MICROSOFT. TCP/IP Protocol Architecture. *TechNet* [online]. Microsoft, © 2016 [viewed 2016-08-05]. Available at: https://technet.microsoft.com/en-us/library/cc958821.aspx.

MICROSOFT. The OSI Model's Seven Layers Defined and Functions Explained. *Microsoft Support* [online]. Microsoft, © 2016. Last reviewed 2014-06-13 [viewed 2016-10-12]. Available at: https://support.microsoft.com/en-us/kb/103884.

Differences between Layer 2, 3, 4 Switching / Multilayer Switching / Layer 3 Routing. *networkPCworld* [online]. [viewed 2016-08-05]. Available at: http://www.networkpcworld.com/differences-between-layer-2-3-4-switching-multilayer-switching-layer-3-routing/.

NOX. noxrepo/nox: The NOX Controller. *GitHub* [online]. Last reviewed 2012-05-12 [viewed 2016-11-20]. Available at: https://github.com/noxrepo/nox.

OPENFLOW SWITCH CONSORTIUM. OpenFlow Tutorial. *OpenFlow Wiki* [online]. OpenFlow Switch Consortium, © 2011a [viewed 2016-11-20]. Available at: http://archive.openflow.org/wk/index.php/OpenFlow_Tutorial.

OPENFLOW SWITCH CONSORTIUM. What is OpenFlow? *OpenFlow* [online]. OpenFlow Switch Consortium, © 2011b [viewed 2016-12-05]. Available at: http://archive.openflow.org/wp/learnmore/.

OPEN NETWORKING FOUNDATION. *OpenFlow Switch Specification* [online]. Open Networking Foundation, 2009. Version 1.0.0 [viewed 2016-08-05]. Available at: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf.

OPEN NETWORKING FOUNDATION. *Software-Defined Networking: The New Norm for Networks* [online]. Open Networking Foundation, 2012a [viewed 2016-08-05]. Available at: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

OPEN NETWORKING FOUNDATION. *OpenFlow Switch Specification* [online]. Open Networking Foundation, 2012b. Version 1.3.0 [viewed 2016-10-12]. Available at: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

OPEN NETWORKING FOUNDATION. *Open Networking Foundation* [online]. Open Networking Foundation, © 2016b [viewed 2016-08-05]. Available at: http://www.opennetworking.org.

OPEN NETWORKING FOUNDATION. What is ONF? *Open Networking Foundation* [online]. Open Networking Foundation [viewed 2016c-08-05]. Available at: https://www.opennetworking.org/images/stories/downloads/about/onf-what-why-2016.pdf.

OPEN NETWORKING FOUNDATION. *SDN Architecture* [online]. Open Networking Foundation, 2016d. Issue 1.1 [viewed 2016-08-05]. Available at: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR-521_SDN_Architecture_issue_1.1.pdf.

OPEN NETWORKING FOUNDATION. OpenFlow. *Open Networking Foundation* [online]. Open Networking Foundation, © 2016e [viewed 2016-08-05]. Available at: https://www.opennetworking.org/sdn-resources/openflow.

ORACLE CORPORATION. Chapter 1. First steps. *Oracle VM VirtualBox ®: User Manual* [online]. Oracle Corporation, © 2004–2016 [viewed 2016-08-05]. Available at: http://www.virtualbox.org/manual/ch01.html.

OPEN VSWITCH. ovs-testcontroller – simple OpenFLow controller for testing. *Ubuntu Manpage* [online]. [viewed 2016-11-20]. Available at: http://manpages.ubuntu.com/manpages/xenial/en/man8/ovs-testcontroller.8.html.

POKORNÝ, Martin and Petr ZACH. First Educational Steps in SDN Application Development. *Acta Universitatis Agriculturae et Silviculturae Mendelianae Brunensis* [online]. 2015, vol. 63, no. 6, p. 2093–2099 [viewed 2016-08-05]. DOI 10.11118/actaun201563062093. ISSN 1211-8516. Available at: http://acta.mendelu.cz/63/6/2093.

ProjectPhoenix. *wxPyWiki* [online]. Last reviewed 2015-01-15 [viewed 2016-11-20]. Available at: https://wiki.wxpython.org/ProjectPhoenix.

Projects/PyGObject. *GNOME Wiki!* [online]. The GNOME Project, © 2005–2015. Last reviewed 2016-02-23 [viewed 2016-11-20]. Available at: https://wiki.gnome.org/Projects/PyGObject.

PySide. *Qt Wiki* [online]. Last reviewed 2016-06-05 [viewed 2016-11-20]. Available at: https://wiki.qt.io/PySide.

PySide FAQ. *Qt Wiki* [online]. Last reviewed 2016-06-05 [viewed 2016-11-20]. Available at: https://wiki.qt.io/PySide_FAQ.

PYTHON SOFTWARE FOUNDATION. 25.1. tkinter – Python interface to Tcl/Tk. *Python 3.6.0b4 documentation* [online]. Python Software Foundation, © 2001–2016. Last reviewed 2016a-11-01 [viewed 2016-11-20]. Available at: https://docs.python.org/3.6/library/tkinter.html.

PYTHON SOFTWARE FOUNDATION. What is Python? Executive Summary. *Python.org* [online]. Python Software Foundation, © 2001–2016b [viewed 2016-08-05]. Available at: https://www.python.org/doc/essays/blurb/.

RIVERBANK. What is PyQT? *Riverbank* [online]. Riverbank Computing, © 2016 [viewed 2016-11-20]. Available at: https://www.riverbankcomputing.com/software/pyqt/intro.

ROSEMAN, Mark. Tk Tutorial. *TkDocs* [online]. Mark Roseman, © 2007–2015 [viewed 2016-11-20]. Available at: http://www.tkdocs.com/tutorial/.

ROUSE, Margaret. What is network management system? *WhatIs.com* [online]. TechTarget, 2013 [viewed 2016-10-12]. Available at: http://whatis.techtarget.com/definition/network-management-system.

NIPPON TELEGRAPH and TELEPHONE CORPORATION. Getting Started. *Ryu 4.8 documentation* [online]. Nippon Telegraph and Telephone Corporation, © 2011–2014 [viewed 2016-11-20]. Available at: https://ryu.readthedocs.io/en/latest/getting_started.html#what-s-ryu.

SHAW, Andy. Qt 4.8.x Support to be Extended for Another Year. *Qt Blog* [online]. The Qt Company, November 27, 2014 [viewed 2016-11-20]. Available at: http://blog.qt.io/blog/2014/11/27/qt-4-8-x-support-to-be-extended-for-another-year/.

SHENKER, Scott. *The Future of Networking, and the Past of Protocols.* [presentation]. Stanford University: Open Networking Summit, October 18–19, 2011 [viewed 2016-08-05]. Recording available at: https://youtu.be/YHeyuD89n1Y.

THE GTK+ TEAM. GTK+ Features. *The GTK+ Project* [online]. The GTK+ Team, © 2007–2016 [viewed 2016-11-20]. Available at: https://www.gtk.org/features.php.

THE QT COMPANY. Cross-platform development. *Qt for developers by developers* [online]. The Qt Company, © 2016 [viewed 2016-11-20]. Available at: https://www.qt.io/developers/.

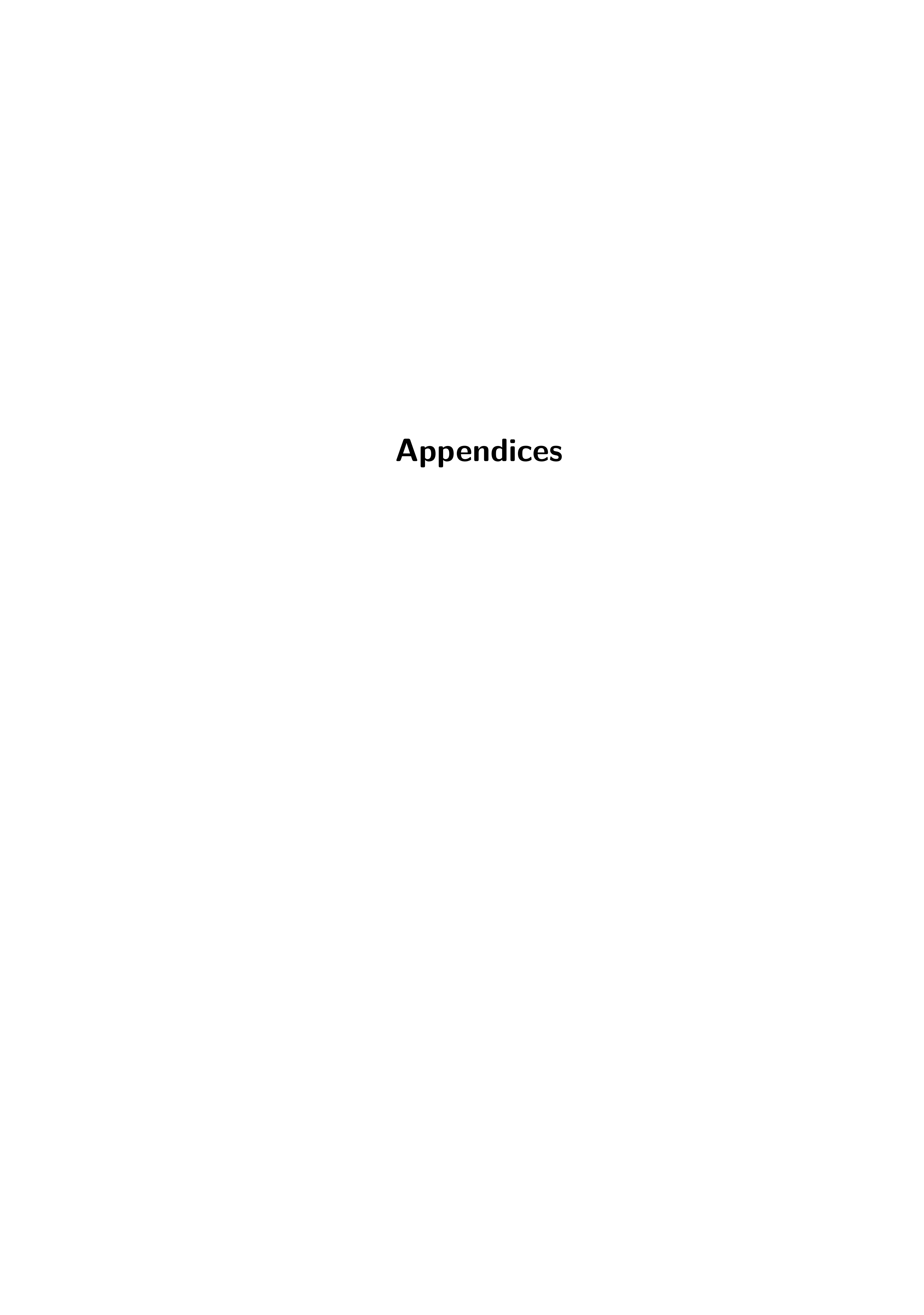
Tkinter. *Python Wiki* [online]. Last reviewed 2014-06-08 [viewed 2016-11-20]. Available at: https://wiki.python.org/moin/TkInter.

