Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Telecommunication Engineering

Deploying SDN architecture in Open Optical Transport Networks

Master thesis

Bc. Ivan Eroshkin

Master programme: Electronics and Communication
Branch of study: Communication Systems and Networks

Author: Bc. Ivan Eroshkin
Supervisor: Doc. Ing. Leoš Boháč, Ph.D.
Company supervisor: M.Sc. Dominique Verchere, Ph.D.

Prague, January 2020

Thesis Supervisor:
Doc. Ing. Leoš Boháč, Ph.D.
Department of Telecommunication Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Technická 2
160 00 Prague 6
Czech Republic
bohac@fel.cvut.cz

Company Supervisor:
M.Sc. Dominique Verchere, Ph.D.
ENSA Lab
Nokia Bell Labs France
7 Route de Villejust
91620 Nozay, France
dominique.verchere@
nokia-bell-labs.com

Declaration

I hereby declare that I have written this master thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis.

In Prague, January 2020

.....
Bc. Ivan Eroshkin

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Eroshkin** Jméno: **Ivan** Osobní číslo: **434675**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra telekomunikační techniky**

Studijní program: **Elektronika a komunikace**

Studijní obor: **Komunikační systémy a sítě**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Využití konceptu softwarově definovaných sítí v otevřených optických transportních sítích

Název diplomové práce anglicky:

Deploying SDN architecture in Open Optical Transport Networks

Pokyny pro vypracování:

Hlavním cílem této diplomové práce je nastudovat architektury softwarově definovaných sítí a určit, jak mohou napomoci při řešení rostoucích požadavků na širší automatizaci v optických přenosových sítích. Úkolem studenta je metodicky vyhodnotit populární open-source kontroléry, jako je OpenDayLight, či ONOS a vybrat ten optimální pro implementaci. Kromě výše uvedené analýzy bude výstupem práce jednoduchý demonstrační příklad sítě ověřující spojení typu konec-konec.

Seznam doporučené literatury:

[1] EDELMAN, Jason, Scott LOWE a Matt OSWALT. Network programmability and automation: skills for the next-generation network engineer. Sebastopol, California: O'Reilly Media, 2018. ISBN 978-1-491931-257.
[2] GORANSSON, Paul, Chuck BLACK a Timothy CULVER. Software defined networks: a comprehensive approach. Second edition. Singapore: Morgan Kaufmann, [2017]. ISBN 0-12804-555-8.
[3] CHADHA, Devi. Optical WDM networks: from static to elastic networks. Hoboken, NJ, USA: Wiley, [2019]. ISBN 978-1-119393-269.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Leoš Boháč, Ph.D., katedra telekomunikační techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **25.09.2019** Termín odevzdání diplomové práce: **07.01.2020**

Platnost zadání diplomové práce: **30.09.2021**

doc. Ing. Leoš Boháč, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Abstract

With the rising demands on the network throughput, latency and security, legacy control networking concepts should be reconsidered. Software-Defined Networking (SDN) is one of the possible solutions, to which telecommunication industry is moving.

This work presents current state-of-the-art in Software-Defined Networking and focuses on some open-source solutions of SDN controllers, like ONOS and OpenDaylight. Main focus is to understand how SDN can help to solve increasing demand for broader automation in Optical Transport Networks.

The practical section is divided in two parts. Within the first part I focused on extending functionality of SDN controller to facilitate more efficient configuration and control of optical network devices. Main contribution was to implement additional features to SDN drivers for Nokia 1830 PSS (ROADM) and extend functionality of Nokia 1830 PSI-2T (Optical Transponder) driver.

Second part is dedicated to the Alarm Correlation problematic in open optical networks. We designed, developed an Alarm Correlation function as a SDN application then we tested it on emulated optical devices to prove the concept.

Keywords: Software-Defined Networking, SDN, NETCONF, OPENCONFIG, Open Network Operating Systems, ONOS 2.x, ODTN, Open and Disaggregated Transport Network, Open Optical Transport Networks, Optical Transponder, ROADM, Reconfigurable Add/Drop Multiplexer, Future Networks, Alarm Correlation, Alarm Localization.

This work has been supported by **Nokia Bell Labs** Paris.

Résumé

Les réseaux optiques élastiques sont prévus pour le déploiement des réseaux de communication à très hauts débits de nouvelle génération à cause des demandes croissantes des services utilisateurs exigeant des contrôles précis en terme de débit, de latence et de sécurité. En conséquence, les solutions de contrôle existantes pour offrir des services de connectivité doivent être reconsidérées. Les approaches orientées logicielles "Software-Defined Networking" dites (SDN) sont les solutions de contrôle possibles pour les réseaux télécom que les opérateurs analysent actuellement en détails.

Dans ce context, cette thèse présente un état de l'art des solutions de contrôles des réseaux de transports dits "SDN" et propose des évaluations et des expérimentations basées sur des implémentations open-source de contrôleurs SDN comme précisément ONOS et OpenDaylight. Le principal objectif de la thèse est définir des cas d'utitisation basées sur ces implémentations logicielles SDN et de les expérimenter pour évaluer comment un contrôleur SDN peut permettre l'ouverture des réseaux optiques pour intégrer des équipements de différents constructeurs et peut permettre l'automatisation des différentes configurations de service de connectivité photonique.

Les expérimentations sont divisées en deux parties.

La première partie consiste à étendre les fonctionnalités d'un controller SDN pour configurer et contrôler de bout-en-bout des équipements de réseau optique. La principale contribution a été de concevoir, développer et d'intégrer des fonctions nouvelles basées sur le noeud optique « reconfigurable optical add drop multiplexer », ou « multiplexeur optique d'insertion-extraction reconfigurable » ROADM de NOKIA 1830 PSS et sur le « transpondeur optique » NOKIA 1830 PSI-2T Optical Transponder.

La deuxième partie traite des problèmes de gestion des alarmes dans les réseaux optiques SDN ouverts, c'est-à-dire multi-vendeurs, et principalement des fonctions de collection et de corrélation d'alarmes. Une preuve de concepts de fonction de corrélation d'alarme a été développée comme une application SDN et a été intégrée dans un banc expérimental de réseau optique pour tester cette application SDN. Ensuite cette application a été évaluée sur un réseau d'équipements optiques émulés pour évaluer son intérêt dans un réseau de transport opérationnel.

Mots-Clés: Software-Defined Networking, SDN, NETCONF, OPENCONFIG, Open Network Operating Systems, ONOS 2.x, ODTN, Open and Disaggregated Transport Network, Open Optical Transport Networks, Optical Transponder, ROADM, Reconfigurable Add/Drop Multiplexer, Future Networks, Alarm Correlation, Alarm Localization.

This work has been supported by **Nokia Bell Labs** Paris.

Abstrakt

Pro udržení tempa s rostoucími požadavky na přenosovou rychlost, latenci a bezpečnost je nutné zvážit současnou koncepci řízení sítí. Software-Defined Networking (SDN) je jedno z možných řešení, ke kterému telekomunikační průmysl směruje.

Tato práce představuje současný stav Software-Defined Networking a zaměřuje se na vybraná open-source řešení v oblasti SDN kontrolerů, jako je ONOS či OpenDaylight. Hlavním cílem této části práce je vysvětlit, jak může SDN pomoci vyřešit rostoucí požadavky na rozšíření automatizace v otevřených optických sítích.

Praktická část této práce je rozdělená do dvou oblastí. V rámci první oblasti jsem se zabýval rozšířením funkčnosti SDN kontroleru pro umožnění konfigurace a řízení optických komunikačních zařízení. Hlavním přínosem je implementace nových funkcionalit SDN driveru pro Nokia 1830 PSS (ROADM) a rozšíření funkcionality driveru pro Nokia 1830 PSI-2T (optický transpondér).

Ve druhé části práce jsem se zabýval problematikou korelace alarmů v otevřených optických sítích. Výsledkem je funkce pro korelaci alarmů ve formě SDN aplikace, kterou jsem dále otestoval na emulovaných optických zařízeních pro prokázání funkčnosti celého konceptu.

Klíčová slova: Software-Defined Networking, SDN, NETCONF, OPENCONFIG, Open Network Operating Systems, ONOS 2.x, ODTN, Open and Disaggregated Transport Network, Open Optical Transport Networks, Optical Transponder, ROADM, Reconfigurable Add/Drop Multiplexer, Future Networks, Alarm Correlation, Alarm Localization.

Tato práce byla stvořena za podpory **Nokia Bell Labs** Paříž.

Word of Gratitude

First of all, I would like to express my gratitude to **M.Sc. Dominique Verchere, Ph.D.** for giving me an opportunity to work as an Intern in Bell Labs and making this Diploma thesis happen. I would like to thank him for his encouragement, navigation during the whole process, useful tips and his belief in me. He is truly incredible person. It was a very precious experience working with him.

I would like to thank **Doc. Ing. Lukáš Vojtěch, PhD.** and **Ing. Zbyněk Kocur, PhD.** for navigating me over whole studies at CTU, helping me to gather practical experience outside the university and setting correct working mentality.

Most of all, I am grateful to my parents, **Ing. Igor Eroshkin** and **Ing. Nina Eroshkina**, for giving me an opportunity to study abroad, for their love, encouragement, infinite belief and financial support through all of my studies. I love you.

I am also especially thankful to my close friends for their support and belief in me during the past several years of my studies.

Acknowledgements

I would like to thank **Doc. Ing. Leoš Boháč, Ph.D.** for his supervision, patience, useful feedback, advises and administrative support during the whole work on this thesis.

Special thanks goes to **M.Sc. Andrea Campanella**, member of ONF, one of the ONOS project developers and ODTN project leader. His useful tips, personal time investments and his technical guidance over the ONOS project has helped this work to be finalized on time (in quite tough time constraints). Without this person it wouldn't be possible to achieve these results.

Also, I want to express my special gratitude to **M.Sc. Sagar Arora** for making my internship happened, for his support during the first time at the new place and for making my integration as an Intern easier. I really do appreciate this.

I am especially grateful to **Doc. Ing. Zdeněk Bečvář, Ph.D.** for providing me an opportunity to study abroad at **EURECOM** and encouraging me on this adventure. It was definitely precious experience with a lot of warm memories kept in mind.

I would also like to thank an Italian crew, who shared with me these beautiful moments during the internship.

Last, but not least, I would like to thank **Joaquim Oliveira** for his support and guidance on how to configure Nokia's equipment.

List of Tables

List of Figures

List of Acronyms

API Application Programmable Interface. xviii

CDC-F ROADM Colorless, Directionless, Contentionless, Flex Spectrum Reconfigurable Optical Add-Drop Multiplexer. xviii, 38, 41, 43

DWDM Dense Wavelength Division Multiplex. xviii, 34, 35, 83

EDFA Erbium Doped Fiber Amplifier. xviii, 84

FIFO first-in first-out. xviii

LCoS Liquid Crystal on Silicon. xviii, xxii, 37

MEMS Micro-Electro-Mechanical Mirror. xvii, xviii, xxii, 36

NETCONF Network Configuration Protocol. xvii, xviii, xxi, xxii, 13, 14, 16–19, 23, 24, 44, 52, 54, 60, 61, 77, 78

NFV Network Function Virtualization. xviii, xxi, 9

OADM Optical Add-Drop Multiplexer. xvii, xviii, 33, 34

ODL OpenDaylight. xviii, 23

ODTN Open and Disaggregated Transport Networks. xviii, 6

ONOS Open Network Operating System. xviii, 6, 23–25, 42

ROADM Reconfigurable Optical Add-Drop Multiplexer. xvii, xviii, xxii, xxiii, 6, 31, 34, 35, 37, 39, 42, 52, 84

RPC Remote Procedure Call. xviii, 14, 17

SDN Software-Defined Networking. xviii, 2

SFC Service Function Chaining. xviii, xxi, 10

T-API Transport API. xviii, 6, 80

WSS Wavelength Selective Switch. xvii, xviii, 34–37, 84

XC Cross-Connection. xviii, 33, 42, 43, 51, 52, 54

YANG Yet Another Next Generation. xviii, 13–15

Contents

Chapter 1

Introduction

1.1 Telecommunication industry evolution

Telecommunication industry came a long path and experienced a rapid evolution since it's beginning. Industry has started as a pure fixed networks used for transferring voice on long distances. During the evolution of the telecommunication industry a lot of different technologies were introduced. Some of them, such as Synchronous Digital Hierarchy (SDH), have been kept functional as a legacy networks. Some of them, i.e. Plesiosynchronous Digital Hierarchy (PDH) or ATM networks, are rarely represented today. Nowadays, telecommunication networks are more dynamic and data-centric.

When networks were used only for voice exchange, telecommunication market didn't evolve fast. From the time, when the internet was introduced to the humanity, requirements on network throughput and quality of service are keeping continuously increase. It triggered variety of changes in the network structure. For example, in transport networks were introduced optical communication as a solution for carrying higher amount of information. It corresponded to raised demand on higher network capacity. Better network performance allowed to the operators introduce new types of services and provide existing ones with better quality.

Important role in telecommunications played mobile networks. They allowed people to communicate from anywhere at any time and became an important part of our life. It pushed the industry again to correspond to even more dynamic requirements on network capacity, elasticity and other network parameters.

Today, telecommunication networks are playing huge role in every day life of millions of people. We can't really imagine our life without mobile phones, cloud-based services and applications which makes our life easier, more comfortable and more interactive.

With introduction of new services and raised demand on the network resources, hierarchical network architecture approach became limiting. Enhancing such architecture is also quite costly for the operator, mainly in terms of money investments. Continuous research in the networking field and introduction of new promising concepts play a huge role in pushing the network evolution and shaping the networking future. In order to satisfy customer's demand, telecommunication operators should reconsider the way of using legacy networks with regard to the rising demand on the network parameters.

1.2 Need for automation

Rise of mobile networks, constantly increasing amount of the content and advent of cloud services raised requirements on the network throughput, latency, dynamicity, privacy, security and many other parameters. It pushed the telecommunication industry to reconsider the traditional network architecture [1]. Most of the conventional networks are hierarchical, built in a tree structure with backward compatibility to legacy networks. This design was appropriate to the client-server communication. Nevertheless, when it comes to the dynamic computing, storage needs for data centers enterprises, campuses and carrier environments, this architecture is weak [1]. Key aspects driving the new transport networking evolution include [1]:

- ***Change of traffic patters***, which is mainly lead by following reasons:
  - *Before* communication between the client and the server was simple - client sends the request, server makes some computations and returns data.
  - *Now*, with introduction of cloud solutions, communication between the user and server has been changed - client's request can require distributed computations. So, server makes requests to another servers in order to provide some additional computations, read external data storage(s) and return computed data from different servers before sending reply back to the client. This approach introduces Machine-to-Machine (M2M) communication and raises requirements on the network throughput and network latency.
  - Also, with raise of the mobile networks, user pushes *"change of traffic patterns"* by trying to access the corporate data and applications from any device, anywhere, at any time.
- ***Growth of cloud services*** mainly triggered by popularity between enterprises. Main benefit here consists in outsourcing of some IT tools that company needs. Later on company borrows these tools from Google, Amazon or any other big cloud provider. This demand pushed cloud providers to build an appropriate infrastructure to satisfy tenants. It resulted in increased requirements on elasticity and scalability of the network resources within Data Center interconnections.
- ***Big data*** usually require thousand of parallel computations to process big amount of data. It includes communication of thousands servers at the same time, all of which need direct and highly reliable network connections with each other.

In the end you don't know, when the network will require high throughput and when it wouldn't. Building infrastructure with the margin on high throughput requires a lot of investments, what is not always possible for the telecommunication operator. It's quite costly - keeping the network equipment running, when it's not used. It results in wasting of an electricity, additional costs on maintenance, operations, infrastructure administration and many other costs.

1.3 Software-Defined Networking paradigm

In past several years Software-Defined Networking (SDN) paradigm is gaining momentum. What is it and why it attracts so much attention?

Originally, SDN was introduced by Stanford University as a solution for configuration and control of Ethernet Switches with OpenFlow protocol. Later on SDN principles were extended to multi-layer transport network and evolved in a solution for Data Center

network management system. Now it gains popularity in embedding into the telecommunication networks.

SDN approach consists in separation of the network data plane (packets) and the network control plane (routing process) [1]. Traditional networks are decentralized and very complex to maintain. Basically, it is required to go inside each network element and configure it separately. When the network contains thousands of such elements, this task is becoming quite complex. As a possible solution, SDN attempts to centralize the network intelligence [1] in one entity (usually running on a server), which will simultaneously take care of the network management, network configuration and troubleshooting. This approach creates dynamic, flexible and scalable network with software-based management and configuration [2].

In general, SDN provides an abstraction between network infrastructure and network services. To achieve this, there are several constraints on SDN architecture [1], [2]. It should be:

- ***Directly programmable*** - decouple forwarding functions from data plane;
- ***Agile*** - flow of traffic according to current conditions;
- ***Centrally managed*** - gather all intelligence of the network into one entity through which maintenance, configuration and troubleshooting could be done. Also it tells the network equipment how forwarding plane should handle the network traffic [2];
- ***Programmatically configured*** - enable quick and dynamic configuration of network resources for better performance optimization of the network;
- ***Open standard-based and vendor-neutral*** - unify management and reduce complexity of network configuration.

Based on these pillars, SDN provides a good degree of flexibility and optimization with quick reaction on changing network environment. This could be a good solution for optimizing the network performance "on demand". It helps the telecommunication operator to save costs on the network maintenance [2].

Somehow, main disadvantage of the SDN approach, due to the intelligence centralization, consists in it's elasticity, scalability and security [1]. These issues are still remain for further investigation.

Let's have a look on some of the SDN use-cases [1]–[3]:

- ***Data Centers interconnection*** - software control of data center's interconnections could help raise reliability and quality of offered services.
- ***Software-Defined Wide Area Network (SD-WAN)*** - introduces SDN principles in the network, which covers wide geographical area. Main motivation here is to reduce costs of WAN network, ease maintenance and configuration of the network [2]. A good example of SD-WAN approach could be data center interconnection scenario (described above), where main target is to unify the multiple connections within an enterprise.
- ***Software-Defined Local Area Network (SD-LAN)*** - introduces software-based approach with policy driven architecture in wireless and wired LANs [1]. This solution could be suitable for enterprises.
- ***Software-Defined Mobile Network (SDMN)*** - targets to design mobile networks with protocol-specific features implemented in a software. It maximizes use of software and hardware in core network and Radio Access Network (RAN) [1]. New generation of mobile networks are moving towards this concept.