TUCKER, Dave. HP SDN Client 1.1.1 documentation. *HP SDN Client* [online]. Hewlett-Packard Development Company, © 2014 [viewed 2016-11-15]. Available at: `http://hp-sdn-client.readthedocs.io`.

VAN ROSSUM, Guido. Comparing Python to Other Languages. *Python.org* [online]. Python Software Foundation, 1997 [viewed 2016-08-05]. Available at: `https://www.python.org/doc/essays/comparisons/`.

WXPYTHON. What is wxPython? *wxPython* [online]. [viewed 2016-11-20]. Available at: `https://www.wxpython.org/what.php`.

WXWIDGETS. Overview. *wxWidgets* [online]. wxWidgets, © 2016 [viewed 2016-11-20]. Available at: `http://wxwidgets.org/about/`.



Appendices

A Digital appendices

On the attached CD, following content may be found:

- Application Visdan - its source code and generated API documentation. To run the application, it is necessary to have following libraries installed together with their dependencies:
  - hpsdnclient
  - networkx
  - PyQt5
- Algorithm flow charts in full resolution
- Screenshots of the graphical user interface in full resolution
- Screenshots from the testing in full resolution

B OpenFlow Specification v. 1.3.0

Through OpenFlow versions 1.1.0, 1.2.0, up to 1.3.0, many features were added or enhanced. Some of the changes are significant, some are hidden in details. In the following sections, I will describe the ones I consider most significant for essential knowledge.

B.1 Extensible match support

In the first versions of OpenFlow, the flow match structure in a flow entry was a fixed list of match fields that could either have a certain value or have the wildcard flag set (Open Networking Foundation, 2009, p. 20–21). Since the range of match fields, some of them even overloaded, was hardcoded in the data structure, the matching logic in switches also had to be tailored to the predefined set of packet header fields. This approach was inflexible and did not allow for easy extension of supported match fields (Open Networking Foundation, 2012b, p. 99; Göransson and Black, 2014, p. 106–107).

Fortunately, the rigid data structure was replaced and match fields are now described using the OpenFlow Extensible Match (OXM) format. OXM is a type-length-value format, which means that a fixed-length header indicates the type of the match field and the length of the value, which is of variable size depending on the type of the match field (Open Networking Foundation, 2012b, p. 39–40).

Table 7: OpenFlow Extensible Match header fields.

<table>
<tr><td colspan="2">oxm_type</td><td rowspan="2">oxm_hasmask</td><td rowspan="2">oxm_length</td></tr>
<tr><td>oxm_class</td><td>oxm_field</td></tr>
</table>

As shown in Table 7, the type is comprised of a class and a field. There are currently two classes – OpenFlow basic class, which encompasses all standard fields, and Experimenter class, which is meant for further experiments. Using this type identification, every logical match field has its own unique type and there is no more a need for overloading. There is also a hasmask flag in the OXM header, which indicates whether the match field uses a bitmask or not. If a mask is used, it follows the match field value and is of the same length (Open Networking Foundation, 2012b, p. 39–41).

The flow match structure used in a flow entry may have zero or more OXM match fields (Open Networking Foundation, 2012b, p. 38). If a match field is not present, it is automatically wildcarded, so a flow entry with no match fields matches every packet (Open Networking Foundation, 2012b, p. 40).

Not only is the matching logic more flexible now and can be implemented in switches without regards to particular match fields, but also the standard list of match fields has greatly expanded. The full list is present in the specification (Open Networking

Foundation, 2012b, p. 44–45), but one of the most important is the support of IPv6 header fields. The extensible matching also brings another advantage – the same fields that can be matched can also be modified when a packet is processed by the flow entry (Open Networking Foundation, 2012b, p. 99; Göransson and Black, 2014, p. 107–108).

B.2 Multiple flow tables

A major advancement in packet processing was introduced by the addition of multiple flow tables. A packet may be matched in one table, applied certain actions, and then sent to next table for further treatment. This allows more sophisticated packet processing because a packet may be modified several times under various conditions in each flow table (Göransson and Black, 2014, p. 99).

Flow entries are now assigned instructions and every packet is associated with an action set, which is empty in the beginning. An instruction may add or remove actions from the action set, or dispatch the packet to one of the following flow tables. The action set is then carried with the packet between the tables. When the pipeline reaches its end, i.e. the packet is in the last table or there is not an instruction to send it to another table, the actions from the action set are executed in an order specified by the protocol (Open Networking Foundation, 2012b, p. 16, 18; Göransson and Black, 2014, p. 99–101).

Except for the packet, its ingress port and action set, there is also a metadata field carried between the tables. Metadata is a 64-bit register that carries arbitrarily defined state and that can be used as a match field. When writing to or matching against the metadata field, a mask must be used to indicate which bits are to be modified (Open Networking Foundation, 2012b, p. 7, 44).

There are following types of instructions, some of them are required to be implemented in a switch, while other are optional (Open Networking Foundation, 2012b, p. 16, 18):

- **Meter** (optional) – direct the packet to a specified meter (meters will be described later in this chapter). The packet may be dropped in the meter and its processing be therefore quit at this point.
- **Apply-Actions** (optional) – immediately apply a list of specific actions without affecting the action set of the packet.
- **Clear-Actions** (optional) – immediately clear the action set of the packet.
- **Write-Actions** (required) – add the specified actions into the action set of the packet. If an action of the same type is already present, it is overwritten.
- **Write-Metadata** (optional) – write the metadata into the metadata field. Affects only the bits specified by the mask.

- **Goto-Table** (required) – point out the next table for the packet processing. The id of the table must be greater than the current id. This instruction cannot be used for flow entries in the last flow table.

A flow entry may only have one instruction of each type in its instruction set and they are executed in the order mentioned in the list above. As well an action set of a packet may include only one action of each type. In case more actions of the same type need to be applied, the Apply-Actions instruction may be used (Open Networking Foundation, 2012b, p. 18).

Actions can be divided into categories listed below, again some are required and some optional for implementation. Each category includes several more specific action types (Open Networking Foundation, 2012b, p. 19–21).

- **Output** (required) – forward the packet to a specified port, be it a physical port, a reserved virtual port, or a switch-defined logical port.
- **Set-Queue** (optional) – set the queue ID for the packet. This action is used in combination with the Output action as it determines the particular queue of the output port of the packet. This queue then influences scheduling and forwarding of the packet.
- **Drop** (required) – packets with no output actions in their action set are dropped.
- **Group** (required) – process the packet through the specified group (groups are described in the next section).
- **Push-Tag/Pop-Tag** (optional) – push or pop a VLAN, MPLS, or PBB tag. The outer-most tag is popped and a new tag is pushed into the outer-most place.
- **Set-Field** (optional) – modify values of specific fields in the packet header. Set-Field actions are identified by the field type, more Set-Field actions may therefore be in an action set, but only one for each field.
- **Change-TTL** (optional) – modify IPv4 Time To Live, IPv6 Hop Limit, or MPLS Time To Live in the packet.

An OpenFlow switch has to accommodate at least one flow table, more flow tables are optional (Open Networking Foundation, 2012b, p. 10–21). While the implementation of this functionality is feasible in software switches, it is challenging to realize the multiple flow tables in hardware (Göransson and Black, 2014, p. 109).

B.3 Multiple flow tables illustration

An example follows to better illustrate the enhanced matching process. Its figures are only extracts of real entities and contain only information necessary for the illustration.

- Table 8 depicts a packet coming into a switch.

- Table 9 represents the first flow table in the pipeline – with id 0 – and one of its flow entries. When a packet matches this flow entry, the flow entry adds a Set-Field action for IPv4 destination address into its action set and modifies its metadata field.
- Table 10 depicts the altered packet, which is then sent to a flow table with id 3 because of the Goto-Table instruction present in the matched flow entry.
- Table 11 represents the flow table with id 3. In case of a matched packet, its flow entry adds a Set-Field action for Ethernet source address into the action set of the packet and then sends the packet out to group with id 17 for further processing and final output.
- Table 12 depicts the packet after passing through the two flow tables. If there were the Goto-Table instruction, the packet would be passed to another flow table in this state. Since this instruction was not present in the last matched flow entry, the action set of the packet is executed.
- Table 13 depicts the packet after the application of all actions of its action set.

Table 8: Example of multiple flow tables – incoming packet.

Header fields	
Ethernet source address	00:01:02:aa:bb:11
Ethernet destination address	00:01:02:ef:ce:1a
IPv4 source address	192.168.131.56
IPv4 destination address	192.168.131.202
TCP source port	50555
TCP destination port	22
Input port	7
Metadata	0x0000000000000000
Action set	
empty	

Table 9: Example of multiple flow tables – flow entry in flow table 0.

Match fields	
IPv4 source address	192.168.131.*
TCP destination port	22
Instructions	
Write-Actions	Set-Field: IPv4 destination address = 192.168.131.1
Write-Metadata	value: 0x0000000000000A00 mask: 0x0000000000000F00
Goto-Table	3

Table 10: Example of multiple flow tables – the packet altered in flow table 0.

Header fields	
Ethernet source address	00:01:02:aa:bb:11
Ethernet destination address	00:01:02:ef:ce:1a
IPv4 source address	192.168.131.56
IPv4 destination address	192.168.131.202
TCP source port	50555
TCP destination port	22
Input port	7
Metadata	0x0000000000000A00
Action set	
Set-Field	IPv4 destination address = 192.168.131.1

Table 11: Example of multiple flow tables – flow table 3.

Match fields	
Input port	7
Metadata	value: 0x0000000000000A00 mask: 0xFFFFF00000000F0F
TCP destination port	22
Instructions	
Write-Actions	Set-Field: Ethernet src. address = 00:01:02:aa:c1:22
Group	17

Table 12: Example of multiple flow tables – the packet altered in flow tables 0 and 3.

Header fields	
Ethernet source address	00:01:02:aa:bb:11
Ethernet destination address	00:01:02:ef:ce:1a
IPv4 source address	192.168.131.56
IPv4 destination address	192.168.131.202
TCP source port	50555
TCP destination port	22
Input port	7
Metadata	0x0000000000000A00
Action set	
Set-Field	IPv4 destination address = 192.168.131.1
Set-Field	Ethernet src. address = 00:01:02:aa:c1:22
Group	17

Table 13: Example of multiple flow tables – the packet with applied actions.

Header fields	
Ethernet source address	00:01:02:aa:c1:22
Ethernet destination address	00:01:02:ef:ce:1a
IPv4 source address	192.168.131.56
IPv4 destination address	192.168.131.1
TCP source port	50555
TCP destination port	22
Sent to group 17 for further processing and output.	

B.4 Groups

Groups provide a framework for configuring symbolic output paths in flow entries. A flow entry may have the output port constantly set to a particular group, while the final output for the packet may vary depending on the setting of the group. A typical use case for groups is multicast since they allow forwarding one packet out multiple interfaces, but there are also other ways to use them as described below.

Groups are defined in a group table using group entries. Each group entry has a unique identifier, a type, counters, and one or more action buckets. An action bucket contains actions to perform on a packet and additional parameters depending on the group type. The last action is an output action (Open Networking Foundation, 2012b, p. 14; Göransson and Black, 2014, p. 101–102).

There are four group types, some of them must be implemented in a switch, some of them do not have to (Open Networking Foundation, 2012b, p. 14–15):

- **All** (required) – when a packet is forwarded to this group, it is cloned for every action bucket and all buckets are executed. This group type provides multicast/broadcast functionality.
- **Select** (optional) – only one bucket in the group is executed for each packet. The selection of the bucket is based on a selection algorithm, but the implementation of the algorithm is switch-specific and not a part of OpenFlow. OpenFlow provides bucket weights that may be used as a criterion in the selection process. An exemplary use case for this group type is primitive load balancing, when the action buckets take turns in processing packets and equally share the load.
- **Indirect** (required) – in this group, only one bucket may be defined. It allows multiple flow entries to be configured for the same output port. This group type is meant for fast and efficient convergence because only one entry has to be updated in case the next hop changes, while normally a number of flow entries proportional to the number of involved switches would have to be updated.
- **Fast failover** (optional) – the first live action bucket in the group is executed. Buckets in this group are watched for liveness and evaluated in the defined order. In case a link goes down, the switch can swiftly change the forwarding to the next bucket in the row. If no buckets are alive, the packets are dropped.

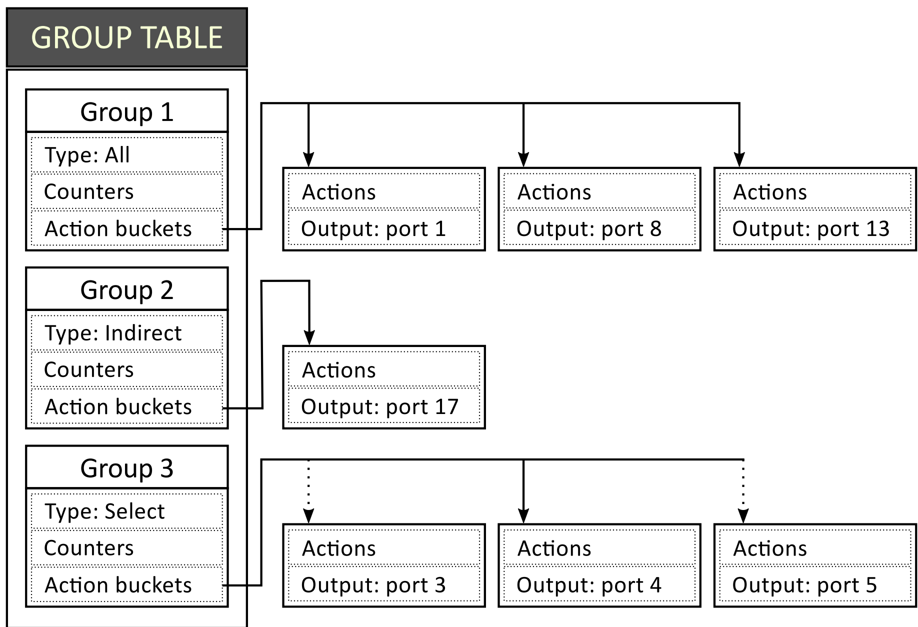


Figure 33: An illustration of a group table with group entries and their action buckets. Inspired by Figure 5.13 from *Software Defined Networks: A Comprehensive Approach* (Göransson and Black, 2014, p. 102).

B.5 Meters

Meters (or per-flow meters) are an OpenFlow instrument for implementing various operations to help ensure the desired Quality of Service. Meters are associated with particular flow entries and measure the rate of packets flowing through. A meter may be attached to more flow entries, than it measures their aggregate rate. Packets are then treated in accordance to the current measured rate. To provide broader functionality, meters can be combined with queues that, on the other hand, are attached to ports (Open Networking Foundation, 2012b, p. 15).

Meters are defined in a meter table using meter entries, which have a unique identifier, meter bands, and counters. Each meter entry may have one or more meter bands. A meter band specifies the rate at which it applies and how the affected packets should be handled. Packets are processed only by one meter band and the choice depends on the current measured meter rate. A meter band with the highest configured rate that is lower than the current rate is chosen. In other words, the rate works as a threshold and the meter band applies when the threshold is exceeded. If the current rate is lower than the configured rate of any meter band, no meter band is applied (Open Networking Foundation, 2012b, p. 15).

A meter band has a band type, a rate, counters, and type-specific arguments. The rate may be expressed in kilobits per second or packets per second and defines the lowest rate at which the meter band applies. There are currently no band types required to be implemented in a switch and two optional band types (Open Networking Foundation, 2012b, p. 15–16):

- **Drop** – discard the packet.
- **DSCP remark** – decrement the drop precedence of the DSCP field in the IP header of the packet, which means the packet will more likely be dropped in case of queue congestion (Göransson and Black, 2014, p. 112).

In Figure 34, a simple meter table with two meters is depicted. Packets 1A and 1B belong to a flow entry associated with meter 1 and are processed by meter bands that have the highest configured rate that is lower than the current measured rate. Or in other words, the packets are processed by the highest meter band whose threshold rate have been crossed. Packet 2A belongs to a flow entry associated with meter 2 and is not processed by the meter band because the current measured rate is lower than the rate set for the band. In other words, the threshold has not been crossed.

Meter-level counters are updated for all packets processed by the meter, while meter-band-level counters are updated only when the band is used (Göransson and Black, 2014, p. 112).

Although the functionality of meters is basic at this point, it can be supposed that the functionality could be greatly enhanced in following versions of OpenFlow and

more means for ensuring Quality of Service could then be available (Göransson and Black, 2014, p. 111).