- ***Link provisioning*** - provisioning of the links (i.e. of transport network) could help to quickly react on changes in the network state. Dynamic analysis of gathered from the network data can help to improve reliability and quality of the connectivity. For example, based on obtained network information you can deploy following services:
  - ***Bandwidth-on-demand*** - control of carrier links gives an opportunity to request additional bandwidth within a network path when it's necessary. It could dynamically increase throughput of the network path. By analogy, bandwidth could be also reduced, when the network path is not used intensively. It could optimize network performance and help to save some costs on maintenance.
  - ***Load Balancing*** - direct reaction on load level of the network can help to equalize the network load balance by rerouting the network flows through the alternative paths. It helps to keep the QoS on the acceptable level. It also requires choice of suitable metric with introduction of the Path Computation Element (PCE) functionality.

In conclusion, SDN allows more dynamic and more programmatically efficient configuration of the network. It can help to improve network performance, optimize resources and make network monitoring easier to perform. SDN basically makes network more cloud-like by introducing software-based approach in it's configuration [1], [2]. SDN is still not fully deployed in real networks, but it stands on the edge of it.

1.4 Network orchestration

When SDN allows dynamic configuration and management of network equipment, the need in solution which can deliver services to the end-user is crucial. This is where the network orchestration could help to achieve this goal.

Network orchestration term covers the ability to program automated network behaviors to support applications and services on top of the network [4]. It is ensured by coordination between the required network equipment and software elements (of the applications). In network orchestration coordination between the software actions and the SDN Controller is crucial. One of the most important elements of the network orchestration is the ability to automate connectivity, based on the network state information obtained from the network monitoring [4].

SDN controller provides you an abstract way to operate with the network device(s), orchestrator deploys network control functions as SDN applications and services on top of it. In order to operate efficiently, you should coordinate hardware equipment and software applications [4], [5]. This is where orchestrators play a big role. To illustrate better this idea, please have a look on Figure 1.1.

As were mentioned above, orchestrator interfaces with the SDN controller and deploys applications based on the network capabilities. Literally, orchestrator tells the SDN controller how it should configure the network in order to deploy and develop certain service to the customer. This interaction is done within certain level of abstraction. For example, orchestrator could send required network parameters. SDN, based on this parameters, would configure the network accordingly. Right now, recognized network orchestrators in industry are ONAP and Kubernetes.

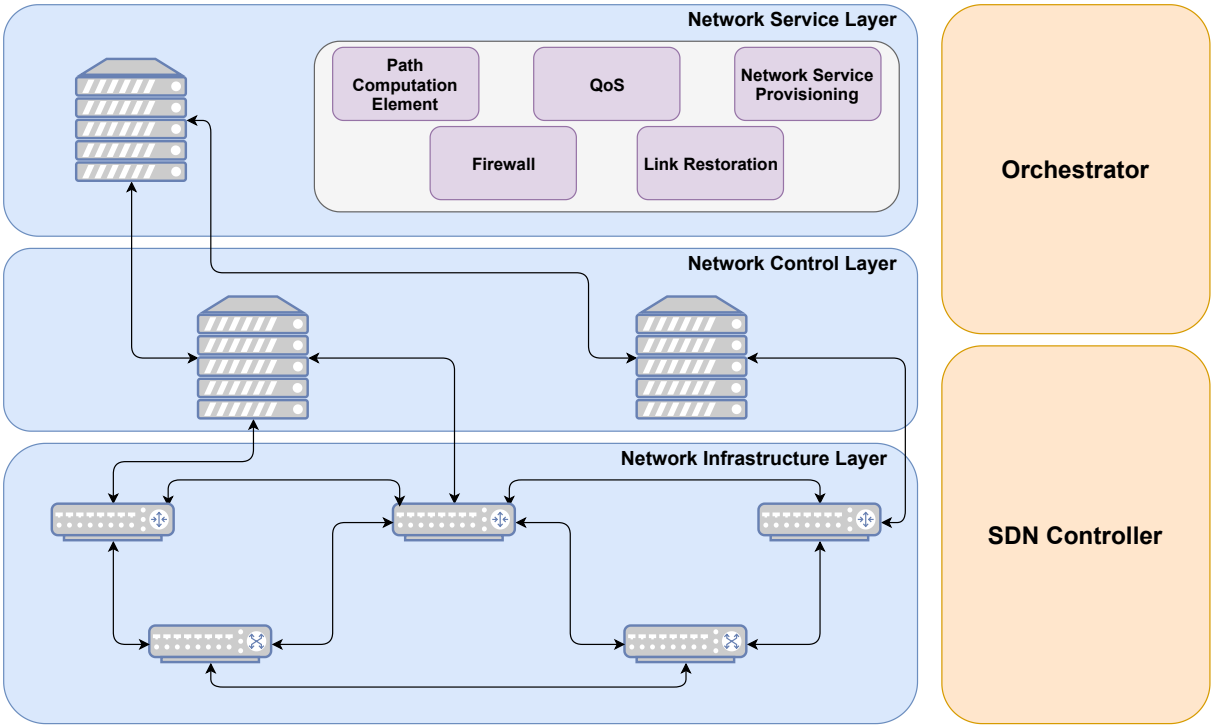

Figure 1.1: Network Orchestration principle

Network orchestration could be applied in following areas [5]:

- Automation of IP-based or OpenFlow-based routing;
- Dynamic security services;
- Traffic engineering based on Path Computation Element (PCE) function, which ensures that workflow follows the correct path in the network;
- Network service provisioning within the workflow path;
- Workflow directioning and management based on obtained from the network information.

There are some initiatives which try to standardize network orchestration. One of them is "institutional" one - Management and Orchestration (MANO) platform based on TOSCA modelling language. It was introduced by ETSI. The other initiative is commercial one. It is lead by AT&T and Orange and resulted into development of an ONAP, the network orchestrator. It allows real-time, policy-driven orchestration and automation of physical and virtual network functions (VNFs) [6]. SDN is one of the ONAP's important components [7].

Summarizing all mentioned before, SDN controller provides tools for network managing. Orchestrator creates services based on these network management tools [4]. Describing management tasks could be complicated and this is where the orchestration steps in the game. It deploys services on top of your network, allows the network to scale as needed and deploy resources as needed. This approach makes the network more agile and responsive.

In the future, network orchestration systems would definitely fill the gap between a wide range of technologies that were enabled by cloud-based network and communication services. For example, between Telecom systems, data-center resources and the customers who are looking to purchase services [4].

1.5 Open Optical Networks

Another trend gaining momentum in past few years is "Open Optical Networks". What does it mean? According to [8], it is an *optical* network which is defined by following milestones:

- ***Open-source software*** developed and contributed by interested parties. It is freely available and shared. Everyone have access to it and everyone can have a look on it.
  - A good example could be an open-source SDN controllers like OpenDaylight and ONOS.
- ***Open-source APIs*** represent a group of APIs defined on an open forum. They are also freely available to everyone in order to implement it into their hardware or software.
  - NETCONF protocol based on OPENCONFIG data models or T-API are good examples of an open API.
- ***Open hardware*** is not yet defined precisely. It is still an open question. Basically, there are two ways to interpret this term:
  - Optical equipment specification set by industrial standards, like Multi-Source Agreement (MSA) projects of AT&T and other participants - OpenROADM. It was designed with equipment inter-operability in mind. This provides more freedom to the telecommunication operators in choosing of a hardware from different vendors. More work is still required to be done there.
  - Hardware "openness" to open APIs. In other words, support of different Open APIs by various vendors. It makes possible to manage proprietary equipment in the same way by third party management system or even SDN controller, since API is known and freely distributed.

Literally, open optical network is an *optical* network driven by open-source technologies and open to multi-vendor cooperation. Why is it beneficial? Main reason consists in the approach to separate various hardware elements (like ROADMs, line cards, transponders and other) and allow "cherry picking" of the best components from the different vendors. This extra degree of freedom gives an opportunity to make the custom built of the network according to the specific requirements of the company.

The telecommunication networks are moving towards multi-vendor environment or, in other words, fully disaggregated systems. This approach provides more flexibility in vendor selection and gives the network operator more advantages in the technology upgrades [8]. What we can say for sure, SDN is going to play a big role in bringing "openness" into the networks. One of the projects, which is targeting this area, is "Open and Disaggregated Transport Networks" (ODTN) lead by ONOS project developer's team [9].

There are still a lot of open questions and barriers which community needs to tackle before it would be possible to apply this concept in practice. Right now, network equipment maintenance, system integration, absence of unified management tools and slow standard development are the main obstacles [8].

Chapter 2

SDN Overview

This chapter provides an overview of a SDN architecture and briefly describes the most commonly used protocols in SDN. Open-source SDN controller solutions as well as custom SDN controllers developed by Google and Cisco are introduced in the second part of this chapter.

2.1 Architecture

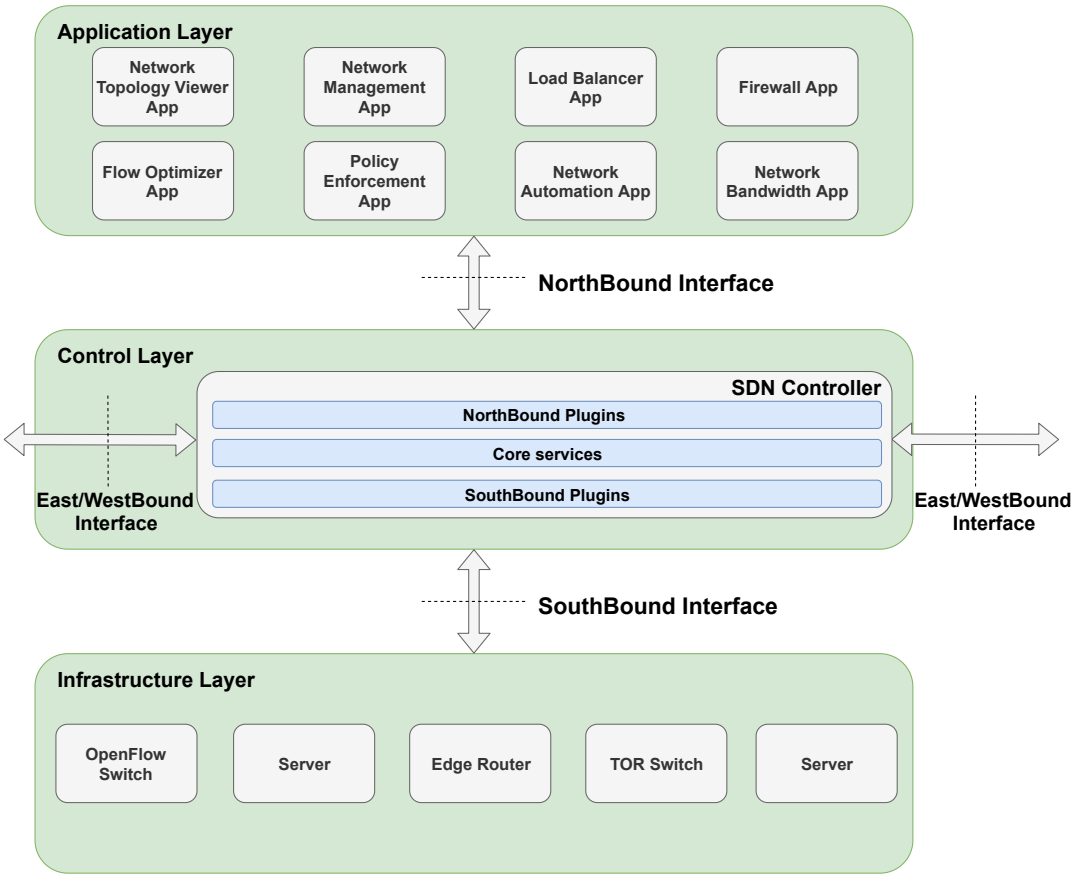

Figure 2.1: SDN Architecture

SDN architecture is represented on Figure 2.1 and could be described with following pillars [1]–[3]:

- ***Application layer*** represents programs or applications that communicate via **NBI** with the SDN controller. They provide a required behavior of the network. Such applications can include Load Balancing, Firewall, Network Topology Overview, Policy Enforcement, Network Bandwidth Optimizing and many more. They concentrate main network intelligence and specify network behavior. Also, such level of abstractions helps to specify and implement network services.

- ***NorthBound Interface (NBI)*** ensures communication between Application and SDN Controller. NBI consists from variety of different APIs/protocols, like REST API or T-API (under development). Literally, NorthBound Interface ensures "understanding" between SDN controller and application specifying network behavior. This interface is implemented in an open, vendor-neutral and interoperable way [1].

- ***Control layer*** is mainly represented by SDN controller or any other Network Operating System (NOS). Provides translation from user requirements into the device-specific format and abstracts the view of network state. Must include several NBI and SBI interfaces.

  - Control layer could also be represented by several SDN controllers communicating with each other through East/WestBound Interface. More on SDN controllers would be described in section 2.3.

- ***SouthBound Interface (SBI)*** ensures communication between SDN controller and Network Equipment. SBI consists from variety of protocols like OpenFlow, NETCONF, RESTCONF, P4 and many more. This interface is also responsible for gathering monitoring data and informing user about immediate changes in the state of the network.

- ***Infrastructure layer*** is represented by various Network Equipments (NEs), like OpenFlow switches, TOR routers or servers, with advertised forwarding and processing capabilities [1]. In SDN, NE is usually covered with following term:

  - ***White Box switch*** is switching and routing hardware providing programmable abstraction from the network functions. Such kind of a switch enables to application on top of the SDN controller specify routing table and tell how to route connections to fulfill appropriate task [10]. In other words, it's programmable hardware (box), where you can specify any behavior you need. White Box switch relies on Operating System which could come from vendor or could be downloaded separately [10].

- ***East/WestBound Interface (EWBI)*** ensures communication between different SDN controllers in distributed networks. It requires standardization and a lot of research to be done.

By defining this pillars, separation between data plane and control plane becomes more clear. Putting out intelligence from the network equipment on the external entity creates an agile network infrastructure, which can be dynamically programmed on certain behavior according to the needs of the customer.

There are two terms which are tightly related to the SDN - Network Function Virtualization (NFV) and Service Function Chaining (SFC). Next two sub-sections are explaining these terms.

2.1.1 Network Function Virtualization

Network Function Virtualization (NFV) complements SDN concept. It decouples hardware and software to enable flexible network deployment and dynamic operations [11]. With NFV hardware-based network services run on a servers as a software and called Virtual Network Functions (VNFs) [12]. Network Function Virtualization aims to accelerate service innovation and provisioning using standard IT virtualization technologies.

For better understanding let's have a look on Figure 2.2, where the NFV architecture is represented.

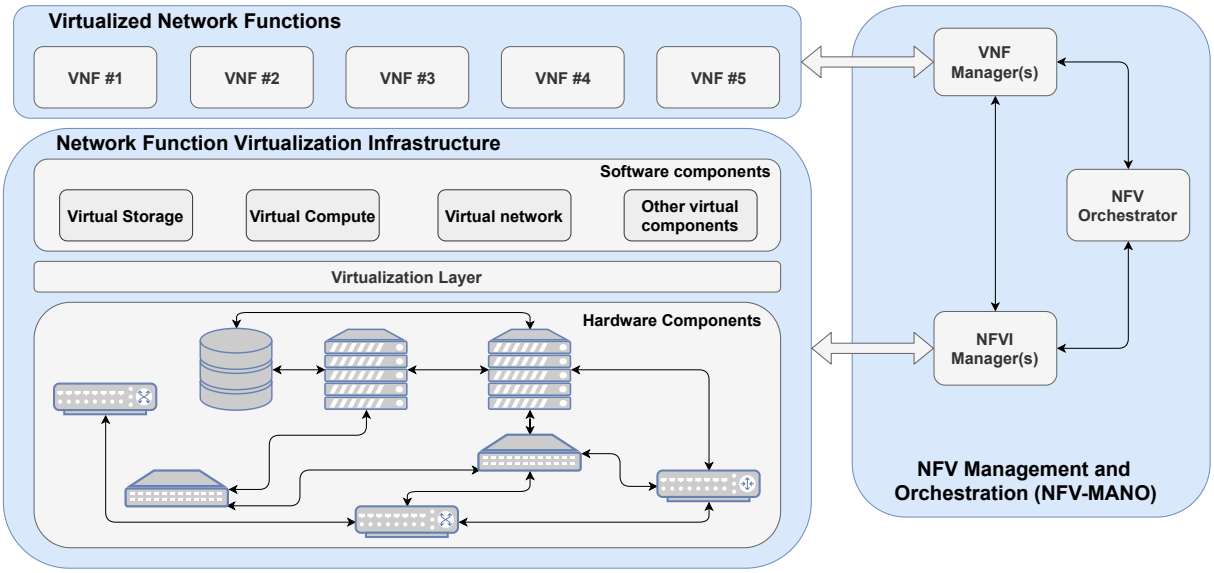

Figure 2.2: NFV Architecture

We can split whole NFV architecture on three main blocks [11], [12]:

- ***Network Function Virtualization Infrastructure (NFVI)*** - defines all hardware and software components that build an environment, where VNFs are deployed. NFVI can spread on several different geographical areas. Physical network interconnections between these areas are part of NFVI.
- ***Virtualized Network Functions (VNFs)*** - implemented software representation of network functions, that NFVI can perform.
- ***Network Function Virtualization Management and Orchestration (NFV-MANO)*** - interface and reference point through which all functional blocks of NFV architecture exchange information. It is done for the purpose of management and orchestration of VNFs and NFVI.

ETSI has already standardized Management And Operations (MANO) framework - an open ecosystem for Network Function Virtualization (NFV) based on automation and orchestration [11]. Virtual Network Functions (VNFs) are interoperable with independently developed management and orchestration systems. Management and orchestration are interoperable within themselves as well [11]. In MANO everything is defined with TOSCA - language which was created to describe components and relationships between cloud-based web services and processes that manage them [13].

As you can see, NFV closely reminds the concept of SDN, but still remains different. SDN came out from separation between data plane and control plane of network equip-

ment. NFV came out from the idea to separate hardware from software. These two concepts are complementary. Their combination, SDN-NFV, provides an ultimate level of agility:

- **SDN** provides control and management of network elements;
- **NFV** provides application agility by using virtualized environment.