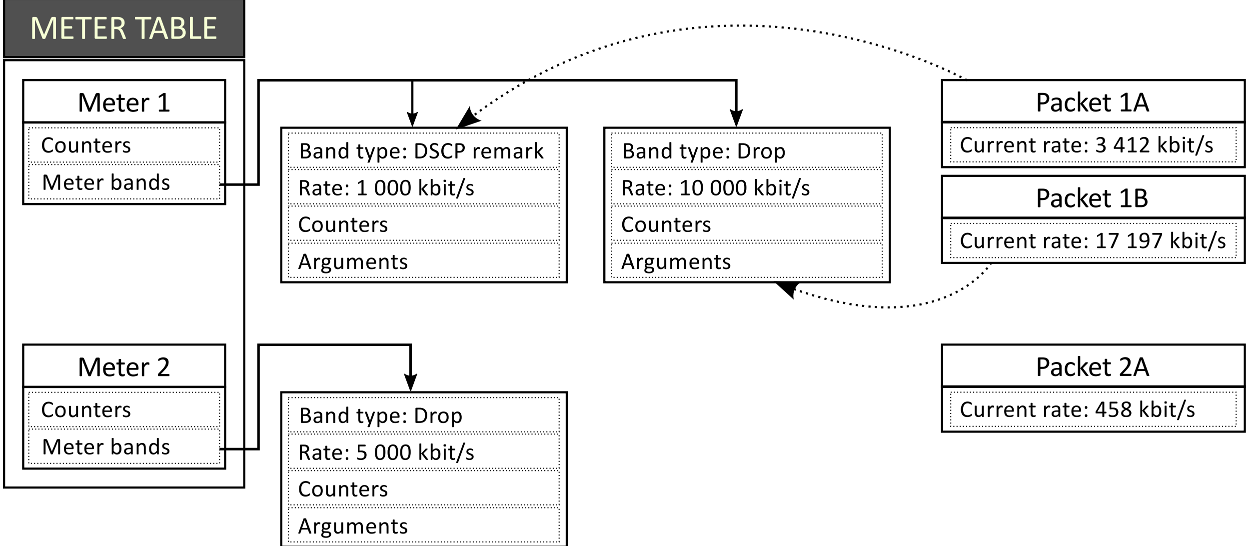


Figure 34: An illustration of a meter table with meter entries and their meter bands. Inspired by Figure 5.16 from *Software Defined Networks: A Comprehensive Approach* (Göransson and Black, 2014, p. 111).

B.6 Other improvements

As was mentioned before, there have been many improvements over the evolution of OpenFlow, but not all of them can be given attention in this work. A few more are briefly mentioned in this section.

VLAN and MPLS are fully supported. There are actions for adding, removing, and modifying VLAN tags and MPLS labels in packet headers. Multiple levels of tags can also be handled (Open Networking Foundation, 2012b, p. 98; Göransson and Black, 2014, p. 102–103).

To ensure high availability, it was specified in greater detail how a switch may be connected to multiple controllers. Controllers now take roles – equal, master, or slave. Equal and master controllers may fully control the switch. Either all controllers are equal or there is one master and the other controllers are slaves. A slave controller may retrieve information from the switch, but it cannot modify its state (Göransson and Black, 2014, p. 108; Open Networking Foundation, 2012b, p. 26–27). A controller can also define what particular messages it wants to receive from a switch, so that unnecessary load is not created (Göransson and Black, 2014, p. 112; Open Networking Foundation, 2012b, p. 103).

Previously, it was expected that all interfaces of a switch were physical. It is now possible for a switch to define arbitrary virtual ports that, for example, represent

a Link Aggregation Group (LAG) or a tunnel. Packets may then be forwarded to these switch-defined virtual ports (Open Networking Foundation, 2012b, p. 98; Göransson and Black, 2014, p. 103).

As for small, but practical changes, a duration field was added to statistics for easy calculations of packet and byte rate from corresponding counters. On the other hand, it is now possible to disable counters for certain flow entries in order to boost flow-handling performance (Open Networking Foundation, 2012b, p. 104).

More information and details regarding OpenFlow v. 1.3.0 and the preceding changes can be found in the complete specification (Open Networking Foundation, 2012b) or in a more talkative way in the book of Göransson and Black (2014, p. 99–116).

C Mininet commands

As for the basic usage, Mininet can be started from a terminal, but it needs superuser privileges to work. The command is following:

```
sudo mn
```

C.1 Starting attributes

Several options and attributes may be added to the command (Mininet, 2016c). The following ones are relevant for the work of this thesis:

- `--controller` – defines the controller used to control the virtual network. A controller residing on a remote server may be connected using the following setting, which requires designating the IP address of the server:
  - `remote,ip=`*`controller-IP-address`*.

  Available for direct installation and use with Mininet, there are as well local controllers:
  - `ref` - OpenFlow 1.0 reference controller simply turns the switches it manages into L2-MAC learning switches. It does not provide a northbound API (OpenFlow Switch Consortium, 2011a).
  - `ovsc` - Open vSwitch controller (ovs-testcontroller, formerly known as ovs-controller) has the same main function as the reference controller, but since it is still worked on, its capabilities are broader. For example, it experimentally supports OpenFlow v. 1.3 or allows pushing arbitrary flow entries into switches. This controller is not intended for production deployments (Open vSwitch, 2016).
  - `ryu` – Ryu is an open-source controller written in Python that provides well-defined API for applications. It is under active development and fully supports OpenFlow up to version 1.5 (Nippon Telegraph and Telephone Corporation, 2014).
  - `nox` – NOX was the first OpenFlow controller and it served as a basis for a number of research projects. It provides a C++ API for OpenFlow v. 1.0 (NOX, 2012; Göransson and Black, 2014, p. 247).

  The network can also be started without a controller using the following option:
  - `none`
- `--topo` – is used to create a network from a topology template, which can be further customized by additional parameters. Following topology templates are shipped with Mininet:
  - `single,k=2` – a single switch connected to k hosts.

   - `reversed,k=2` – a single switch connected to k hosts, but with reversed ports, which means that the lowest-numbered host is connected to the highest-numbered switch port.
   - `minimal` – one switch with two hosts.
   - `linear,k=2,n=1` – a linear topology with k switches, each having n hosts.
   - `torus,x,y,n=1` – a 2D torus topology with specified x and y dimensions of the torus. Minimal allowed dimension is 3x3 switches with each switch having n nodes. This topology requires careful preparation as it creates loops and has to be used with Spanning Tree Protocol.
   - `tree,depth=1,fanout=2` – a tree topology with *depth* levels of switches, each having *fanout* of hosts.

  Every topology, except for the torus, can be used without specifying any parameters since all of them have predefined default values.

- `--switch` – defines the switch software used for the virtual switches. By default, Open vSwitch is used, which is an open-source production-quality multilayer virtual switch. Open vSwitch has full OpenFlow v. 1.3 support since version 2.3, for lower versions, only experimental OpenFlow v. 1.3 support is provided and not enabled by default. In case a lower version of Open vSwitch is used on the system, the support of OpenFlow v. 1.3 can be enabled by these attributes:
   - `ovs,protocols=OpenFlow13`
- `--custom` – reads classes from Python script files provided as attributes. Using this option, for example custom topologies may be imported, which may then be used by the option `--topo`.

When the command `sudo mn` is used with no options, the network is started with a default setting. As for the default controller, one of the internal controllers is to be used if available – OpenFlow reference controller or Open vSwitch controller (Mininet, 2016a). As for the default topology, the minimal topology is used, and as for the default switch, Open vSwitch is used (Mininet Team, 2016c).

In the following examples of the starting command, various options are combined:

```
sudo mn --custom campus_river.py,campus_park.py --topo groundfloor
```

```
sudo mn --controller remote,ip=192.168.0.12
--switch ovs,protocols=OpenFlow13 --topo tree,depth=3
```

In the first command line, two Python script files are imported and a custom topology *groundfloor* is used for building the network. The default controller and switch software are used. In the second command line, a remote controller is connected to control the network, which is created from the tree topology template with 3 levels

of switches. As for the switch software, Open vSwitch is used with OpenFlow v. 1.3 enabled.

For other possible options, the interested reader is encouraged to investigate the manual page – through the command `man mn` or online at Ubuntu Manpage (Mininet, 2016c) – and the Mininet documentation (Mininet, 2016b).

C.2 CLI commands

When the network is prepared and the CLI is ready, several commands can be executed. The following are the basic ones for elementary operation:

- `help` – lists all available CLI commands. It can also provide a brief help for a specific command if used as - help command.
- `exit` – destroys the virtual network and exits Mininet.
- `net` – lists all network connections.
- `dump` – dumps information about all nodes.
- `pingall` – performs a ping between all hosts, that is from each host to every other host.
- `link` *`node1 node2`* `[up/down]` – brings up or down a link between the two specified nodes.
- `switch` *`switch-name`* `[start/stop]` – starts or stops the designated switch.
- `xterm` *`node1 node2`* `...` – starts an xterm terminal window for each of the given nodes.
- `gterm` *`node1 node2`* `...` – starts a gnome terminal window for each of the given nodes.

The commands `xterm` and `gterm` allow direct manipulation of the nodes. Commands may also be sent to specific nodes from the CLI, as if they were executed through a specific terminal window, by first stating the desired node and then the command for it to execute, for example:

```
h1 ping h4
h2 ifconfig h2-eth1 10.0.0.32/8 up
```

When necessary, the names of the nodes are automatically replaced by the interpreter for corresponding IP addresses, such as in the `ping` command.

D HPE VAN SDN Controller REST API – Used structures

In this chapter I describe the resources from the REST API of the HPE VAN SDN Controller that I used in the application to retrieve or push data to the network through the controller. Messages that are received or pushed are thoroughly explained for every resource.

For both data retrieval and data pushing, the base address for resource identifiers is always following:

```
https://controller-IP-address:8443
```

The content of messages interchanged between an application and the REST API is in the JSON format.

D.1 Data retrieval

D.1.1 Datapaths

Because a switch as a networking device may be comprised of different components, the name *datapath* is used throughout the OpenFlow specification and OpenFlow messages to point to the fast packet forwarding part (OpenFlow Switch Consortium, 2011b). This term will be used in the following chapters to preserve unity.

To retrieve information about datapaths, the following resource must be accessed at the controller:

```
GET /sdn/v2.0/of/datapaths
```

A message of the following structure is then received. Descriptions of the items are taken from the REST API JSON data model mentioned in chapter 3.3.1:

```
{
  "datapaths": [
    {
      # Datapath identifier
      "dpid": "00:00:00:00:00:00:00:01",
      # Highest common supported version of the OpenFlow protocol
      # in the datapath and the controller
      "negotiated_version": "1.3.0",
     # Time when the datapath successfully connected to the controller
      "ready": "2016-11-27T17:22:53.257Z",
      # Time when the last message was received from the datapath
      "last_message": "2016-11-27T17:23:20.558Z",
      # Maximum number of packets the datapath can buffer at once
      "num_buffers": 256,
```

```
      # Maximum number of flow tables supported by the datapath
      "num_tables": 254,
      # Manufacturer description
      "mfr": "Nicira, Inc.",
      # Hardware version
      "hw": "Open vSwitch",
      # Software version
      "sw": "2.0.2",
      # Serial number
      "serial": "None",
      # Datapath description
      "desc": "None",
      # Datapath IP address via the main connection
      "device_ip": "192.168.0.89",
      # Datapath TCP port via the main connection
      "device_port": 36834,
      # Datapath capabilities
      "capabilities": [
        # Statistics for flow entries
        "flow_stats",
        # Statistics for flow tables
        "table_stats",
        # Statistics for datapath ports
        "port_stats",
        # Statistics for datapath queues
        "queue_stats"
      ]
    }
  ]
}
```


Concerning items necessary for the application, `dpid` and `device_ip` are used to identify the datapath in both the visualization and OpenFlow messages, and `negotiated_version` serves to choose the appropriate messages for communication.

D.1.2 Nodes

Nodes represent end-user devices connected to switches. To retrieve their data, the following resource must be accessed at the controller:

```
GET /sdn/v2.0/net/nodes
```

A message of this structure is then received:

```
{
  "nodes": [
    {
      # Node IP address
      "ip": "10.0.0.2",
      # Node MAC Address
      "mac": "22:51:c1:54:43:04",
      # Node VLAN identifier
      "vid": 0,
      # Parent datapath DPID
      "dpid": "00:00:00:00:00:00:00:01",
      # Parent datapath port to which the node is connected
      "port": 2
    }
  ]
}
```

For the application, `ip` and `mac` are used to identify the node itself, and `dpid` and `port` are used to indicate its relationship with the parent datapath.

D.1.3 Links

Links represent connections only between datapaths and do not include connections between datapaths and nodes, which are expressed in the data available in the resource *nodes*.

To retrieve data for all links, the following resource must be accessed at the controller:

```
GET /sdn/v2.0/net/links
```

To retrieve data for links that connect to a particular datapath, the DPID of this datapath has to be provided in the resource address:

```
GET /sdn/v2.0/net/links?dpid={dpid}
```

A message of this structure is then received:

```
{
  "links": [
    {
      # Source datapath DPID
      "src_dpid": "00:00:00:00:00:00:00:01",
      # Source datapath port
      "src_port": 1,
      # Destination datapath DPID
      "dst_dpid": "00:00:00:00:00:00:00:02",
```

```
      # Destination datapath port
      "dst_port": 3,
      "info": {
        # Link type (directLink, multihopLink, tunnel)
        "link_type": "directLink"
      }
    },
    {
      "src_dpid": "00:00:00:00:00:00:00:02",
      "src_port": 3,
      "dst_dpid": "00:00:00:00:00:00:00:01",
      "dst_port": 1,
      "info": {
        "link_type": "directLink"
      }
    }
  ]
}
```

To identify connections between datapaths, `src_dpid`, `src_port`, `dst_dpid`, and `dst_port` are used in the application. There may be two records for every connection – one from point A to point B, and another from point B to point A, but there may also be only one of these records.

D.1.4 Port statistics

Traffic statistics are based on data gathered from datapath port counters. To retrieve corresponding data, the following resource has to be accessed at the controller:

To retrieve statistics for ports of a particular datapath, the DPID of this datapath must be provided in the resource address:

```
GET /sdn/v2.0/of/stats/ports?dpid={dpid}
```

A message of the following structure is then received. Descriptions of the items are taken from the OpenFlow Switch Specification (v. 1.3.0 quote):

```
{
  "stats": [
    {
      # Datapath DPID
      "dpid": "00:00:00:00:00:00:00:01",
      # Datapath OpenFlow protocol version
      "version": "1.3.0",
      "port_stats": [
        {
```

```
     # Datapath port number
     "port_id": 1,
     # Number of received packets
     "rx_packets": 39,
     # Number of transmitted packets
     "tx_packets": 177,
     # Number of received bytes
     "rx_bytes": 3582,
     # Number of transmitted bytes
     "tx_bytes": 12774,
     # Number of packets dropped by RX
     "rx_dropped": 0,
     # Number of packets dropped by TX
     "tx_dropped": 0,
     # Number of receive errors (a~super-set of more specific
     # receive errors, should be greater than or equal to
     # the sum of rx_*_err values)
     "rx_errors": 0,
     # Number of transmit errors
     "tx_errors": 0,
     # Number of collisions
     "collisions": 0,
     # Time the port has been alive in seconds
     "duration_sec": 652,
     # Time the port has been alive in nanoseconds
     # beyond duration_sec
     "duration_nsec": 690000000,
     # Number of receive CRC errors
     "rx_crc_err": 0,
     # Number of receive frame alignment errors
     "rx_frame_err": 0,
     # Number of packets with RX overrun
     "rx_over_err": 0
    },
    {
     "port_id": 4294967294,
     ...
    }
   ]
  }
 ]
}
```

To measure the utilization of links between datapaths, `rx_bytes` and `tx_bytes` are used in the application. Except for standard ports, a port number 4294967294 may also appear in the statistics. This port is a local port used for the out-of-band communication between the switch and the controller.

Items present in messages for datapaths and port statistics depend on the particular switch. For a different switch, different items might appear depending on the implementation of optional features. The interested reader is encouraged to investigate the REST API JSON data model mentioned in chapter 3.3.1 for details on all possible parameters and values of the messages.

D.2 Data pushing

D.2.1 Flow entries

Flow entries work as forwarding rules in the datapaths they are installed to. They may be accessed only for a specific datapath, its DPID therefore always has to be provided in the address that is used to access and modify the resource:

```
POST /sdn/v2.0/of/datapaths/{dpid}/flows
```

The message for the controller is different for OpenFlow v. 1.0 and for OpenFlow v. 1.3. because of the changes in the flow_mod message throughout the evolution of the OpenFlow protocol.

For OpenFlow v. 1.0, the structure of the message to be sent to the controller is following:

```
{
  "flow":{
    # Flow entry priority
    "priority":30000,
    # Flow entry idle timeout
    "idle_timeout":60,
    # Flow entry hard timeout
    "hard_timeout":600,
    # Match fields
    "match":[
      # IPv4 source address
      {"ipv4_src":"192.168.2.12"},
      # IPv4 destination address
      {"ipv4_dst":"192.168.22.135"},
      # Datapath input port
      {"in_port":4},
      # Ethernet frame type - EtherType
      {"eth_type":"ipv4"},
```

```
      # IP protocol
      {"ip_proto":"tcp"},
      # TCP source port
      {"tcp_src":50550},
      # TCP destination port
      {"tcp_dst":80}
    ],
    # Actions
    "actions":[
      # Output action with datapath output port
      {"output":"1"}
    ]
  }
}
```

For OpenFlow v. 1.3, the structure of the message to be sent to the controller is following:

```
{
  "flow":{
    "priority":50000,
    "idle_timeout":0,
    "hard_timeout":0,
    "match":[
      {"ipv4_src":"192.168.191.43"},
      {"ipv4_dst":"216.58.211.4"},
      {"in_port":7},
      {"eth_type":"ipv4"},
      {"ip_proto":"udp"},
      {"udp_src":50123},
      {"udp_dst":517}
    ],
    # Instructions
    "instructions":[
      {
        # Apply Actions
        "apply_actions":[
          # Output action with datapath output port
          {"output":"12"}
        ]
      }
    ]
  }
}
```

Depending on the configuration of the path defined in the application, match fields regarding IP protocol and TCP/UDP ports may or may not be present.

E Network data – algorithms

E.1 Loading data

Once the interface is connected to a controller, data about the network can be downloaded and necessary information stored in the data structures. In Figure 35, loading of datapaths is depicted. First, all datapaths are retrieved, and then they are one by one processed in a cycle, where required information is taken and saved in the Datapath object. A reference to this object is then kept in the Network object, which aggregates all the data structures.