2.1.2 Service Function Chaining

Another important term which complements SDN and NFV is Service Function Chaining (SFC). This is another level of abstraction based on top of SDN and NFV. It creates a chain of connected network services and connects them in a virtual chain [14]. The main idea of SFC is represented on Figure 2.3.

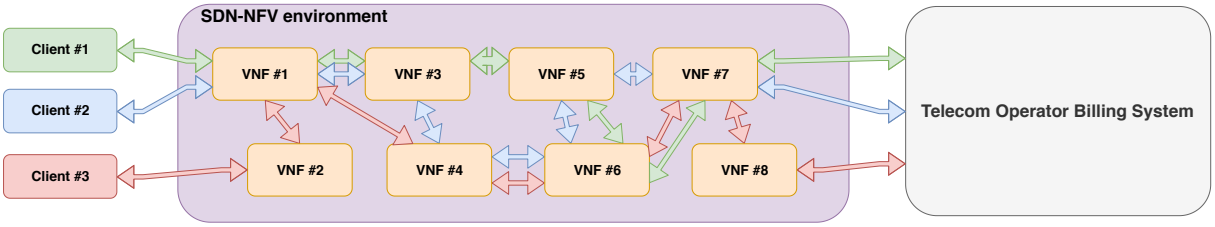

Figure 2.3: Service Function Chaining principle

This capability can be used by network operator in order to setup various "catalogs" of connected services based on a single network connection [14]. Each "catalog" could have different characteristics and later could be sold as a service according to the needs of the customers.

The main advantage of SFC consists in the automation of the way virtual network connections can be set up to handle the traffic for certain service [14]. Another big benefit of this approach is optimization of network resources.

2.2 Protocols

Let's switch to the protocols which allow to embed SDN paradigm into networks. This section provides a brief overview of the most common protocols implemented as a SBI of different SDN controllers.

2.2.1 OpenFlow

OpenFlow is the first protocol which decoupled Control Plane from the Forwarding Plane of the network [15]. It's added value comes from enabling the direct access and manipulation with the forwarding plane of the network device, which could be switch or router (generally OpenFlow switch). That kind of separation allowed more sophisticated traffic management.

OpenFlow protocol enables the network controller to determine the network path of the packet between several (or many) switches [15]. This protocol defines packet matching rule and action, which the device supposed to execute, once the match is found. Controller

installs this rule in a *"Flow Table"* of the device [16]. Once the packet enters the device and matches the rule in a *Flow Table*, the device executes corresponding action [16]. To adopt the protocol for dynamically changed network and reduce complexity, each of the rules in a *Flow Table* has it's own timeout, expiry date. In this way OpenFlow allows to do routing decisions periodically [11].

Packets, which didn't find the match on the switch, are forwarded to the controller. It installs either temporary rule, just for this specific packet, or the "permanent" rule, with the timeout [15]. In any case, the decision is taken on the controller side. Forwarding then is done on the device.

OpenFlow allows SDN controller to push changes to the *Flow Table* in order to control flows for optimal network performance and manage traffic patterns [16]. Controller can decide to modify existing rule on one or more OpenFlow switches. It also helps to avoid the situation when the device asks the controller what to do with each packet entered the switch [15].

In general, each rule installed in the *Flow Table* could be described with following parameters [16]:

- ***Rule*** - defined mainly by matching criteria. You can match:
  - source or destination IP;
  - source or destination MAC;
  - source or destination port;
  - Vlan ID;
  - Ethernet type or specific IP protocol;
  - and many more parameters depending on the OpenFlow protocol version.
- ***Action*** - tells the device what to do. It could be:
  - Forward packet to the specific port;
  - Encapsulate packet and forward it to the controller;
  - Drop the packet;
  - Set the packet to normal processing pipeline (usually default rule).

OpenFlow protocol provides high level of flexibility in network management. Device functions are no more dependent on the specific hardware. OpenFlow protocol, by providing various set of rules, makes the OpenFlow switch very versatile device. Basically, you could compose from the device a firewall, load balancer, put any other functionality on demand, or even put them all just in one box. Ability to run the device with a necessary functionality and possibility of dynamic reconfiguration according to the specific network needs is the main advantage of OpenFlow.

Somehow, there are already plenty of versions of the OpenFlow. Any new protocol support requires upgrading the existing OpenFlow protocol to a newer version. It makes the protocol more complex because of the variety of different matching rules. Also, once any new packet is coming, device always asks the server what to do with a packet. Decisions are not taken on the device side anymore. It adds some complexity to the communication process and can dramatically increase latency in the worst case scenario (device asks server what to do with every packet). This is the main disadvantage of the OpenFlow protocol. Most probably, in the near future OpenFlow would be substituted with a P4 protocol (described in 2.2.2).

2.2.2 P4

P4, or Programming Protocol-independent Packet Processor, is a language which describes how packets should be processed by forwarding element [17], [18]. P4 specifies only the data plane forwarding functionality of the target device. Specified forwarding behavior is then converted by P4 compiler into the data needed for control plane and data plane to communicate [17]. P4 doesn't specify the behavior of control plane of the device [17].

As the language targets on protocol independence, there are some constraints on it's design [18]:

- ***Target independence*** - P4 programs are designed to be implementation-independent, so they can be compiled on any device.
  - Each device is called **P4 target**.
  - Each target must provide a **P4 compiler**, which maps the P4 source code into the device-specific model. Compiler could be embedded into the device or run as an external software or as a cloud-based service.
- ***Protocol independence*** - achieved by **not** enabling native support for protocols as Ethernet, TCP/IP or any other. Instead of that, P4 describes the header format and/or the field names of the required protocols. This information is interpreted and processed by the target compiler.
- ***Reconfigurability*** - P4 targets should be able to dynamically change the way they process the packets.

With regard to the constraints above, P4 program specifying forwarding behaviour should have following components [18]:

- ***Parsing logic*** - forwarding rule for specified custom packets, not only limited on TCP/IP or Ethernet.
- ***Headers*** - description of the packet format with name of the fields within the packet. To provide required protocol independence custom fields of random length are allowed.
- ***Parser*** - finite state machine, which extracts required header bits according to the specification provided through the P4 program.
- ***Generic match action table*** - set of user-defined match-action tables. User can add a match-action through the control plane.
  - ***Match-action pipeline*** - packet forwarding process can be broken into several actions, where each action corresponds to the table lookup or specific header manipulation.
  - ***Tables*** - contain state (matching criteria + action) used to forward the packets.
  - ***Actions*** - P4-specific description of required manipulations with the packet.

In order to process the packet, P4 determines relative sequence of the tables and allows conditional execution of each table based on if/else statements [18].

In comparison with OpenFlow, P4 is concentrating only on data plane layer, without any interaction with control plane layer. P4 language introduces ultimate level of agility and provides great flexibility by adding new features and removing unnecessary and unused protocols [19],

2.2.3 Open vSwitch Database Management Protocol

The Open vSwitch Database Management Protocol (OVSDB) is a management protocol in a SDN environment. It was specified in RFC 7047 [20] several years ago.

OVSDB was originally created as a part of the open-source software Open vSwitch (OVS), the virtual switch for Linux-based hypervisors with a lot of various features [21]. Main focus of OVS was to create a modern programmatic management protocol. This attempt resulted in an OVSDB as a solution.

A lot of people think that OVSDB is quite similar to OpenFlow, but it isn't. OpenFlow allows to manage flows, or forwarding rules, when OVSDB is a solution for device configuration. Open vSwitch Database Management Protocol allows to configure ports, bridges, create/delete interfaces on the device [21].

OVSDB functionality could be described with following milestones [22]:

- Each device with an OVSDB support has an OVSDB database schema. It specifies device configuration information. This database contains control and statistical information.
- Information is stored in a different tables inside the database. It can contain, for example, learning information about MAC layer and many others,
- Entity, communicating with an OVSDB device, monitors the state of each table in a database and can add, delete or modify these information according to the user's needs.

Unfortunately, right now exist very few management platforms (SDN controllers) with support of OVS or OVSDB in particular. OVSDB is now supported by more vendors, than OVS itself [21], [22].

2.2.4 Network Configuration Protocol

Since in this work we mainly use NETCONF protocol, it will be described in details within this section. Some prerequisites are required for better understanding of the NETCONF protocol. Following two sub-sections will provide a brief overview of YANG data modeling language and vendor-neutral data models developed by OpenConfig working group, which are tightly aligned with the Network Configuration Protocol.

2.2.4.1 YANG Data Modeling language

Yet Another Next Generation (YANG) is a data modeling language used to model configuration and state data, which are being manipulated during the NETCONF communication. It is defined in RFC 6020 [23], where all exact details of the language are explained. With YANG data modeling language, it is possible to describe various interfaces of particular network equipment. For example, in YANG representation simple interface of the switch can look like [24], [25]:

```
list interface {
  key “name”;
  leaf name {
    type string;
    description “Interface’s name”;
    }
  }
  leaf type {
    type string;
    description “Type of the interface”;
    }
  leaf speed {
    type string;
    units ”Mb/s”;
    description “Supported speed”;
    }
```

Later on it could be translated into XML format:

```
<interface>
  <name>eth0</name>
  <type>Ethernet</type>
  <speed>40</speed>
</interface>
<interface>
  <name>eth1</name>
  <type>Optical Ethernet</type>
  <speed>100</speed>
</interface>
```

Such statements combined in a big structure can result into definition of a module representing the whole device or specific aspect of it’s functionality.

YANG data modeling language also provides some important features, which is good to mention.

- YANG can model state data as well as the configuration data based on a config statement. When a node is tagged with *config* ***false***, it’s hierarchy flagged as a state data and returned with NETCONF ***get*** message. Otherwise, if a node is tagged with *config* ***true***, it’s hierarchy flagged as a configuration data and returned with NETCONF ***get-config*** message [24].

- YANG has a set of built-in types, which are similar to the ones used in many programming languages [24]. Nevertheless, it has some specific differences due to the requirements of the management domain. As an extension, you can define derived types from the base ones [24]. A base type can either be a built-in type or another derived type [24], [25].

- It is possible to extend modules by insertion of additional vendor-specific parameters (nodes) into the data model. You should define location in the data model and conditions when the vendor-specific module is valid [24]. It is probably the most important feature of YANG.

Flexibility provided by YANG data modeling language turns out in a high integrity into NETCONF protocol. For example, NETCONF RPCs are based on YANG data modules. [24]

2.2.4.2 OPENCONFIG

OpenConfig is a collaborative effort in the networking industry which aims to move towards more programmable and dynamic multi-vendor network configuration. It tries to adopt software-defined networking principles such as declarative configuration and model-driven management and operations [26]–[29]. OpenConfig supports vendor-neutral data models to configure and manage the network [29]. These data models define configuration and operational state of the network equipment for the most common protocols or services [26]. The main goal of the OpenConfig, as an operators initiative, is to use the same set of standards in order to configure the network devices from the multiple vendors [26], [27], [29]. Data models are written in YANG data modeling language, which is briefly described in the section 2.2.4.1 above.

OpenConfig modules define a data model through its data. Each model is uniquely identified by namespace URL to avoid possible conflicts. OpenConfig working group is leveraging YANG to model not only configuration data, but also telemetry information coming from the device [28]. The Figure 2.4 specifies all data models produced by OpenConfig working group on 15th of May 2019.

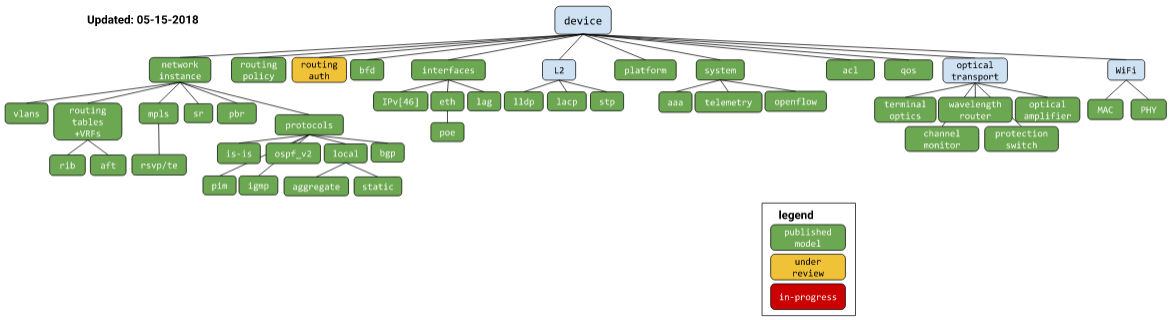

Figure 2.4: OpenConfig data model tree [30]

If you wish to have a look on some of the OpenConfig data models which must be implemented by optical devices in order to support NETCONF protocol, please refer to Appendix A.

OpenConfig provides already quite detailed and highly-developed data model for the optical transport networks. There is also another competitor, OpenROADM, which targets to specify even more detailed data models for the transport optical networks.

To conclude, OpenConfig is made up for big network operators, which are shaping the direction for the developed data models. OpenConfig standard, by it's nature, targets to cover transport network and provides data models for Border Gateway Protocol (BGP), Multiprotocol Label Switching (MPLS) and many more protocols used in big transport networks. When it goes to the optical equipment, especially in transport network, we always abstract from the protocols and go on the lowest possible layer extending the OSI on L0, photonic layer. At this point we are pushed to describe physical properties of the optical channels, what brings a bit different understanding and abstraction of the network.

However, OpenConfig working group activity is not about developing a standard data model for every feature the network equipment supports [28]. It's a great first step towards the open network automation. It provides consistency across all vendors and ensures that you are not using any vendor-proprietary features [28]. A consistent predictable model across the network devices is where the whole magic is [28].

2.2.4.3 NETCONF protocol

Network Configuration Protocol (NETCONF) was created with purpose to unify management and configuration of the network equipment. In order to be able to do this, network device should implement a database, where it stores all it's configuration. There are four types of databases [31]:

- ***Running*** – stores current configuration of the device during it's running;
- ***Candidate*** – stores sample configuration of the device, which could later replace the actual configuration of the device;
- ***Startup*** – stores default configuration of the device to run on start-up;
- ***Writable-running*** – basically the same as *"Running"* database. It is being implemented as the only device database or if the device doesn't support *"Candidate"* type of database.

Configuration information is passed in one of the formats – OpenConfig or OpenROADM. In simple words, both of these standards specify sets of the parameters which are necessary to pass in order to configure the device. Main difference between them is in their naitre. OpenROADM targets on any optical network, including transport one. OpenConfig targets on transport network in general. In the optical network domain, OpenROADM describes more in details optical networks than OpenConfig.

Once all parameters are collected and composed to a NETCONF message, controller is ready to send the configuration request to the device. Process of the device configuration could be done in several ways:

- You're directly changing *"Writeable-running"* database.
- You're storing changes inside *"Candidate"* database, validating them and then copying them to the *"Running"* database.

One of the necessary things to do is to lock your target database before writing changes to it [31]. Locking/Unlocking process is implemented in order to prevent writing to the same database from several NETCONF sessions at the same time. Once you wrote your configuration, don't forget to unlock target database.

In the beginning of each NETCONF session, parties should exchange a list of supported capabilities with each other. This feature was implemented in order to ensure that the communicating parties will understand each other. It is one of the most important things in the NETCONF protocol. Here is the list of capabilities which could be exchanged during the NETCONF session establishment with their brief description [31]:

- *:base:1.0* – indicates support of NETCONF v1.0;
- *:base:1.1* – indicates support of NETCONF v1.1;
- *:writable-running* – this capability is enabled by default. If the candidate data storage is used, this capability should be disabled; [32]
- *:candidate* – enables to store configuration which will then replace *"running"* configuration. Could be implemented either as an external or as a built-in database. This capability is also enabled by default. If the candidate data storage is not used, this capability should be disabled; [32], [33]
- *:startup* – indicates ability to store the configuration which is then will be loaded as a default in *"running"* mode in case of an unexpected device shutdown or a reboot. Disabled by default; [32], [33]
- *:validate* – indicates ability of the network equipment provide a semantic validation of the stored configuration. Constraints should be specified in the YANG data model. Validates complete database, not particular <*edit-config*> requests; [33]
- *:confirmed-commit* – this mode requires a server to send two commit *RPC* requests instead of one, to save any changes in the *"running"* database. If the second request does not arrive within a certain interval, the server will automatically revert the running configuration to the previous version; [32], [33]
- *:rollback-on-error* – allows the client to set the <*error-option*> parameter to *rollback-on-error* (other permitted values are stop-on-error and continue-on-error). What is taken as an *"error"* should be defined by the data model. If any error occurs during the edit operation, target database will not be affected. [32], [33]
- *:url* – indicates which schemes (file, https, sftp) the server supports within a particular URL value.
- *:notification* – indicates, if the server supports the basic notification delivery mechanism defined in RFC 5277 [34] (<*create-subscription*> will be accepted by the server).
- *:interleave* – the server will accept <*rpc*> requests while notification delivery is active. Otherwise, client can't send <*rpc*> request during the notification subscription. The *:notification* capability must be present as well, if this capability is advertised.
- *:partial-lock* – indicates, if the server supports <*partial-lock*> and <*partial-unlock*> operations defined in RFC 5717 [35]. It allows to each of multiple independent clients write to a different part of the <*running*> configuration data store at the same time.
- *:xpath* – the server fully supports the W3C XPath 1.0 specification [36] for filtered retrieval of configuration and other database contents. Type attribute within the <*filter*> parameter may be set to the *'xpath'*. Also, server may partially support XPath retrieval filtering, it can't advertise the *:xpath* capability.