Figure 36 depicts loading of nodes, which is similar to the process of loading datapaths. Since a node is always connected to a particular datapath, the information about their relationship has to be stored. Connections are made between the Node and Datapath objects, and also a Link object capturing the relationship is created. The new Node and Link objects then have to be added to the Network object as well.

Figure 37 is concerned with processing links. While the Link object may connect two datapaths or a datapath and a node, the link retrieved from the controller always connects two datapaths. Based on this link, the source and destination Datapath objects are connected and a Link object capturing the connection is created. The Link object is then added to the Network object.

Figure 35: Loading datapaths – a flowchart of the process.

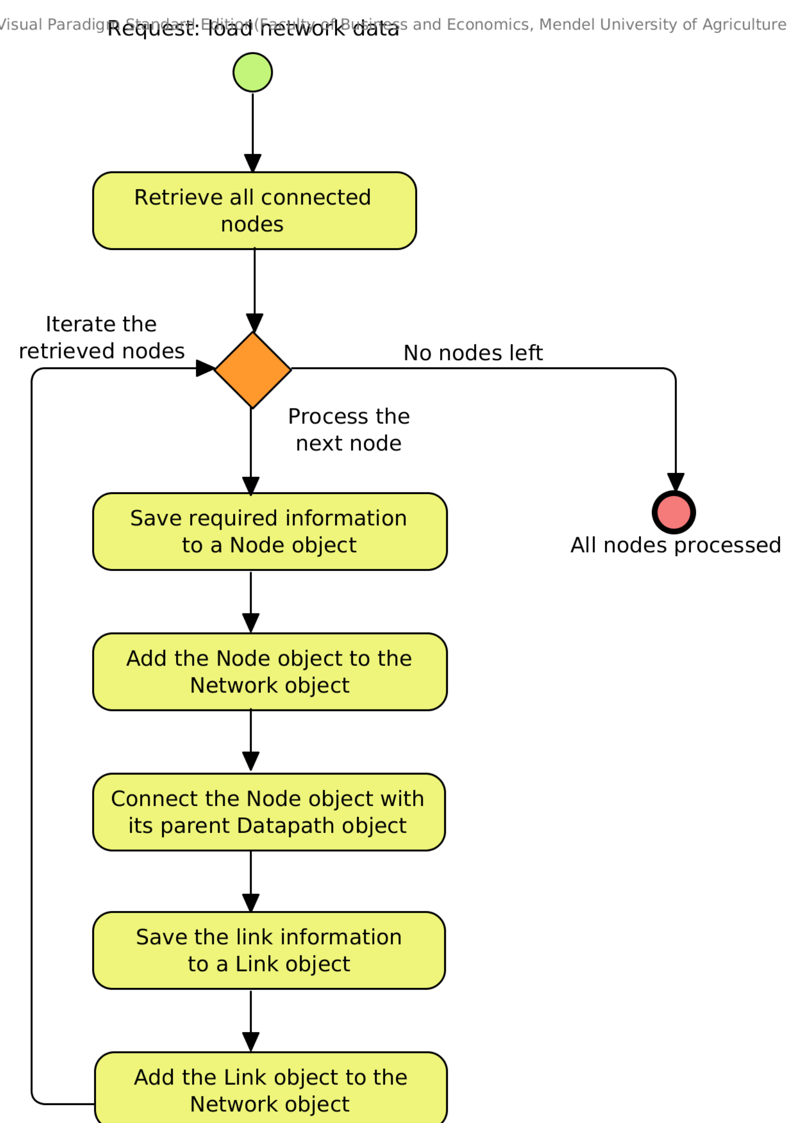


Figure 36: Loading nodes – a flowchart of the process.

Visual Paradigm Standard Edition(Faculty of Business and Economics, Mendel University of Agriculture and Fores

Figure 37: Loading links – a flowchart of the process.

E.2 Updating data

To present to the user the most recent state of the network, Visdan periodically inquires the network controller to find out about any occurred changes. In Figure 38, the process of updating datapaths is depicted. First, all datapaths are downloaded from the controller and one by one checked in a cycle. If a datapath exists in the original network state that has the same identifier (DPID) as the checked datapath, their objects are compared. If any attributes changed, the objects are not equal and the original datapath is removed. In case the checked datapath is not present in the original network state, which also applies to the case of removing the original datapath, it is saved. All datapaths that are left from the original network state, but are not present in the new network state (that is in the updated one), are removed.

Steps related to saving a datapath are the same as described in the previous section in Figure 35. When removing a datapath, these steps have to be undone. It means that the Datapath object has to be removed from the Network object and its reference lost.

Figure 39 depicts the process of updating nodes, which is essentially the same as for the datapaths. Since the links retrieved from the controller are concerned only with connecting datapaths, links between nodes and their parent datapaths cannot be checked in the same operation. A node therefore has to be individually disconnected from its parent datapath when being removed. As it was for datapaths, the Node object then has to be removed from the Network object and its reference lost. Steps for saving a new node are the same as described in Figure 36.

Figure 40 is concerned with updating links between datapaths, and their port statistics. In a cycle, each of the datapaths present in the new network state is taken. First, links connected to this datapath are download. For each link, the other datapath is connected to this datapath and the link is saved as described in Figure 37. The datapath is then checked whether it has any ports that used to be connected to another datapath, but are not anymore, and these ports are cleared. When the datapath has a new object, it will not have any vacated ports. Statistics for all active ports on the datapath are then downloaded, and the port statistics stored in the PortStats objects are updated with recent data for each port.

It is not possible to merge these two datapath cycles into one. The first cycle ensures that there is an object representation for every datapath. It would be possible to check the links during the first cycle, but a missing object for the other datapath would cause problems.

As the last step, depicted in Figure 41, the Link objects that are no more valid are removed.

During the whole updating process, information about added or removed datapaths or nodes, and connected or disconnected links is gathered in the Update object in

the form of references to the involved objects. The Update object is then returned as a result of the update function.

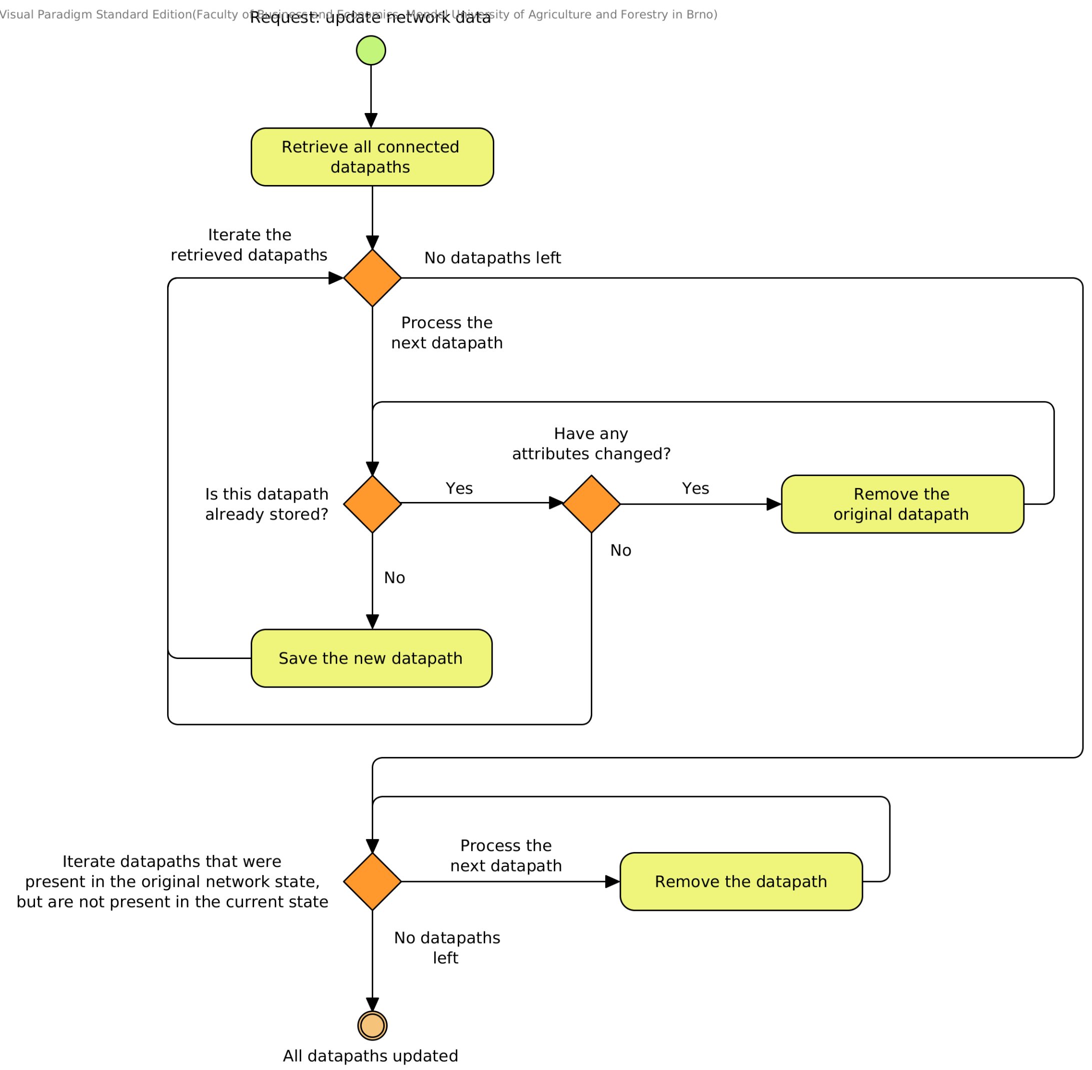


Figure 38: Updating datapaths – a flowchart of the process.

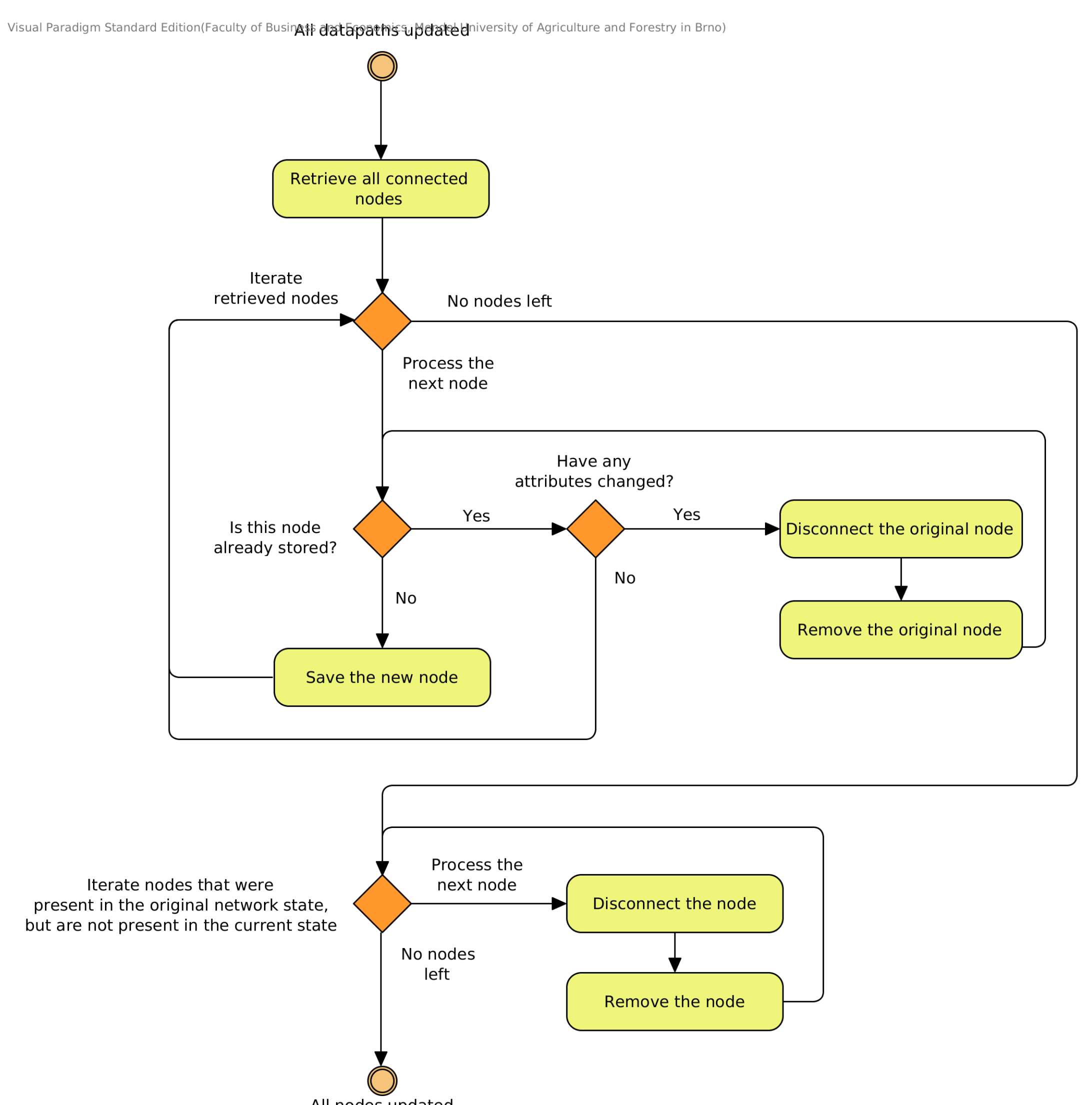


Figure 39: Updating nodes – a flowchart of the process.

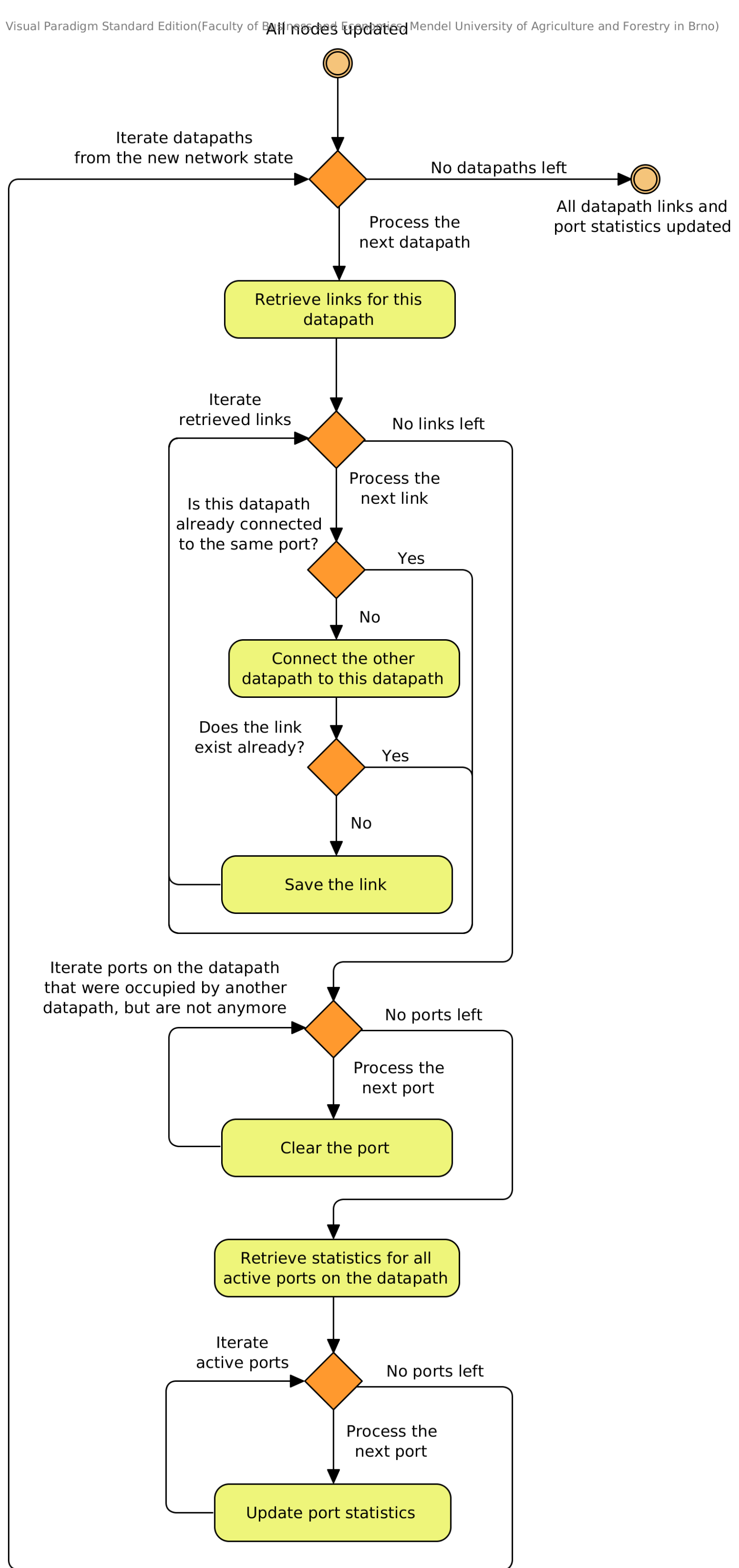


Figure 40: Updating datapath links and port statistics – a flowchart of the process.

Visual Paradigm Standard Edition(Faculty of Business and Economics, Mendel University of Agriculture and Forestry in Brno)

All datapath links and
port statistics updated

Process the
next link

Iterate links that were
present in the original network state,
but are not present in the current state

Remove the link

No links
left

All network data processed

Return the summary
of the changes

Updating finished

Figure 41: Removing invalid links – a flowchart of the process.

F Tree-like data structure – algorithms

F.1 Creation

The process of creating the tree-like data structure is depicted in Figure 42. To create the structure, it is necessary to have one or more live core or distribution datapaths, which serve as roots for the tree. These datapaths are then processed in a breadth-first search. For every datapath a TreeDatapath object is created and the corresponding graphical object is assigned. It is also necessary to connect the new TreeDatapath object with its predecessor TreeDatapath object. Next, the devices connected to the datapath are processed. If the device is a datapath, it is added to the queue for the next row in the tree unless it is already there or unless it has been already processed, which means it would be on the same or previous row. If the device is a node, a TreeNode object is directly created for it, the corresponding graphical object is assigned, and the object is added to its parent TreeDatapath object.