All NETCONF operations are carried out within a session, which is attached to the transport layer connection. Protocol doesn't have standard security model yet. Each session itself is wrapped inside the SSH connection with a statically assigned port 830. SSH provides encryption and security constraint into the communication. NETCONF is a session-based network management protocol, which uses XML-encoded Remote Procedure Call (RPC) with configuration data to manage network devices. [33]

NETCONF communication is Client-Server based and simple by itself. Right in the beginning of the NETCONF session, Client and Server should exchange *<hello>* messages advertising their capabilities. In order to communicate correctly, set of the exchanged capabilities should be the same. Once it's done and the communication is established, Client can send *<rpc>* requests in order to retrieve the data and configure the device. Server's request queue is serialized, so requests will be processed in order of their reception. Beginning of the NETCONF communication is shown on the Figure 2.5:

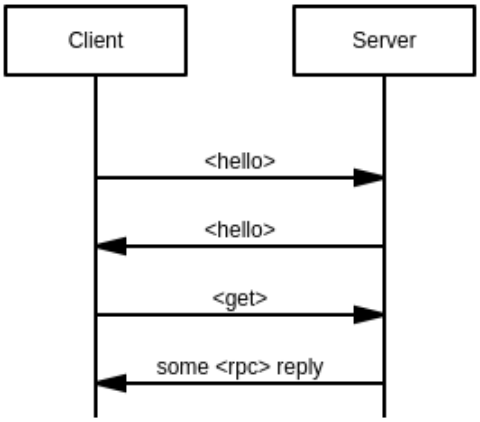

Figure 2.5: Beginning of the NETCONF communication

Most of the operations are designed to select one or two specific configuration databases. Let's have a look on some of them:

- ***<get>*** - retrieves data from the running configuration database and/or device statistics.
- ***<get-config>*** - retrieves data only from the running configuration database.
- ***<edit-config>*** - allows to provide modifications to the configuration database.
- ***<copy-config>*** - copies a configuration database. Device should support at least two different types of databases. Source and destination databases could be specified.
- ***<delete-config>*** - deletes chosen configuration database (should be specified).
- ***<commit>*** - commits (basically copies) the contents of the <candidate/> configuration database to the <running/> configuration database.
- ***<discard-changes>*** - clears all changes from the <candidate/> database and makes it (the <candidate/> database) match the current <running/> configuration database.
- ***<validate>*** - checks the entire content of whole configuration database on semantic correctness. Target database for checking could be specified.
- ***<lock>*** - locks chosen configuration database and allows only to current session apply changes to it.
- ***<unlock>*** - unlocks configuration database and allows another users to apply changes in it.
- ***<create-subscription>*** - creates a NETCONF notification subscription. For example, telemetry data could be reported through this mechanism.
- ***<close-session>*** - terminates the current session

- ***<kill-session>*** - terminates another session, the different one from the session, which invoke this <rpc> request. [33]

Extending the NETCONF Server [32]

All enabled NETCONF capabilities are advertised within the initial *<hello>* message, which Server sends to the Client right in the beginning of the communication. Nevertheless, to indicate device capabilities (what the device can actually perform), supported YANG modules are exchanged via *<hello>* message as well [32]. This allows us to extend the NETCONF protocol on custom *<rpc>* operations defined separately from the standard ones. This feature provides a higher degree of flexibility and versatility of the protocol and allows to the vendors provide specific *<rpc>* requests for more precise device tuning.

2.2.5 RESTCONF

RESTCONF protocol is specified in RFC 8040 [37]. It implements some of the NETCONF functionality on top of the HTTP/HTTPS connection [38]. RESTCONF, as well as a NETCONF, was developed with purpose to manage the device in a standard way.

Since RESTCONF protocol is quite similar to the NETCONF protocol, it also implements different types of data storage. There are currently two types [39]:

- ***Operational*** - contains data inserted via the network.
- ***Config*** - contains data inserted via controller.

Each data store is defined by set of device-specific resources written in YANG data modeling language. Each resource defines it's own content and notification events [39]. All RESTCONF content splits to one of the following categories [39]:

- *data* resource,
- *operation* resource,
- *event stream* resource.

Network equipment running a RESTCONF agent could be manipulated with following HTTP commands [38]:

- ***GET*** - retrieves data from specific resource. Supports all types of the resources, except *"operation"* one.
- ***PATCH*** - partially modifies resource information. Similar to the NETCONF *<merge>* request.
- ***PUT*** - creates or replaces the target resource.
- ***POST*** - creates a data resource or invokes an operation resource.
- ***DELETE*** - deletes the target resource.

RESTCONF protocol is quite close to the NETCONF by it's functionality, but still missing some crucial functions [38] like multiple datastores, locking of a datastore, rollback function and many others. RESTCONF protocol is not intended to replace a NETCONF protocol. It targets to provide simplified interface with REST principles that follows device abstraction through a data model [39].

2.2.6 gRPC

Google RPC, or gRPC, is a high performance RPC framework using Protocol Buffers (Protobuf) and HTTPv2 as a transport protocol. Protocol Buffer is a platform-neutral and language-agnostic serialization format introduced by Google. Each message is a small logical record of information containing a series of name-value pairs represented in a JSON-like format [40].

Since gRPC is based on HTTPv2, it exploits many of it's features, for example [41]:

- *Persistent TCP session* - everything is done within one TCP session (you don't need to re-open it again);
- *Compressing headers*;
- *Cancellation and timeout contracts* between client and server;
- *Flow control on data frames* - now client or server can set their own values for flow control. It adds extra degree of complexity.

In gRPC all communication is Client-Server based [40], where device is always a Server. We can differentiate two types of the gRPC communication [41]:

- ***Unary*** - synchronous communication. Client sends one request and waits until Server's respond. Once the answer is obtained, Client sends next request.
- ***Streaming*** - could be accomplished in three different configurations:
  - Client pushing messages to a stream;
  - Server pushing messages to a stream;
  - Client and Server both send data into a (bidirectional) stream.

  In all streaming cases Client initiates RPC method [42]. There is no acknowledgement until stream is completed, what adds complexity when you need to debug failure of one of the nodes. Somehow, it could be solved with bidirectional stream, where Server streams acknowledgements.

Great advantage of gRPC consists in compatibility provided by Protocol Buffer (Protobuf). Protobuf is an Interface Definition Language for describing the service interface and the structure of the payload messages [42]. It is also a serialization format for the sent data. Protobuf allows quick encoding and decoding procedure, refuses zero-copying of data and instead chooses which data to encode and which to decode. This makes the data smaller from the encoding/decoding prospective, usage of CPU. Because of that, the gRPC protocol is quite fast [41].

Protobuf allows to define *"Schema(s)"*, where you specify semantics of your object and basically tell which functionalities does the device support. You can specify there which field of the packet should be validated [40]. Once you introduce new *Schema* in the communication, entities, which don't implement it, can easily parse this new feature further without even inspecting it [40].

Protocol Buffer defines how to interpret messages and allows the developer to create stubs making encoding and decoding process quicker [41]. You can freely add or remove stubs. Name of the field inside the stub should be different. It ensures backward compatibility.

Protobuf looks like:

```
message Foo {
  string name = 1
  int32 age = 24
}
```

There are some constraints on Protobuf structure:

- Every Protobuf encoder/decoder should be able to skip the fields it doesn't know
- Every Protobuf encoder/decoder should be able to set default values for fields it can't find.
- Any field in Protobuf should start from the wire type to define how the message should be decoded.
  - Decoding strategy could vary with regard to the field type.

In conclusion, gRPC[1] is quite robust and quick protocol. It is backward compatible and supports code generating feature. Also, you can change out any encoding method to the one you like. Protocol buffers offers a great speed advantages in encoding, decoding and size of the data transferred over the wire [40].

2.2.7 Protocol comparison

Comparison of the protocols described in previous sub-sections is depicted in the Table 2.1, where we tried to summarize all benefits and drawbacks of each them.

[1] gNOI and gNMI protocols are carried by gRPC protocol.

Protocol	Device is	Body of usage	Benefits	Drawbacks
OpenFlow	Client	Device forwarding function	Separation of control and data plane layer, Device could execute whatever network functionality you want	Centralized intelligence (interaction with control layer), Device makes request to the central entity in order to make a decision, Sophisticated and complex to implement, Must be extended with any new protocol support
P4[2]	Server	Device forwarding function	Protocol-independent, Highly agile and flexible, Device could execute any function you wish	Each device must run P4 compiler (could be heavy on resources)
OVSDB	Server	Device configuration	Enables precise and agile configuration of the device	Few management platforms support this protocol
NETCONF	Server	Device configuration	Standardized management based on unified data models, Flexible in configuration	XML representation, RPC calls could differ dependently on particular Data Model implementation
RESTCONF	Server	Device configuration	Standardized management based on unified data models, Relatively easy to implement, Runs over HTTP	Less functionality and flexibility than in NETCONF
gRPC	Server	Device configuration	Fast, Language-agnostic, High degree of compatibility, Secure and Robust	Each implementation is specific one, Schema names should be tracked

Table 2.1: Protocol comparison

[2]It is **not** correct to compare P4 with other protocols, since it's not yet fully deployed in industry. Following comparison is just author's subjective opinion.

2.3 SDN Controllers

The most popular SDN controller solutions are described within this section. There are a lot more SDN controllers, but some of them are already out of date and not maintained anymore. Because of that Author focuses on actual state-of-the-art of SDN controllers. In main focus are solutions, on which major industry players are betting.

2.3.1 Open-source solutions

Open-source solutions are not (yet) widely used in real networks, but they are certainly almost ready to be put in practice. In the following sub-sections we will take a look on the most famous controllers - OpenDaylight (ODL) and Open Network Operating System (ONOS) - which are being developed on regular basis for already a long time.

2.3.1.1 ONOS

Open Network Operating System (ONOS) is an open-source controller written in JAVA and initially developed by Open Networking Lab (ONLab) which was later merged with Open Network Foundation (ONF). This controller was initially designed for scalability and high-availability [43] targeting to become a solution for Wide Area Networks (WANs) and Service Provider Networks.

Before diving deeper into the ONOS architecture, let's have a look on interesting "know-how" introduced during the development of ONOS - ***Intent-Based Networking*** [43], [44]. Intent Framework of ONOS [44] allows application just to specify it's desire, so there is no need to tell precisely which parameters to set up on the device side. In other words, you're specifying "What do you want to do" instead of "How to do". Such policy-based directive is called an ***intent***. This functionality could be achieved by introducing special Intent Compiler which translates the desire into device-specific information. Main goal of such an approach was to shift focus from the network details level to the network application level.

Since ONOS is positioning as a scalable platform, it introduces distributed architecture in order to scale with increasing number of the devices [43].

2.3.1.1.1 Architecture

Let's have a closer look at the ONOS subsystem infrastructure.

As you can see on the Figure 2.6, we can logically split ONOS on four layers [43]:

- (Network) Applications (L2 Forwarding, Learning switch and etc.);
- Northbound API (REST API, Web GUI, Cli);
- Distributed Core Services (Device Drivers, Intents, Flow Rule and others);
- Southbound Protocols (APIs) (OpenFlow, NETCONF, OVSDB, etc.).

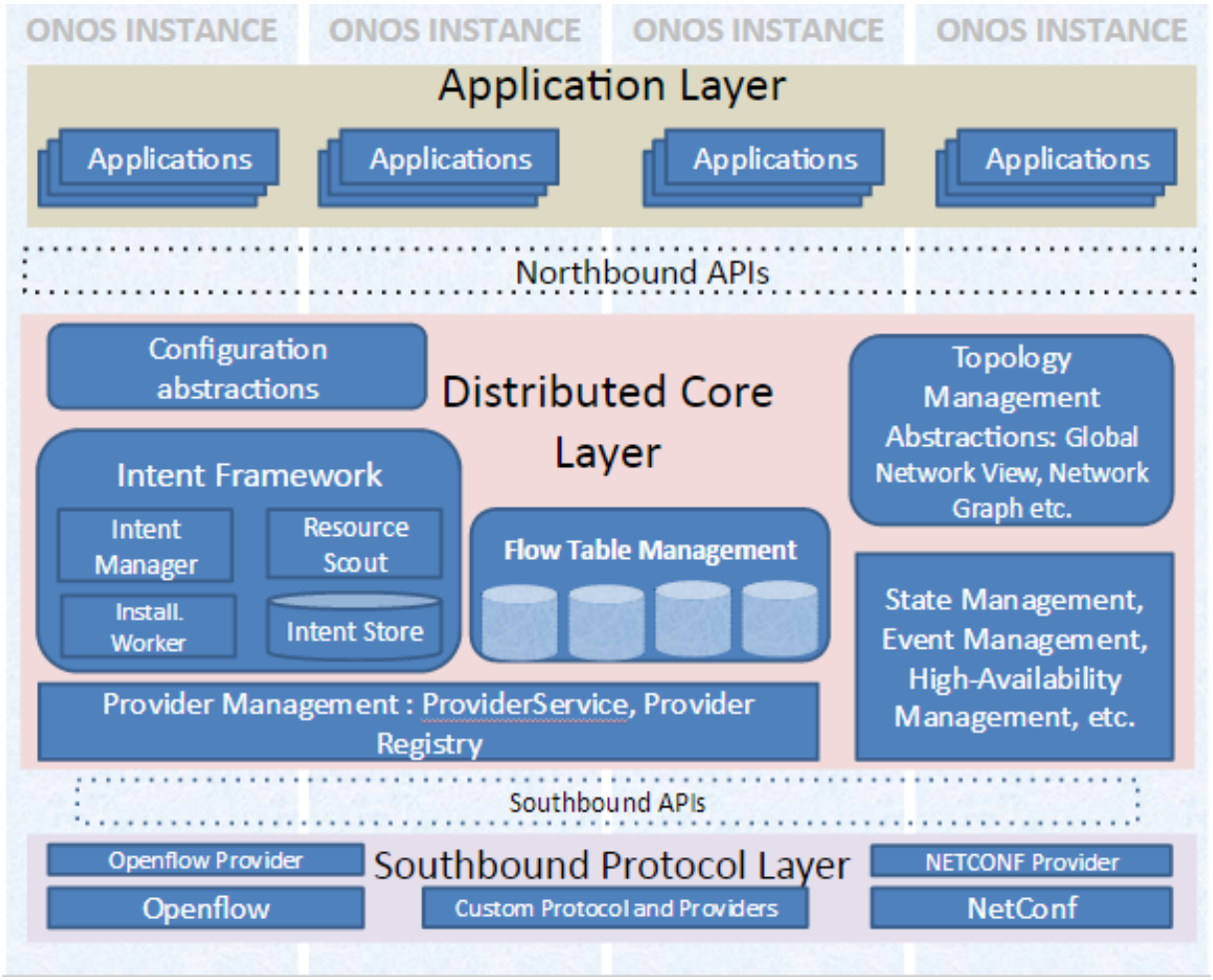

Figure 2.6: ONOS Architecture [43]

Application layer represents main "intelligence" of the ONOS. In general, applications implement network functionality. They are responsible for routing, switching and more sophisticated services which could be built on top of the ONOS controller. Every application is based on top of the microservices provided by the controller's distributed core.

Northbound API provides us a lot of freedom in interacting with ONOS core. You can configure controller through the REST API or using the Transport API[3] [45], [46], you can just simply use embedded in ONOS Web GUI to manage your network or simply use the command line. As an external option, you can use python programs stored in the directory *"tools/test/scenarios/bin"* of the ONOS controller to enable some functionality.

Distributed Core Services has microservice structure. It is divided on different subsystems which interact with each other. When one of the services crashes, it doesn't lead to the crush of the whole controller. Different microservices could be independently restarted. It provides quite agile core structure.

Main task of the ONOS core is to receive information from the network devices and provide it to the applications running on top of the network. This is one of the reasons, why ONOS core implements a lot of different microservices (which are built based on

[3]Conceptually Transport API looks similar to the NETCONF protocol. It also uses XML-like format to parse the network data. You can see content of Transport API by invoking *"odtn-show-tapi-context"* from the client console of the ONOS. It will parse you all information about connectivity services, devices, links and different topologies registered in the controller.

Object-Oriented Programming philosophy). *Intent Framework* is a part of the the ONOS distributed core.

Southbound API is represented by many different protocols. ONOS Southbound Interface implements such protocols as OpenFlow, P4, OVSDB, NETCONF, RESTCONF and gRPC. They ensure communication and data exchanging (like statistics, device information and device configuration) between the network device and the ONOS core. Every Southbound protocol in ONOS implements it's own provider in order to unify and simplify interaction with the network device [43]. Main purpose of the "provider" is to provide necessary description of the network device. Device subsystems supports multiple providers [43].

One of the magnifique features of ONOS is that it could be deployed in several separate instances which coordinate with each other [43]. With this feature we can achieve resiliency, fault-tolerance and better load-balancing management [43].

ONOS is essentially an OSGi-compliant framework [43] used for binding together developed[4] applications and microservices. OSGi is a Java framework which enables development of modular software programs. Each microservice (bundle) is represented by set of JAVA classes packed into a jar file and could be dynamically reloaded on demand [47].

ONOS uses Karaf as an OSGi framework implementation. Whole controller runs in a Karaf container. ONOS developer's team provide very well-written documentation and plenty of tutorials on ONOS basics and how to implement certain functionality.

2.3.1.2 OpenDaylight

OpenDaylight (ODL) is another open-source controller written in JAVA and based on the OSGi architecture as well. It is the most advanced controller in terms of *"out of the box"* functionality. It focuses on building multi-vendor, multi-project ecosystem to drive innovation and open network approach towards SDN [48].

ODL is developed and maintained by the Linux Foundation. ODL's projects are focused on adding specific features to the controller [48]. Similar to other SDN controllers, ODL supports network programmability through the various Southbound protocols and different network services. It also has Northbound Interface and set of applications performing wide variety of network functionality.

2.3.1.2.1 Architecture

Let's now have a closer look on OpenDaylight architecture. It is depicted on Figure 2.7.

In OpenDaylight ecosystem controller acts as a middleware connecting together applications and protocols talking to the network devices [48]. Controller allows to application to be agnostic to the device-specific things. Such an approach allows developers to concentrate more on the application itself, rather than think about how to tell the device what to do.

Southbound Interface (SBI) of OpenDaylight supports various communication protocols, like OpenFlow, BGP, SNMP and many others. Each protocol is represented by

[4]In past applications used to be developed with Maven. Now development of any functionality, including applications, is done mainly with Bazel. Maven is still supported, but not recommended to use.