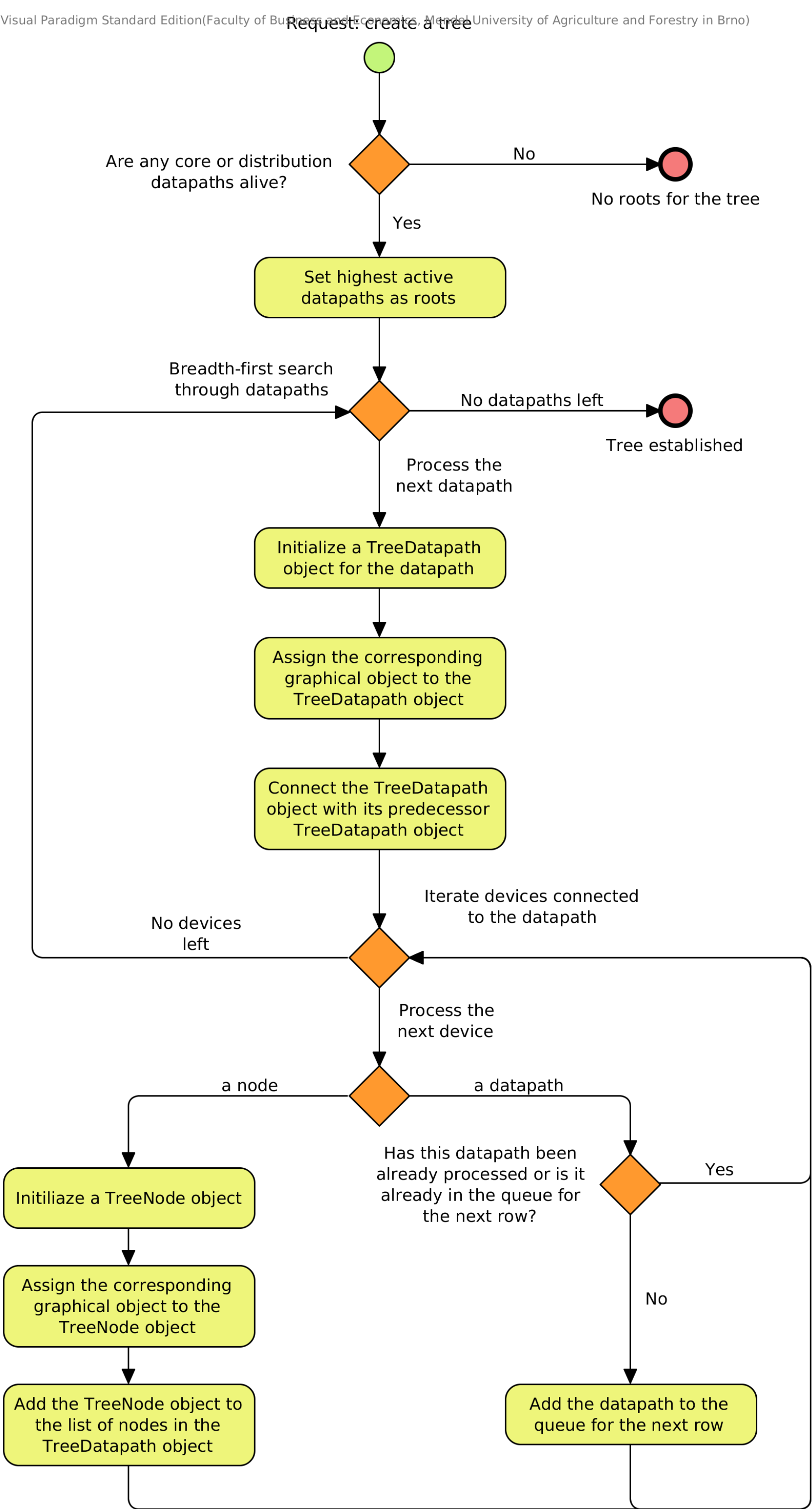


Figure 42: Creating the tree-like structure – a flowchart of the process.

F.2 Drawing

Once the data structure is complete, it may be used for drawing, whose process is depicted in Figure 43. In the beginning, the initial position is defined. The tree roots are then iterated and for each of them a depth-first search is performed, which is further described below. Once the search is done, it returns a position, where the sub-tree of the root ended, that is used to set the new initial position for the next root. When all roots are processed and the tree is drawn, elements that are not included in the tree are added to the tree visualization using the graph algorithm that is used to create the plain graph.

The algorithm of the previously mentioned depth-first search is depicted in Figure 44. If the given object is a TreeDatapath, the algorithm begins with processing its sub-tree. The initial position for its successors is set and the succeeding datapaths as well as connected nodes are processed, again using the depth-first search. The position returned from the search is then set as the initial position for the next successor or node. Once the sub-tree of the datapath is completely processed, its graphical element is placed on the scene in a position such that it is above the sub-tree vertically and in its center horizontally. The end position of the sub-tree is then returned to serve as the initial position for the neighboring sub-tree. If the processed object is a TreeNode, its graphical element is placed on the scene and the end position is returned.

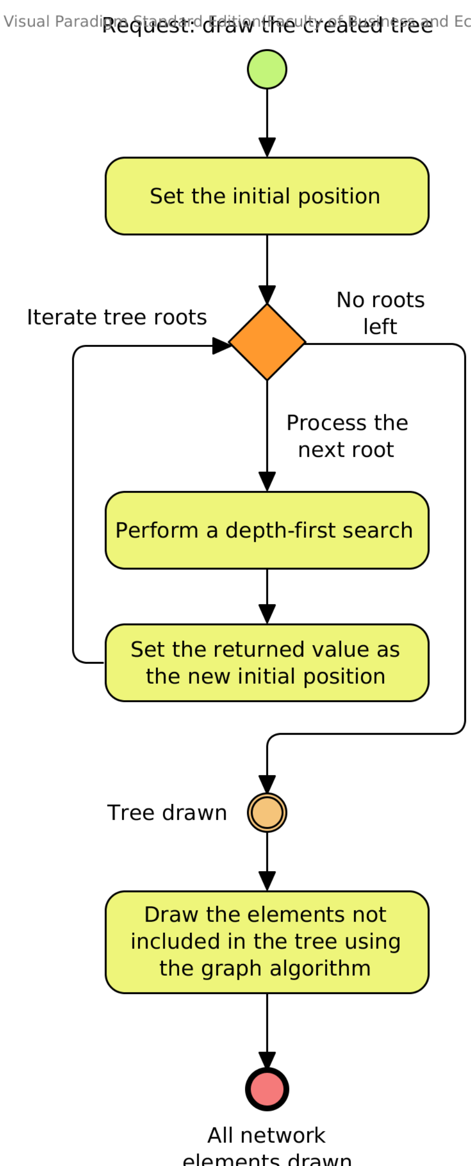


Figure 43: Drawing the tree-like structure – a flowchart of the process.

Visual Paradigm Standard Edition(Faculty of Business and Economics, Mendel University of Agriculture and Forestry in Brno)

Perform a depth-first search

TreeDatapath object

TreeNode object

Set the initial position for the succeeding row

Place the graphical element on the scene according to the initial position

Return the end position of the graphical element

Iterate successing datapaths

No datapaths left

Process the next datapath

Perform depth-first search

Set the returned value as the new initial position for the next successor in the row

Iterate connected nodes

No nodes left

Process the next node

Perform depth-first search

Set the returned value as the new initial position for the next node

Place the graphical element on the scene in the centre above its sub-tree

Return the end position of the sub-tree

Depth-first search finished

Figure 44: Depth-first search – a flowchart of the process.

G Mininet testing topology

G.1 Custom topology

The topology depicted in Figure 25 can be created as a custom topology. The Mininet command is following:

```
sudo mn --controller=remote,ip=192.168.1.89 --custom=~/topo.py
--topo=visdan1
```

Soure code 17: topo.py

```python
#!/usr/bin/python

from mininet.net import Mininet
from mininet.topo import Topo

class VisdanOne(Topo):
    def build(self):
        core1 = self.addSwitch('s1')
        dist1 = self.addSwitch('s2')
        dist2 = self.addSwitch('s3')
        accs1 = self.addSwitch('s4')
        accs2 = self.addSwitch('s5')
        accs3 = self.addSwitch('s6')
        accs4 = self.addSwitch('s7')

        self.addLink(core1, dist1, port1=1, port2=1)
        self.addLink(core1, dist2, port1=2, port2=1)
        self.addLink(dist1, accs1, port1=2, port2=1)
        self.addLink(dist1, accs2, port1=3, port2=1)
        self.addLink(dist2, accs3, port1=2, port2=1)
        self.addLink(dist2, accs4, port1=3, port2=1)

        host1 = self.addHost('h1')
        host2 = self.addHost('h2')
        host3 = self.addHost('h3')
        host4 = self.addHost('h4')
        host5 = self.addHost('h5')
        host6 = self.addHost('h6')
        host7 = self.addHost('h7')
        host8 = self.addHost('h8')

        self.addLink(accs1, host1, port1=2)
        self.addLink(accs1, host2, port1=3)
        self.addLink(accs2, host3, port1=2)
        self.addLink(accs2, host4, port1=3)
        self.addLink(accs3, host5, port1=2)
        self.addLink(accs3, host6, port1=3)
        self.addLink(accs4, host7, port1=2)
        self.addLink(accs4, host8, port1=3)
```

```
topos = {'visdan1': lambda: VisdanOne()}
```

G.2 Default topology

The same topology can be generated using the default topologies using the following command:

```
sudo mn --controller=remote,ip=192.168.1.89 --topo=tree,depth=3
```