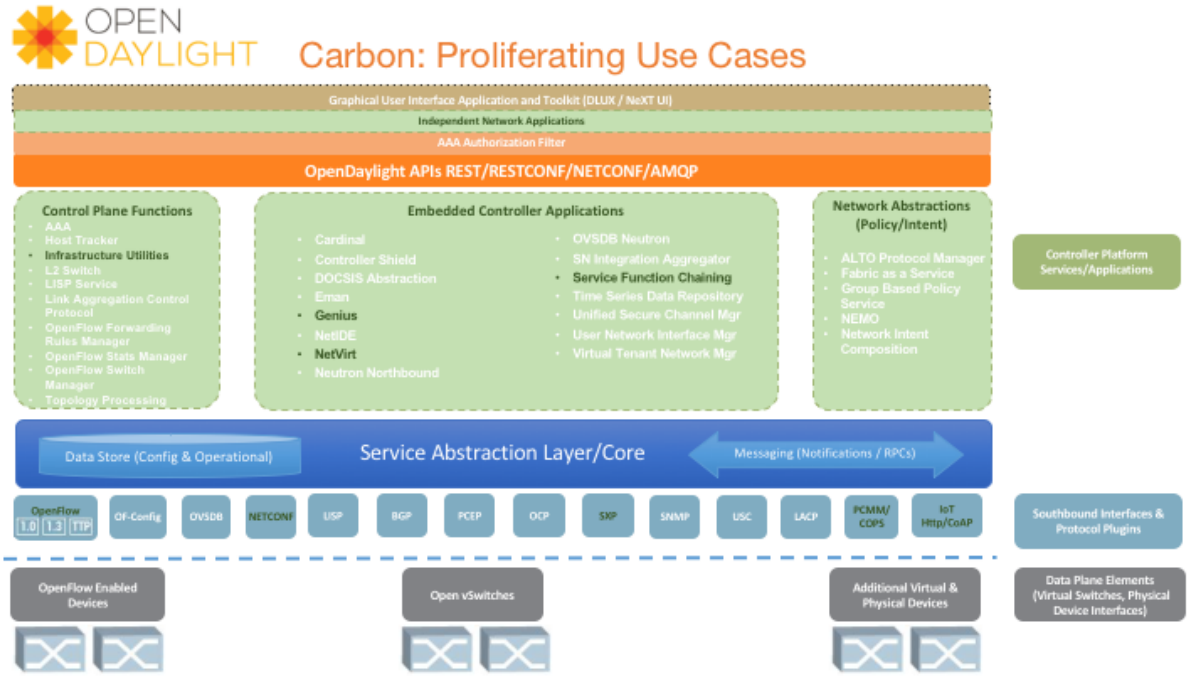

Figure 2.7: OpenDaylight Architecture. Carbon release [49]

it's own module, which is linked to the Service Abstraction Layer (SAL). SAL determines how to fulfill application request with respond to underlying protocols and devices.

Service Abstraction Layer (SAL) is the key aspect in ODL design. It provides abstraction of services between the service consumers (applications) and producers (network devices). Basically SAL acts as an entity registering and interconnecting different services within the core. There are two ways of implementing this entity [48]:

- ***Application-Driven SAL (AD-SAL)*** focuses on providing abstraction from the device to the application developer. It allows the developer to concentrate on the application logic, rather than focusing on how the device behaves.
  - The primary problem on which AD-SAL concentrates is providing set of universal APIs with support of all device functions. Device talks to the SDN controller via driver-specific modules. These protocol plugins communicate with SAL layer exposed API [48]. Basically, SAL converts device-specific functionality into the set of API plugins and allows any application to use these APIs.
- ***Model-Driven SAL (MD-SAL)*** goes a step further (than AD-SAL) and allows the developer to work with service-agnostic interfaces, which are provided by "service modules" (i.e. any protocol plugin). The key difference with AD-SAL lies in a way how these plugins are used by different providers and consumers.
  - Providers (generally southbound plugins) create the model of the service they use and store it in a YANG format. After that, YANG compiler creates uniform API for the consumer and become part of the plugin. It provides a very high level of uniformity between different plugins in terms of definition and usage. It also complicates debugging process, cause you can't modify the generated code.

The controller platform itself has various network service functions included - service

for topology discovery, forwarding manager, switching manager and many more [48]. It also contains vendor components needed to interact with the underlying network devices in specific format. Network service functions and vendor components are interconnected within the SAL core.

Northbound Interface (NBI) interconnects applications and SAL core. NBI supports OSGi framework (Apache Karaf) and bidirectional REST API [48]. OSGi is used by the application that is being run in the same environment as a SDN controller, REST API is used by the application that is running in the external environment [48].

Applications, the top layer of ODL, concentrate main network intelligence. Most of the applications are directly mapped to the appropriate services of the SAL core. These apps could also be used as a way to orchestrate the network (i.e. Load Balancing).

In the beginning of it's path, ODL was the most documented SDN controller, but with the time, due to the rapid frequency of new releases, the documentation became outdated and doesn't precisely describe it's actual state.

2.3.2 Custom solutions

With the rise of open-source SDN solutions, some big companies made their own distributions of SDN controller. Generally, it is classic open-source SDN controller enriched with adaptors and functionality of their own devices. For example, ***Cisco*** provides their own distribution of ODL controller [50]. It leaves all functionality of ODL needed for communication with Cisco network equipment.

Nokia's Network Service Platform (NSP) is based on ODL and embeds capability to communicate with Nokia network equipment, enriching it with some specific functionality.

As a pioneer in SDN field, ***Google***, in order to provide services of better quality, developed it's own controller platform called Andromeda. The main problem it targets to solve is orchestrating of Data Center network interconnections. Andromeda is used for provisioning, configuring, and managing virtual networks [51]. The main goal of Andromeda is to maximize network performance and at the same time expose NFV.

In 2017 Google announced, that their SDN solution goes to the public internet on the edge of cloud [52]. That's how the Espresso, new SDN stack, was introduced. It's main enhancement consists in providing scaling and provisioning metro-type of networks. Espresso extends SDN to the edge of Google's network, where it connects to other networks across the world [52]. To generalize, Espresso is a SDN stack, Andromeda is a NFV stack [52].

2.3.3 μONOS - next-gen SDN

Roots of both open-source SDN controllers, ODL and ONOS, go in the beginning of 201x. At that time in fashion were the micro-service architecture, mainly based on OSGi framework. With introduction of Docker containers, which can provide higher agility and (software) reliability, OSGi-based architecture became outdated. Right now, there is a need to rework legacy SDN controller architecture and introduce the fresh one, where different controller's modules could scale independently. That's the main idea behind the μONOS.

By the developer's team were already defined main pillars of μONOS architecture. They are following [53]:

- Native support of new-generation control and configuration interfaces and standards, like P4, gNMI, gNOI and others.
- Basis for zero-touch operation support.
- Modular structure based on poly-language gRPC interface for inter-module interactions.
- Platform should be composed as a set of microservices with possibility to be deployed in cloud or Data Center environment. In other words, become more *Cloud-native*.
- Module system based on microservices should be dynamically scalable and performance efficient in terms of throughput and latency.

μONOS is mainly written in Go language and, if necessary, could support low-level C/C++ insertions for better performance optimization. Developer's team defined some functional requirements on μONOS platform design [53], which include:

- Configuration, monitoring and maintenance of network devices.
- Configuration and programming of the forwarding plane.
- Validation of the network topology and of forwarding plane behavior.
- Efficient collection of network performance metric.

μONOS is still in it's beginning and didn't get to the alfa release yet. Currently configuration subsystem is being developed. It is designed to be a separate entity, what allows it to co-exist with current ONOS[5] release (2.x) and fit with next-generation SDN concept [54]. You can see it's architecture on the figure below.

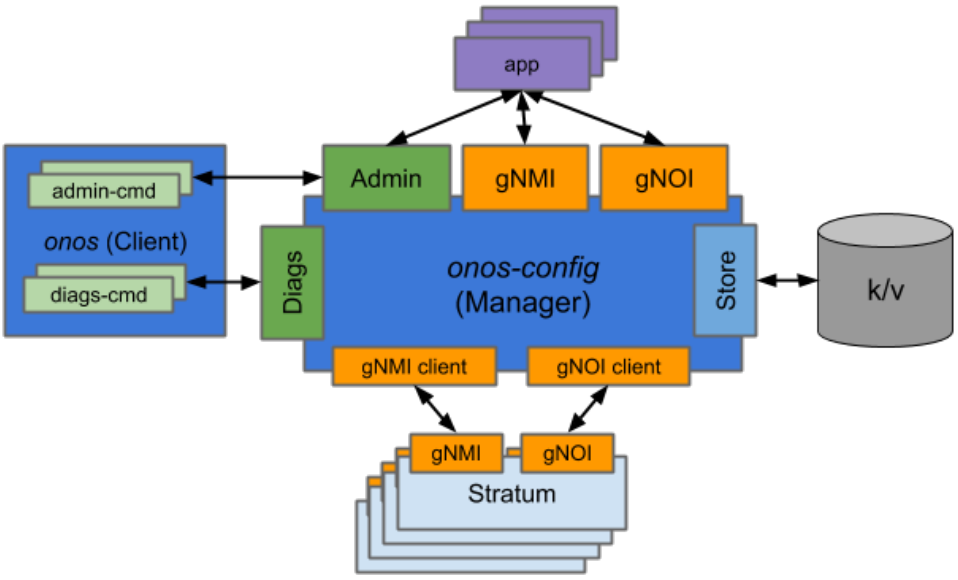

Figure 2.8: μONOS Architecture [54]

[5]Currently ONOS does **not** support configuration subsystem. Device configuration should be done manually through the Cli or WebGUI.

As you can see from the Figure 2.8, principle Northbound API, as well as Southbound API, would be gNMI and gNOI. Transfer of information between interfaces will be done through gRPC. In case of Stratum[6]-based switches no adaptation layer is needed due to the homogenity of Northbound and Southbound API. For the devices, that don't support gNMI and gNOI, adapters could be developed either as an external service or as a proxy agent on the device [54].

2.4 SDN controller comparison

This section presents brief comparison of two main open-source SDN controllers - ONOS and ODL.

As you may think, ONOS and ODL are the same, providing same functionality, but in reality, from architectural point of view, they are very different [56]. Let's have a closer look on the Table 2.2 with the use-cases of both controllers [57].

Use-case	**OpenDaylight**	**ONOS**
Legacy Network Interoperability	YES	YES
Service Insertion and Changing	YES	YES
Network Monitoring	YES	YES
Network Virtualization	YES	YES
Traffic Engineering	YES	YES
OpenStack Neutron Support	YES	YES
ONAP SDN-C Integration	YES	NO
Transport Networks	YES	YES
Path Computation Element	YES	YES

Table 2.2: ODL and ONOS comparison

As you can see, with the time, from the use-case point of view, controllers became almost similar.They have different architectural approaches. ONOS is staying at Application-Driven abstraction, ODL moves forward with Model-Driven abstraction. MD-SAL has it's own advantages and disadvantages with regard to former AD-SAL architecture. It is great concept simplifying application development, but making the debugging process way more harder.

On the one hand, ODL is truly lead SDN platform from the functional point of view. ONOS is a bit behind, but provide all necessary services and APIs in order to develop same set of functionality according to the customer's needs.

On the other hand, ONOS stays simple by it's nature and this way more developer-friendly. Needless to mention, that ONOS has almost perfect documentation (including tutorials) what provides an easy beginning with the developing process. Last, but not least, ONOS community, comparing to the ODL one, is open and very interactive. You can always find answers or get any feedback. It is quite important, especially in the beginning of SDN developer's path.

[6]Stratum is an OS for whitebox switches developed by ONF [55]. It enriches whole SDN ecosystem deployed by ONF.

Chapter 3

Optical Networks

3.1 Brief technological overview

The challenge to open the control and to automate the configuration of Optical Networks are way more wider problematic than is explained in this chapter. This thesis focuses mainly on the provisioning of photonic connectivity services by configuring layer zero, L0 per OSI model, of optical networks. In this chapter technological overview of xPON, WDM or any other technology used in optical networks would **not** be provided. If the Reader is interested in some of these topics, Author summarized some useful links to go through:

- If the Reader wants to read more about Passive Optical Networks (PON), Author suggests to look here [58] or here [59]. Also, it's good to read this [60] beautiful set of articles dedicated to the different aspects of PON networks;
- For better overview in Active Optical Networks (AON) Author suggests to the Reader to have a look at [61];
- For overview in GPON, the Reader can go through [62] and some articles at [60];
- If the Reader wants to learn more about wavelength-division multiplexing (WDM) systems, Author invites him to go through this [63], [64] article.

3.2 Optical elements brief overview

In this chapter we will describe the most common elements which are used in active optical networks. Going through the passive optical components within this work is not really beneficial. Because of that we will go directly to the active optical components overview with special focus on ROADM systems. It is beneficial for better understanding of the implementation part purpose.

3.2.1 Optical amplifiers

Amplifier is important component in any transport optical network. It allows to extend the transmitting distance of the optical network and enlarge it's coverage, what is a key factor. Main advantage of an optical amplifier consists in it's ability to amplify signal directly on photonic layer without any conversion to the electric domain and back to the

optical one. It reduces delay of the system and eases it's maintenance. Different types of amplifiers were invented to meet different signal amplifying requirements [65] in different situations.

In optical transport networks we can differentiate 3 types of the amplifiers according to their usage:

- ***Pre-Amplifier*** is usually being installed near the receiver to ensure that the optical signal could be detected by the receiver. It amplifies optical signal on the certain level. Offers higher gain;
- ***Booster*** is installed right after the transmitter to amplify the signal inserted into the fiber link. Offers lower gain and higher output power;
- ***In-line Amplifier*** is installed every 80-100 km of optical fiber to ensure, that within the whole transmission signal stays above the noise ratio, in other words - detectable. Has moderate gain. [65]

Gain of the amplifier should be calculated carefully before installing it to the optical network.

Reliable work of the optical amplifiers is crucial in optical transport networks. In this sub-chapter we will have a brief overview of a different optical amplifiers.

EDFA

Erbium-doped fiber amplifier (EDFA) is probably the most famous and the most used fiber amplifier. Working principle of this amplifier is based on constant pumping of the energy to the optical fiber doped with Erbium. Laser diode emits the light on 980 nm or 1480 nm wavelength [66]. Such energy pump transfers the fiber to the active state (stimulated emission) where it starts to emit photons with the 1550 nm wavelength. This wavelength is typically used in telecommunications due to the small losses in the optical fiber. Emitted photons then amplify target signal. [66], [67]

Could be used as a Booster, In-line or Pre-amplifier.

YDFA

Ytterbium-Doped Fiber Amplifier (YDFA) has the same concept as an EDFA amplifier, but doped with Ytterbium. It makes slight difference the area of amplifier's usage. For example, amplifiers based on ytterbium-doped fiber can be used to boost 1-μm laser sources [67].

Main drawback of YDFA amplifiers is their non-linear spectral amplification characteristic [67]. For example, in this case some wavelengths could be amplified more than others. It could evolve into other bad effects during the transmission.

YDFA amplifier generates really huge gain and because of that it isn't really used in telecommunication industry.

Raman amplifier

Raman amplifier is a special type of fiber amplifier. It is based on the effect of Raman scattering. To compose such an amplifier you need a Raman-active medium, usually optical fiber, and a laser (pump beam) which transmits optical signal with a wavelength of a few nanometers shorter, than the wavelength of the signal we are trying to amplify [68]. By pumping the energy from the another laser, we are developing Raman scattering effect - evolving emission on the different wavelengths (corresponding to the target wavelength) which then amplifies our target signal [67].

Raman amplifier is very versatile and could be tuned to any wavelength typically used in telecommunications. It competes mainly with EDFA amplifiers in it's usage. [67], [68]

Semiconductor optical amplifier

Semiconductor Optical Amplifier(SOA) uses the semiconductor as the gain medium. Usually operates at a signals in between 850 nm and 1600 nm [69]. Working principle is basically the same as for semiconductor laser - it amplifies incident light through the stimulated emission [69]. Emitted photons have the same wavelength as targeting one. Since SOA amplifier can work on different wavelength, it should be tuned precisely.

Semiconductor amplifier is cost-effective solution typically used as a Booster or In-line amplifier.

3.2.2 Optical Add-Drop Multiplexer

Before diving deeply inside the problematic of Reconfigurable Add-Drop Multiplexers (ROADMs), let's have a short look on it's parent, the OADM.

In the Wavelength-Division Multiplexing (WDM) systems there is a need for multiplexing and routing different channels of light *In* or *Out* of the fiber. Optical Add-Drop Multiplexer (OADM) provides such functionality like adding or dropping one or several wavelengths (channels) and passing those signals to the another network path. OADM can be considered as a specific type of optical Cross-Connection (XC) [70]. OADM is a common type of optical node used in optical telecommunication networks. A traditional OADM consists from three stages:

- ***Optical Demultiplexer*** – separates wavelength at output of the fiber onto input ports;
- ***Optical Multiplexer*** – gets together all wavelengths to the output port;
- ***Method of reconfiguring paths*** between the demultiplexer, the multiplexer and a set of ports for adding or dropping signals. It could be:
  - fiber patch panel;
  - optical switch [70].

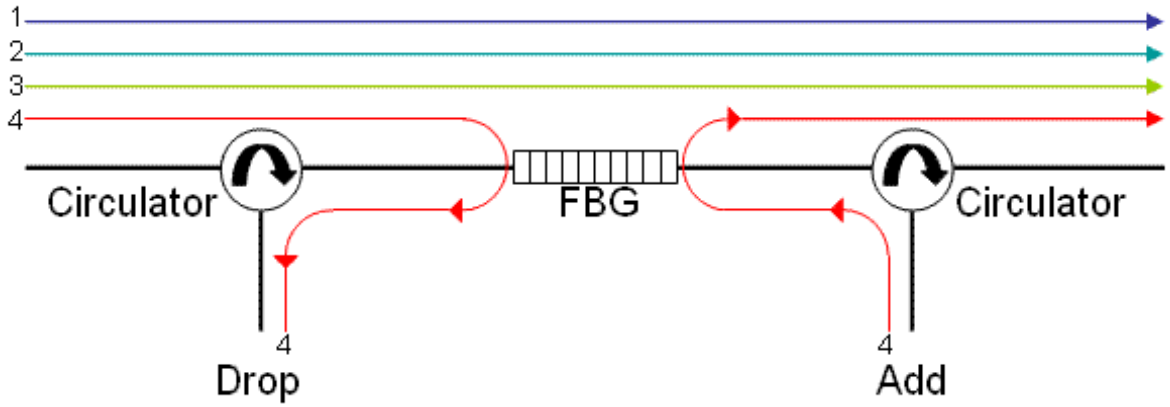

Figure 3.1: Structure of OADM [71]

All lightpaths which go directly through the OADM are called *"cut-through lightpaths"*, while added or dropped lightpaths are respectively called *"added"* or *"dropped"* lightpaths [72].

3.2.3 Reconfigurable Optical Add-Drop Multiplexer

Reconfigurable Optical Add-Drop Multiplexer (ROADM)[1] is active optical element critical for optical transport networks. Next several sub-chapters are targeting to provide a slightly deeper overview of ROADM's functionality and it's main components.

Reconfigurable Optical Add-Drop Multiplexer (ROADM) is basically OADM which adds the ability to remotely switch traffic from a WDM system at the wavelength layer [72]. It is achieved by using the Wavelength Selective Switch (WSS) module, which allows individual or multiple wavelengths carrying data channels to be added and/or dropped from a transport fiber right on the optical layer (L0). There is no conversion from the optical signal to the electronic signal and back, what reduces the delay of such system [72].

ROADM provides a lot of advantages in terms of operating with the network. For example:

- There is no need to plan the entire bandwidth assignment during the initial deployment of the system. Configuration could be done "on-the-fly", when it is needed and without affecting the data traffic passing through the ROADM.
- It allows remote configuration and reconfiguration.
- Automatic power balancing. In ROADM it is not clear, where a signal can be potentially routed, so there is a necessity of power balancing for the signals. [72]

Functionality provided by ROADM originally appeared in a long-haul Dense Wavelength Division Multiplex equipment. It also began to appear in the metro optical systems because of the increasing demand on the network capacity [72].

[1] One of the main tasks of this work is to enable software-based configuration of ROADM. Chapter 4 is dedicated to it.

3.2.3.1 ROADM Architecture

Let's take a deeper look on a general concept of ROADM and try to understand, how it works. ROADM generally consists from two main elements: wavelength splitter and a Wavelength Selective Switch. Let's take a look on the Figure 3.2.

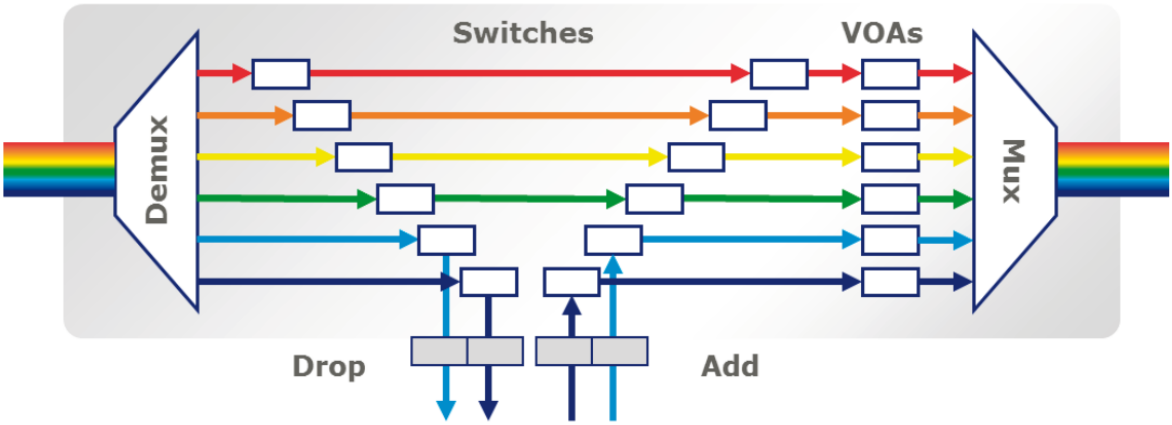

Figure 3.2: ROADM architecture [73]

Optical signal, after entering the ROADM on certain interface, passes through the demultiplexing section (wavelength splitter) and divides into several channels [74]. Each of the channels correspond to the certain wavelength. After that wavelength passes through the Add/Drop section, or Wavelength Selective Switch (WSS), which could respectively drop or also reroute the wavelength to the another direction. It can also let the wavelength pass through. If the wavelengths was redirected, WSS can add the wavelength from another direction on this port. After passing the WSS, all wavelengths are being joined together in a multiplexing section and go out from the interface.

One of such modules is needed per one direction. In a ROADM terminology, direction (or the DWDM line interface) can be also called a degree. For example, for four direction connectivity you need a four degree ROADM [74].

3.2.3.2 Wavelength Selective Switching

To route/switch signals (wavelengths) between optical fibers, Wavelength Selective Switch (WSS) component is being used. Functionality of such element is simple to understand, but hard to implement. The various incoming channels of a common port are dispersed (demultiplexed) continuously onto a switching element which then directs and attenuates each of these channels independently [75]. Operation of such mechanism can be bidirectional, so the wavelengths can be multiplexed together from different ports into a single common port.

WSS is responsible for switching in ROADM. This functionality could be achieved by various techniques. Next few sub-chapters would briefly describe some of these methods.

3.2.3.2.1 Micro-Electro-Mechanical Mirror

The simplest and the earliest commercial solution is based on movable mirrors using the Micro-Electro-Mechanical Mirror (MEMS) [75] technology. The incoming light is broken into a spectrum by a diffraction grate. Each wavelength then focuses on separate MEMS mirror. By tilting the mirror in one dimension, the channel can be directed back into any of the fibers in the array.

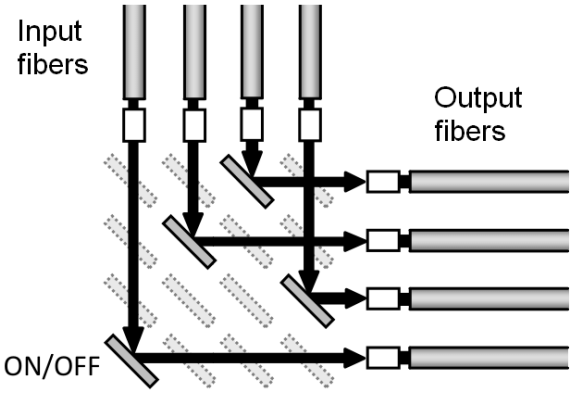

Figure 3.3: WSS with MEMS principle [76]

The technology has an advantage of a single steering surface, not necessarily requiring the polarization diversity optics. It also works well in the presence of a continuous signal, allowing the mirror tracking circuits to dither the mirror and maximize coupling [75].

This kind of WSS typically produces poor open loop performance, but good extinction ratios [75]. During manufacturing, the channels must be carefully aligned with the mirrors, what complicates the manufacturing process. Also, phase of the light is not really well controlled during the mirroring and artifacts can appear due to the interference of light from the other channels. [75]

3.2.3.2.2 Binary Liquid Crystal

Liquid crystal switching avoids complexity of manufacturing and high costs. Again, diffraction grating breaks the incoming light into the spectrum. Array of liquid crystals, controlled by the software, performs switching function. Each liquid crystal (LC) corresponds to one wavelength. Liquid crystal individually can let the light pass and *"adds"* it, or doesn't let the light pass and *"drops"* it. At the output, all left wavelengths are multiplexed into the one port. [75]

This technology has an advantage of relatively low cost parts, simple design of control electronics and stable beam positions without any active feedback. Somehow, this simple design is valid for one fiber only. In case of the transport network, where we must operate with hundreds of optical fibers, complexity of such system increases significantly.

Main disadvantage of this technology consists in the sickness of switching elements. It is hard to keep the optical beam focused over this depth. [75]

3.2.3.2.3 Optical filtering

Same concept as in Binary Liquid Crystals is applied in terms of optical filters. Each filter is meant to let only certain wavelength pass through, and drops the rest. By composing arrays of such filters we can create a selective switching array. [75]

3.2.3.2.4 Liquid Crystal on Silicon

Liquid Crystal on Silicon (LCoS) is attractive as a WSS mechanism because of continuous addressing capability, which enables new functionality [75]. The main enhancement comparing to the Binary Liquid Crystal is that the device shouldn't be pre-configured for the specific application. It can be configured remotely via software. Additionally, it is possible to reconfigure the channels while device is still operating. LCoS technology introduces more flexible wavelength grids, which help to unlock the full spectral capacity [75] of optical fibers. New features also include shaping the power levels within a current channel, broadcasting the optical signal to more than one port and fine-grained channel control (central, minimum, maximum frequencies, channel bandwidth) via embedded software [75].

3.2.3.3 New generation ROADMs

It's been a while, since first concept of the ROADM was introduced to the market. Over the time, design of the ROADM became more sophisticated and new functionality were introduced. We can split ROADMs on following categories.

Colorless ROADM is the ROADM which enables flexible allocation of any wavelength (or color) to any port [74]. It contains one WSS switch per one degree (direction). Complete software control. For realizing colorless feature, filter modules should be implemented.

Directionless ROADM does not require physical reconnection of the transmission fibers. This kind of the device is being deployed for temporary installations or for restoration purposes [74]. It avoids restrictions regarding to the directions. *"Directionless"* technology could also be used to reroute the wavelength in a different direction. Directionless ROADM is very important for true optical flexibility [77].

Contentionless ROADM is the ROADM which eliminates the possibility of collision at the same port for two identical wavelengths [74]. It must provide dedicated internal structure to avoid this. Contentionless architecture allows multiple copies of the same wavelength on a single add/drop structure with no particular restrictions [77]. It eliminates the necessity of manual intervention (physical recabling) in some cases [77].

Gridless (Flex Spectrum) ROADM supports various channel grids specified by ITU-T G.694 [78] within the same optical signal. Such ROADM could be adapted for future transmission speed requirements [77]. When you need a transfer speed more than 100 Gb/s, standard bandwidth of 50 GHz could not be enough [77]. Also, different modulations within the same signal (on different parts of spectrum) could be used. At the same time operator could require such high speed in combination with 40Gb/s or 100 Gb/s, what makes flexibility of spectrum provided by gridless ROADM very important. This type of ROADM could be useful for transmission containing different modulations or for coherent transmission.

Combination of Colorless, Directionless, Contentionless and Flex Spectrum (CDC-

F) properties provides an ultimate level of flexibility [74]. According to [77], **CDC-F ROADMs** could help to save expenses on the network provisioning, since technicians don't need to manually reconfigure network equipment anymore. Automation also greatly accelerates bandwidth (or capacity) provisioning. Operators will have an ability to respond to the rising demand on the capacity (i.e. for delivery of cloud based services). Even greater benefits could be achieved in the topology flexibility and simplified operations with the network equipment. What is more important, human factor will be eliminated. CDC-F ROADM is a necessary prerequisite for telecommunication operators, if they want to implement openess and automation with SDN in their transport network [77].

Chapter 4

Implementation

Deploying SDN architecture in Open Optical Networks is a challenging task. First of all, we should choose a proper open-source SDN controller with regard to the available equipment. After that we should define the goals we want to achieve.

Beginning of this chapter explains choice of the SDN controlling platform and sets the goals. Second part of this chapter describes the implementation process.

4.1 Targets

In the lab setup we have in disposition following Nokia's flagship devices[1]:

- **2x *Nokia 1830 PSI-2T*** - Optical Transponder.
- **2x *Nokia 1830 PSS*** - Reconfigurable Optical Add-Drop Multiplexer (ROADM).

Both of these devices implement Open-APIs. They support capability to communicate over NETCONF protocol, which could be considered as an Open-API. Important to mention, devices support OPENCONFIG data models.

Since one ROADM has 1-degree and other ROADM has 2-degree, we don't have too much freedom in setting various topologies in the lab. Also, we would like to manage and configure the devices through the SDN controller in order to establish end-to-end optical channel connectivity. This task is already quite challenging. Let's choose a proper SDN controller through which we would manage these devices over the NETCONF protocol. Next sub-section is dedicated to it's choice.

4.1.1 SDN controller choice

Open-source SDN solutions, like ONOS or ODL, were already described and compared in section 2.3. With respect to the basic functionality, they have no differences. Somehow, getting familiar with an internal structure is more complicated in case of OpenDaylight.

Author's definitive choice of the SDN controller was ONOS. There are several reasons behind it:

[1] Overview of both Nokia's flagship devices is provided in sub-section 4.1.2 and sub-section 4.1.3.

1. **Documentation** is not outdated comparing to the OpenDaylight.
2. More **interactive community** which responds and navigates you.
3. **No need in advance functionality** provided by OpenDaylight. ONOS has already everything what you need inside. Just use it.
4. **ODTN project** inside the ONOS lead by Andrea Campanella which targets to embed functionality for Open and Disaggregated Transport Networks (ODTN).
    - The driver with basic functionality to manage the Nokia 1830 PSI-2T Optical Transponder was already contributed to the ONOS community.
    - Somehow, ONOS doesn't implement possibility to manage the Nokia's ROADM. We need to develop a driver on our own.

4.1.2 Nokia 1830 PSI overview

Nokia 1830 Photonic Service Interconnect (PSI) is an optical transponder developed for Data Center Interconnection (DCI) applications [79]. It is shown on Figure 4.1.2

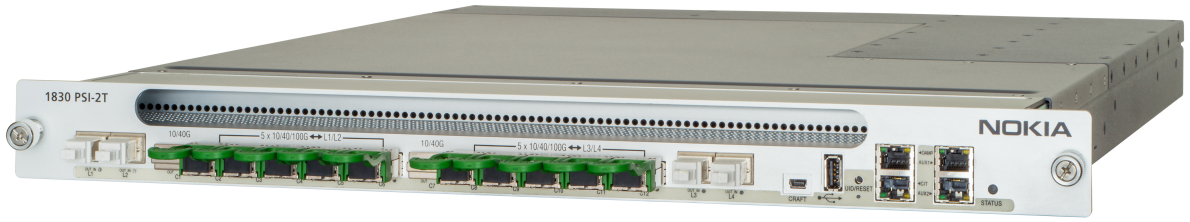

Figure 4.1: Nokia 1830 PSI-2T [79]

This device has following specifications [79]:

- 100G, 200G and 250G coherent optical *line ports*;
- 40GE and 100GE *client ports*;
- Integrated L1 encryption based on AES-256 algorithm;
- Streaming telemetry for real-time provisioning;
- Support of Open-API interfaces.

It is a high capacity and cost efficient network solution for DCI applications over metro, regional and long-haul distances. This device offers optimized network solutions for cloud era [79]. It's also possible to have an alternative solution on modular basis, Nokia 1830 PSI-M, where you can attach different modules from the catalogue corresponding to your demand.

In the lab setup we have two Nokia 1830 PSI-2T.

4.1.3 Nokia 1830 PSS overview

Nokia 1830 Photonic Service Switch (PSS) is an optical CDC-F ROADM. It supports next-generation DWDM multi-service, multi-layer P-OTN transport [80].

This device supports 100G-600G wavelength transport. It tents to transform traditional DWDM into a flexible transport layer by providing an agile wavelength routing and scalable multi-layer switching and services. [80]

Figure 4.2: Nokia 1830 PSS [80]

Meeting unpredictable traffic demands by optimizing optical networks is the main focus of this device. Area of application is following [80]:

- Metro and long-haul transport;
- Broadband backhaul;
- Data Center Interconnection;
- Carrier Ethernet;
- Wavelength services.

100G connectivity services with 100G multi-terabit OTN switching and flexible (100G–600G) wavelength transport are supported by Nokia 1830 PSS [80].

This platform allows to deploy services rapidly, reduce network TCO and extend it's lifecycle [80]. It also provides support of Open-API in order to deploy SDN architecture and distributed GMPLS control option. These capabilities let us dynamically maximize network capacity and efficiency.

Nokia 1830 PSS allows creating of cross-connectivity between any input *client* port and any output *degree* port (usually amplifier's card ports). Such an approach provides mapping of input ports to the output ports and de-facto makes the device to act as a wavelength router.

There are different versions of the device, which basically differ in the scale of usage (from metro to international). Somehow, each platform has common software, common hardware and common control functions [80].

In the lab setup we have one 1-degree Nokia 1830 PSS-16II and one 2-degree Nokia 1830 PSS-16II.

4.1.4 Setting the goals

We're targeting to manipulate with the optical device configuration over the SDN controller. A good result could be an ability to configure both devices under ONOS and establish end-to-end optical channel connectivity. Steps we need to accomplish in order to deploy SDN architecture in Open Optical Networks are following:

1. Part I
   - Implement an ONOS driver for the Nokia 1830 PSS.
   - Write a Cli command through which we could set the XCs on the device.
   - Implement PowerConfig interface for the existing ONOS driver for Nokia's 1830 PSI-2T Transponder.
   - Perform some tests to verify and validate functionality of this solution in terms of end-to-end optical channel connectivity.
2. Part II
   - Implement AlarmConfig interface, which would treat alarms in proper way on both devices.
   - Develop an application (or skeleton), which is capable of reconfiguring the network devices in given topology.

4.2 ROADM driver

To deploy automatically configurable transport networks, we need to manage ROADMs under the ONOS controller. For this purpose we need to develop a driver and integrate it inside the SDN controller.

This section explains general steps for implementing of such driver in ONOS. Detailed explanation of the whole process is also provided.

4.2.1 Driver overview and general steps for it's creation

Let's have a look on what does the "Device Driver" mean in terms of ONOS. What does it do? There are some pillars which define any driver implementation in ONOS:

- Each driver is defined by the set of the behavioral models (and their implementations).
- Behavioral model is implemented as a Java interface. It describes set of functions (or "behaviors") which device is capable to perform.
  - Functionality of behavioral models mostly aligns with OPENCONFIG Data Models.
- Any behavioral model could require specific implementation for any device.
- Device driver could have as many **different** behavioral models as it needs.
- Set of the behaviors is specified inside the *xml* file of the driver with corresponding XML tag.

Based on this definition, we can state that ONOS is not device-independent. Once we want to control a new device under the controller, we need to write a custom driver. Because of that almost all drivers are more less similar. They differ[2] only in the way of extracting necessary data and parsing them to the corresponding internal ONOS structures. Below are summarized minimal requirements that any driver needs to implement, if we want to manage the device under the ONOS controller:

1. Parse device to the ONOS and register it inside the controller.
2. Establish communication session (over NETCONF protocol).
3. Read out all basic information from the device about the software and hardware components, including number of ports and their type.
4. Implement some management logic in order to configure the device.

Our task is to manage the Nokia 1830 PSS-16II device, which is CDC-F ROADM. We need to implement our own custom driver in order to discover the device information and manipulate with it's XCs. To achieve this, we need to implement two following behavioral models:

- **DeviceDescriptionDiscovery** - read out all basic information from the device about the software, hardware and ports. Store this information inside the internal structures of ONOS;
- **FlowRuleProgrammable** - main configuration logic of the device is implemented here. In our case, this interface is responsible for creating and removing XCs on the device.

Goals are set, let's dive deeper into the implementation part.

Before reading following sub-chapters, Author would like to notify the Reader, that it would be much more comfortable and easier to read the diagrams introduced below in the electronic version of this work.

[2]On the time of publishing of this work, there were some actions on consolidating existing drivers and unifying them. You can find more here [81]

4.2.2 Implementing DeviceDescriptionDiscovery behavior

"DeviceDescriptionDiscovery" behavior is responsible for extracting of the basic information from the device and storing it inside the ONOS. This behavioral model is fundamental to implement. Later on, you will see that all management would be based on the obtained information within this behavior.

To implement it, we created a file called *"NokiaPssOpenConfigDeviceDiscovery.java"* and stored it inside the *"/odtn"* folder of the driver:

$ONOS_ROOT/drivers/odtn-driver/src/main/java/org/onosproject/drivers/odtn/

Each behavior is defined by the Java interface instance. In that case, it's implementation always means that you should strictly implement all functions defined by the interface. If necessary, you can decompose functionality of mandatory functions on the set of smaller functions. It is very useful, especially when such functions have common routines. These assumptions make the code cleaner and easier to read. Behavior's functionality of *"NokiaPssOpenConfigDeviceDiscovery.java"* is depicted on the diagram Figure 4.3.

As you can see on Figure 4.3, there are two main functions - **discoverDeviceDetails** and **discoverPortDetails**. They are being triggered every time the driver is invoked. First of all, both of them establish the NETCONF session in a specific way. All Nokia's devices running on OpenAgent (software, operating system of the device) require two-level authentication with the following method:

1. Exchange **<hello>** messages with capabilities.
2. Pass the **<login>** request with correct credentials to authenticate the session.
    - This RPC request is a custom one. It is implemented in one of the custom data models passed during the initial **<hello>** message exchange.

New operations are typically identified with a new capability. It should be added to the list of capabilities sent by the NETCONF server during the initial *<hello>* message exchange. In case of Nokia's ROADM, and any other OPENCONFIG-based device, a lot of custom *<rpc>* commands were added to the NETCONF protocol. It helps to extend operability and provides more precise tuning of the ROADMs (i.e. allows to create cross-connections, set frequency/power, implements additional layer of security by adding necessary *<login>* requests to the communication flow and many more). Because of that, in the file ***odtn-driver.xml***, in one of the properties, were passed the capabilities obtained from the device ***<hello>*** message. Due to the privacy of this information, appropriate data models wouldn't be shown there. You can have a look on the existing Nokia 1830 PSI-2T driver [82].

Once two-layer authentication is passed, we can obtain all information we need by composing other (custom) RPC requests. One of them is **<get system-software>**, which is invoked inside the **discoverDeviceDetails** function. We're sending this request to obtain an answer, from where we can extract all necessary information and store it inside the *DefaultDeviceDescription* instance (it is responsible for storing a general information about the device).

After **discoverDeviceDetails** function ends it's execution, ONOS triggers execution of **discoverPortDetails**. This function is a bit more complex. As were previously described, it establishes NETCONF session by passing two-layer authentication first. Then, in **buildGetPlatformComponentsRpc**, it composes a custom RPC request to get an

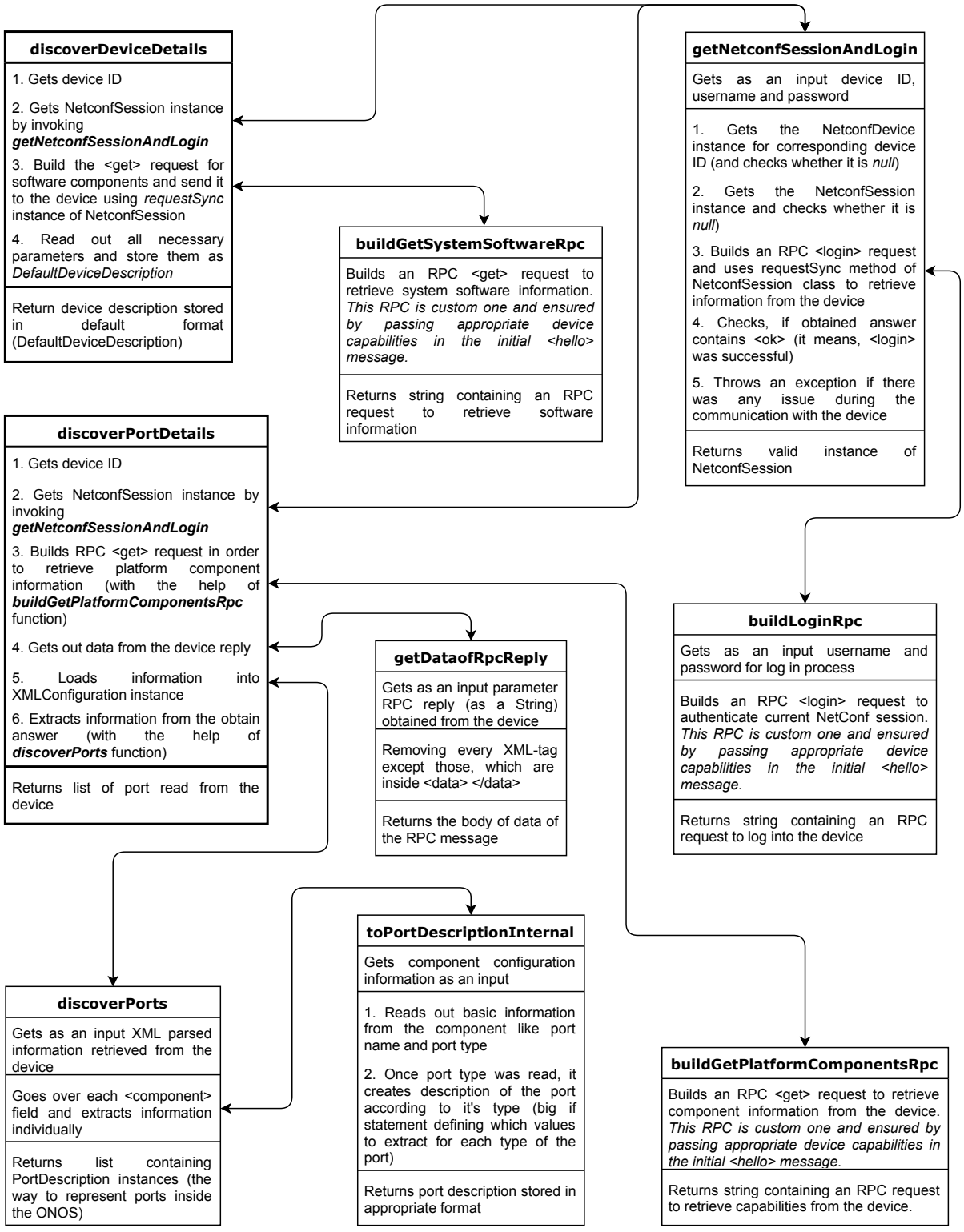

Figure 4.3: DeviceDescriptionDiscovery interface workflow

information about the device ports and parses obtained information to the internal ONOS storage in appropriate data structure.

Since the device can have multiple ports, obtained reply is decomposed on smaller sections. Each of them contains information about the single port. It is done inside

the **discoverPorts** function. On each section is applied universal algorithm to extract necessary data - type of the port, name of the port and many other parameters. Simultaneously with the extraction, all information is stored in the internal storage of the ONOS controller.

During the development of this behavior an issue described in sub-section 6.1.1.1 was found.

Once the basic information about ports is extracted and stored, we can move on and implement the main device functionality - cross-connection creation. Next sub-chapter is dedicated to this task.

4.2.3 Implementing FlowRuleProgrammable behavior

Implementing of *FlowRuleProgrammable* behavior is always tricky. It tightly depends on the device functionality and can vary from device to device. This behavioral model manages "Flow Rules" installed on the device. In terms of ONOS, "Flow Rule" represents the "Rule" installed inside the device or, in other words, piece of device-specific configuration [83]. Inspired by OpenFlow protocol, ONOS, as a SDN controller, installs "Flows" into the device. Flow Rule Subsystem interacts with a specific part of the driver, implementation of *FlowRuleProgrammable* behavior, in order to manage installed on the device rules.

In general, we can divide *FlowRuleProgrammable* interface on three parts:

- ***Get Flow Entries***[3] is responsible for obtaining configuration already installed on the device. It parses configuration to the internal structures of the ONOS in order to display it later inside the controller.
- ***Apply Flow Rules*** is responsible for *installing* Flow Rule on the device. It parses information from the FlowRule structure to the device-specific code. This part of the behavior is triggered by corresponding microservice of the ONOS (RoadmManager in our case).
- ***Remove Flow Rules*** is responsible for *deleting* Flow Rule on the device. It parses information from the FlowRule structure to the device-specific code. This part of the behavior is triggered by corresponding microservice of the ONOS (RoadmManager in our case).

Describing all three parts simultaneously is quite complex. We spread our explanation on the next three sub-sections. Corresponding workflow diagrams would be introduced to ease the understanding.

4.2.3.1 Get Flow Entries

As were said before, "Get Flow Entries" part of the *FlowRuleProgrammable* behavioral model ensures obtaining of the installed "Flow Rules" (or configuration) from the device. Workflow of this part is depicted on Figure 4.4. It represents the **"Get"** part of the FlowRuleProgrammable interface implementation.

Basically, main task of this part of the interface is to read out the configuration from

[3]Difference in the name - Entry instead of the Rule - is given by the origin of the obtained data. Since, we are talking about the configuration already installed on the device and we want to pass it to the ONOS, it means, that this configuration is literally "enters" the controller. That's where the name comes from.

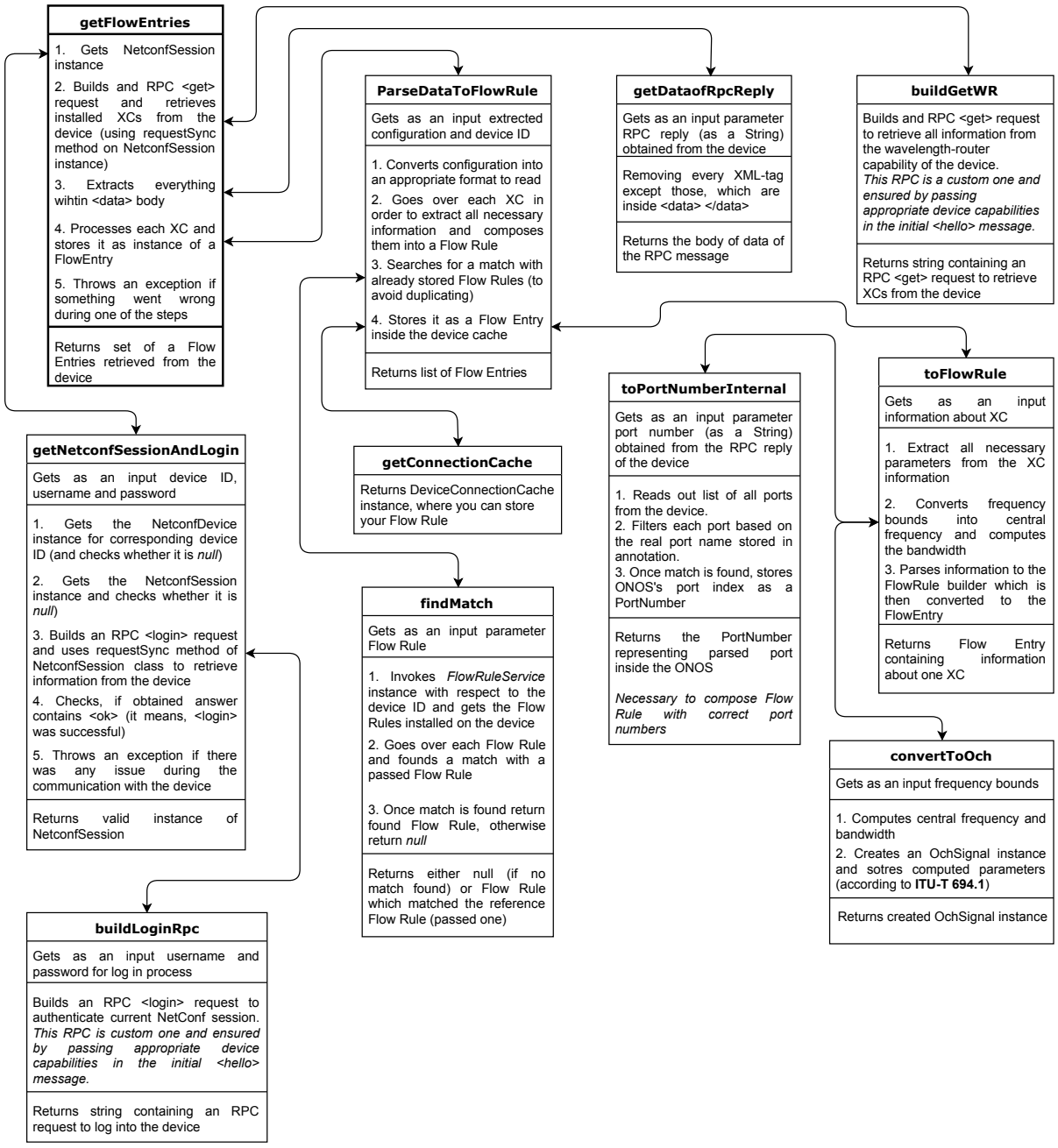

Figure 4.4: FlowRuleProgrammable interface workflow. Getting the Flow Rules installed on the device

the device and interpret it with regard to the internal ONOS structures. It means, that we need to perform following steps:

- ”Ask” the device about it’s configuration;
- Obtain reply and extract necessary data;

- Convert extracted data into the appropriate format and compose *FlowEntry* instance (Flow Rule coming in the direction from the device to the ONOS).
  - Also, we must be sure, that composed *FlowEntry* doesn't duplicate. We would search inside the FlowRule Storage and make sure, that the Flow Rule is unique one. Once we found a match in a store, let's just update a status of the found Flow Rule on **"ADDED"**.

When the device is added to the SDN controller, whole routine is triggered by invoking of the **getFlowEntries** function. After that, corresponding chain of functions is performed. As you may see from the diagram on Figure 4.4:

1. We start our communication with the device by passing two-level authentication.
2. After that, we compose a custom RPC request to obtain configuration about the existing XCs. Wavelength-router, the OpenConfig data model, is responsible for handling that. It was described in Appendix A.
3. Obtained data are parsed and converted to the Flow Rule (by **toFlowRule** function). We also search for it's duplicates (with **findMatch** function).
4. Finally, Flow Rule is converted to the Flow Entry(s) (mandatory format). It is stored in the device cache instance provided by the **getConnectionCache** function. Later on it will allow us to display and manage Flow Rules under ONOS.

This is the general flow of the ***GetFlowEntries*** part of the behavior. For more details please go through the diagram on Figure 4.4.

The main issue and main complexity there came from necessity to parse all required parameters in a correct format to compose the Flow Entry. Certain level of complexity comes from Java. ONOS controller extends standard interfaces and classes by introducing it's own ones. These new instances are built on top of the existing Java classes. For example, ONOS project introduces class for storing the IPv4 addresses - *IPv4*, or *PortNumber* class as an internal representation of a port number. Almost all of these classes are extended by other classes, where each brings something new.

Such an approach complicates relations between different substructures of the controller. It's not always clear on how to parse the certain data in a correct format. This is one of the reasons, why some support functions like **toPortNumberInternal**, **convertToOch**, **toFlowRule** and **findMatch** were introduced.

4.2.3.2 Apply Flow Rules

Next step is to implement the "Apply Flow Rules" part of the *FlowRuleProgrammable* behavior. Specifically this part is responsible for installing the Flow Rules on the device. To do this we must handle the interpretation of the Flow Rule coming from ONOS back to the device-specific format.

The idea behind is following:

- User composes and passes the parameters through the appropriate Cli command inside the ONOS shell (karaf).
- These parameters are passed to the corresponding instance, which composes specific Flow Rule.
- Then, newly born Flow Rule is passed to the appropriate device driver.

- Invoked driver instance triggers execution of **applyFlowRules** function inside the *FlowRuleProgrammable* behavior implementation. It executes corresponding routine to parse passed *FlowRule* and "installs" it to the device.

You can see the workflow diagram of this part of the FlowRuleProgrammable interface on the Figure 4.5

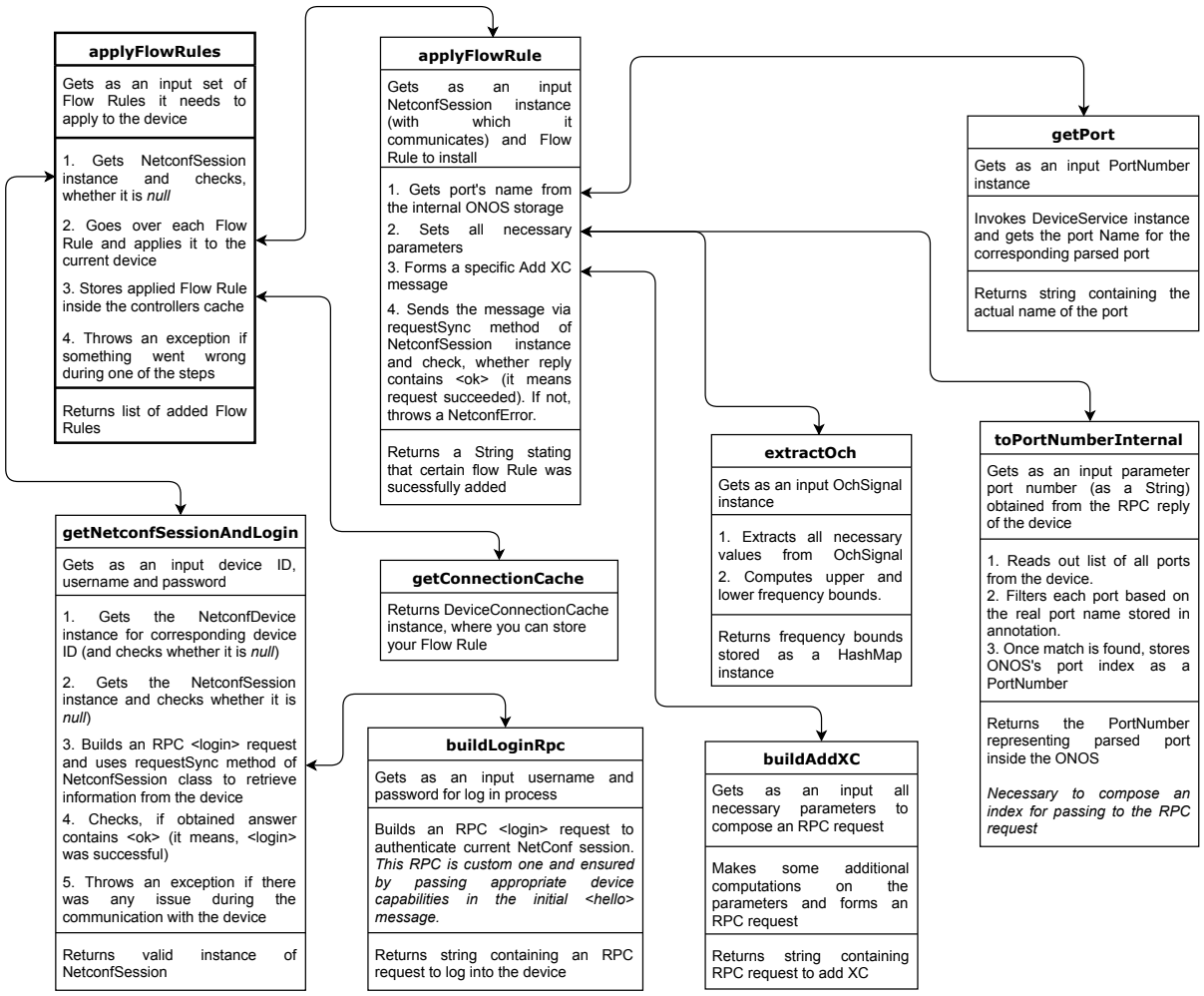

Figure 4.5: FlowRuleProgrammable interface workflow. Applying the Flow Rules to the device

Again, as in the previous section, communication with the device starts from passing the two-level authentication. After that we're iterating over each passed Flow Rule and decompose it on parameters which we need to pass to the device. Since we're working with NETCONF protocol, we need to convert all parameters to the corresponding format to pass them as a missing puzzle of the final RPC request. For this purpose some additional functions, like **getPort** (to get port name), **extractOch** (to get frequency), were introduced.

Once the RPC request is ready, we can send it to the device and get the reply. If the reply states, that request was satisfied (usually contains **<ok/>** tag in the reply), we can store this Flow Rule inside the device cache. It would be visible inside the ONOS Flow Rule database.

4.2.3.3 Remove Flow Rules

If we want to delete the existing Flow Rule, same routine, as were described in the section 4.2.3.2, is applied. The only difference now is that the driver executes **removeFlowRules** function, which triggers a bit different sequence of necessary functions.

This part of the *FlowRuleProgrammable* behavior implementation is very similar to "Apply Flow Rules" part. You can see it's workflow on Figure 4.6.

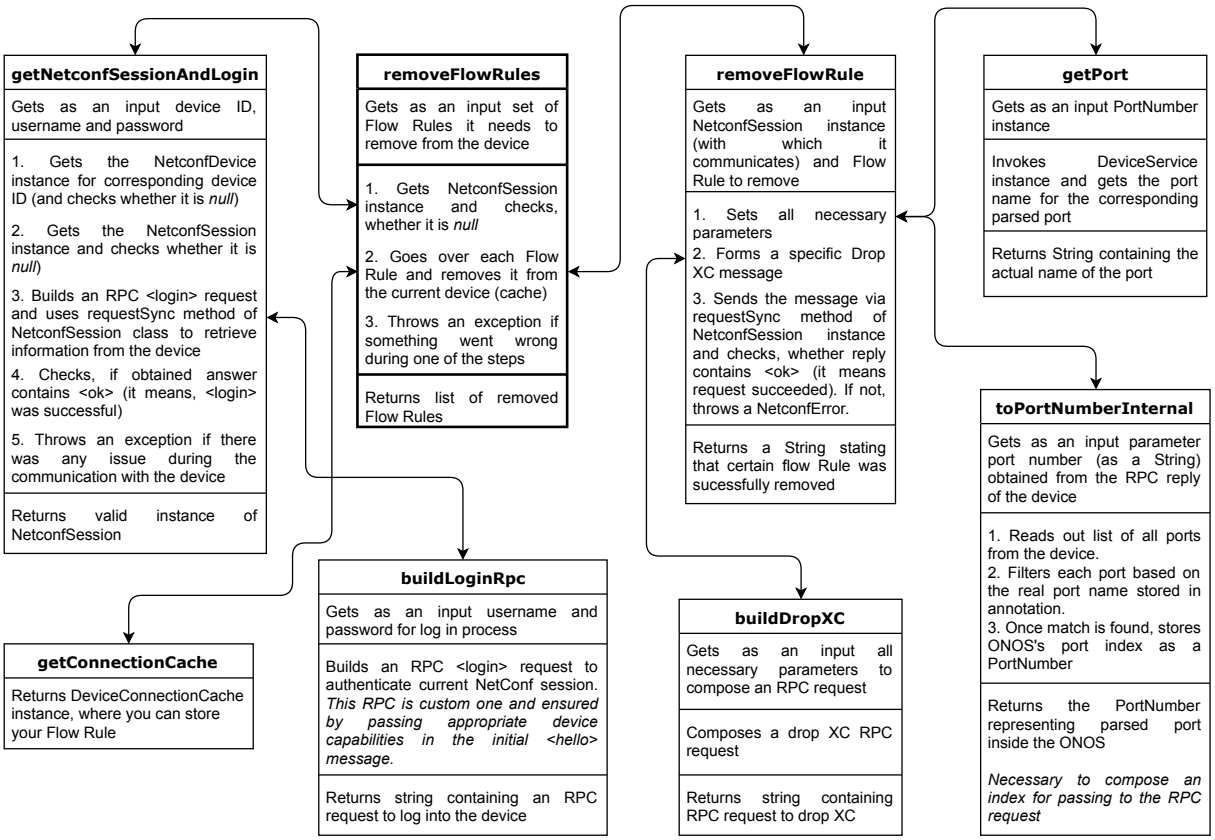

Figure 4.6: FlowRuleProgrammable interface workflow. Removing the Flow Rules from the device

4.2.4 Some more modifications

In order to integrate full device functionality, during the driver development we needed to make some additional changes in some ONOS subsystems. Some of these fixes were contributed to the ONOS community and would be included in ONOS 2.3 release. Some of them stayed proprietary. Let's assume, that we want to run the driver in ONOS 2.2. We should make some additional changes in ONOS core.

First of all, we need to modify *FlowRuleManager* instance in order to make ONOS storing Flow Rules already installed on the device. Proposed solution was following - extend current API of *FlowRuleManager* on another flag, *importExtraneousRules*. You can see this short fix below.

```
// the device has a rule the store does not have
if (!allowExtraneousRules) {
    extraneousFlow(rule);
} else if (importExtraneousRules) { // Stores the rule, if so is
 ↪  indicated
    store.addOrUpdateFlowRule(rule);
}
```

Main idea behind is following:

- ***allowExtraneousRules*** indicates, if you want to leave existing configuration on the device before it's been connected to the ONOS.
- ***importExtraneousRules*** specifies, if you really want to import these configuration inside the ONOS in order to manipulate with it in the future.

By default ONOS doesn't save existing device configuration, or Flow Rules. It deletes it right after the device discovery procedure is finished. To make existing XCs, represented as a set of Flow Rules, be stored in ONOS, we should pre-set the controller right after it's start through the command line with following commands:

cfg set org.onosproject.net.flow.impl.FlowRuleManager allowExtraneousRules true
cfg set org.onosproject.net.flow.impl.FlowRuleManager importExtraneousRules true
cfg get org.onosproject.net.flow.impl.FlowRuleManager

Last command shows you the values of *allowExtraneousRules* and *importExtraneousRules*. You can verify that desired variables are set on *True*. This patch of *FlowRuleManager* was contributed to the ONOS community. You can find it here [84]

After we extended *FlowRuleManager* API, we should modify *FlowRuleParser*. Currently, Flow Rule for optical devices contains only one port. For the ROADM is crucial to parse two ports between which we want to create cross-connections. Following short snippet represents the way how the problem was solved.

```
if (c instanceof PortCriterion) {
    inputPortNumber = ((PortCriterion) c).port(); // obtain input port
    portNumber = ((PortCriterion) c).port();
}
    ....
if (i instanceof Instructions.OutputInstruction) {
    outputPortNumber = ((Instructions.OutputInstruction) i).port(); //
     ↪  obtain output port
    portNumber = ((Instructions.OutputInstruction) i).port();
}
```

Input port is always stored as a *Criterion* instance, output port is always stored as an *Instruction* instance inside the *Flow Rule*. We also must specify according functions to return desired values.

```
    public PortNumber getInputPortNumber() {
        return inputPortNumber;
    }

    public PortNumber getOutputPortNumber() {
        return outputPortNumber;
    }
```

Once these changes are implemented, we can run the driver without any issues.

4.2.5 In conclusion about the driver

Developed version of the driver is applicable for ONOS 2.2. Upcoming version, ONOS 2.3, would have some changes inside the driver subsystem, which is currently being consolidated. Drivers are unified with regard to the OPENCONFIG data models. You can follow the driver consolidation flow here [81].

Developed driver allows user to create unidirectional cross-connections in the Nokia's ROADM. Driver uses NETCONF protocol for communication with the device. Software running on the Nokia's ROADM supports creation of *only* unidirectional XCs over NETCONF protocol. In case, you want to have bidirectional cross-connection, you need to create XC twice.

Developed driver, somehow, has some limitations.

- **Port indexing**. From unknown reasons port indexing was each time different, **NokiaPssOpenConfigDeviceDiscovery** is executing twice. Because of that, device ports are written to the ONOS twice. A while after, store is being updated and removes all duplicated ports. It results in a each time different indexing of a ports.
- **Inability to drop existing XCs**. In order to remove XC from the ROADM, we should pass some specific information[4] related to this XC. This information, somehow, doesn't align with the Flow Rule representation inside the ONOS. It is being generated from the parameters stored inside the Flow Rule, particularly from the port index. As were stated before, port indexing is a bit random. In case, we connect ROADM with already existing XC and we want to manage them, it is not possible.

During the driver development few bugs were encountered. You can find their description in sub-section 6.1.1. Some of these issues were contributed to the ONOS community as a patch.

4.3 Cli command development

Once the driver is finalized, we should develop the Cli command. We want to allow the configuration of Cross-Connections (XCs) on the device through the command line of the ONOS.

During the development we decided to add command's implementation to the *"ROADM"* application of the ONOS. It's done in order to benefit from the services and related packages provided by the *"ROADM"* application. This application could be found in:

$ONOS_ROOT/apps/roadm/app/src/main/java/org/onosproject/roadm/

In this directory we create another one called ***"/cli"***, where we would store the code of the new Cli command. Next, we create a file called "RoadmCrossConnectCommand.java". It implements the Cli command logic. Below is provided the skeleton code of such Cli command. Comments are explaining some implemented features.

[4]This specific piece of information couldn't be disclosed, since it's proprietary information about the device.

```
/**
 * This is the command for creating/removing cross-connections on/from
 ↪  the ROADM.
 *
 */
@Service
@Command(scope = "onos", name = "roadm-xc",
        description = "Creates/Removes cross-connection on/from the
         ↪  ROADM")
public class RoadmCrossConnectCommand extends AbstractShellCommand {

    @Argument(index = 0, name = "operation",
            description = "Specify Create or Remove action",
            required = true, multiValued = false)
    @Completion(RoadmCrossConnectCommandCompleter.class)
    private String operation = null;

    @Argument(index = 1, name = "deviceId",
            description = "ROADM's device ID (from ONOS)",
            required = true, multiValued = false)
    @Completion(DeviceIdCompleter.class)
    private String deviceId = null;

    @Argument(index = 2, name = "srcPort",
            description = "XC's source port {PortNumber}",
            required = true, multiValued = false)
    private String srcPort = null;

    @Argument(index = 3, name = "dstPort",
            description = "XC's destination port {PortNumber}",
            required = true, multiValued = false)
    private String dstPort = null;

    @Argument(index = 4, name = "freq",
            description = "XC's central frequency in [GHz]",
            required = true, multiValued = false)
    private String freq = null;

    @Argument(index = 5, name = "sw",
            description = "Frequency Slot Width in [GHz]",
            required = true, multiValued = false)
    private String sw = null;

    @Argument(index = 6, name = "gridType",
            description = "Frequency grid type. Could be FLEX, CWDM, DWDM
             ↪  or UNKNOWN.",
            required = false, multiValued = false)
    private String gridType = null;

    @Argument(index = 7, name = "channelSpacing",
            description = "Channel spacing in [GHz]",
            required = false, multiValued = false)
    private String channelSpacing = null;

    @Override
    protected void doExecute() throws Exception {
        // Main logic of the command is implemented here
        if (operation.equals("create")) {
            print("Creating XC !");
        } else if (operation.equals("remove")) {
            print("Removing existing XC !");
        } else {
            print("Unspecified operation -- {} --", operation);
        }
    }
}
```

You may notice "**@Service**" tag in the beginning of the code. It is really important to add it. This statement registers new Cli command inside the Karaf. There are some more "**@**" options, which implement different functionality. Some of them define variables to pass, some of them enable autocomplete feature.

Based on the definition above Cli command in action should look like:

roadm-xc create/remove deviceId srcPort dstPort centralFrequency slotWidth gridType channelSpacing

For example, if we want to create XC on certain device, we should type in the command line following:

roadm-xc create netconf:123.123.123.123:830 17 21 195500 50 DWDM CHL_6P25GHZ

From the example above we can see following parameters passed:

- option ***"create"*** represents the **ADD** action, ***"remove"*** accordingly specifies **DROP** action;
- ***"netconf:123.123.123.123:830"*** is a device ID, where **123.123.123.123** represents an IP address of the device, ***830*** corresponds to the common port of the NETCONF communication;
- ***"17"*** is a port number representation (stored in ONOS) of a source port;
- ***"21"*** is a port number representation (stored in ONOS) of a destination port;
- ***"195500"*** is a central frequency of the channel in [GHz];
- ***"50"*** is a slot width of the connection in [GHz];
- ***"DWDM"*** is a type of the frequency grid;
- ***"CHL_6P25GHZ"*** is a channel spacing distance in [GHz]. Value corresponds to 6.25 GHz

Now, after we've got familiar with the skeleton code, let's have a look on the Figure 4.7 to get structural overview of the logic implemented in our solution. You can also checkout the full code of the Cli command here [85] as it was contributed to the ONOS community.

As you can see on Figure 4.7, the main function, which triggers all actions, is **doExecute**. First parameter passed to the Cli command decides with the help of *if* statement, which part should be executed - **addRule** or **dropRule**.

If we're creating XC, we need to convert input parameters into appropriate format in order to pass them to the corresponding instances later. To convert Central Frequency and Frequency Slot Width to the *OchSignal* instance, **createOchSignal** function is invoked. This function is tightly bound to the shared utility instance, *OpticalChannelUtility*. This shared utility contains various functions for converting optical channel parameters with regard to the ITU-T 694.1 standard [78] to the *OchSignal* instance and back. You can have a look on brief overview of this standard in Appendix B.

Once all parameters are converted, we can invoke *RoadmService* instance and pass these parameters to the appropriate function *(createConnection)*. *RoadmService* instance is responsible for composing the "Flow Rule" from the passed parameters. Later, it passes

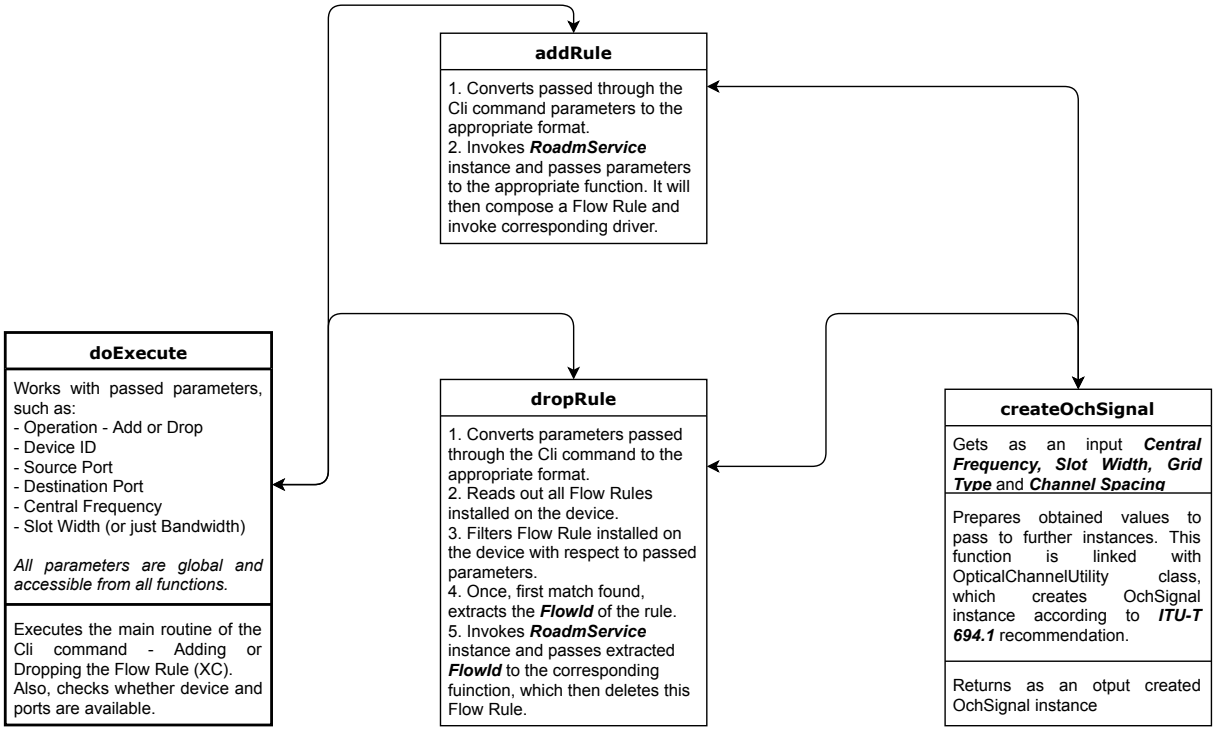

Figure 4.7: Cli command, *roadm-xc*, workflow

the "Rule" to the corresponding driver instance. In our case *RoadmService* would compose a "Flow Rule" and install it on the device by invoking *NokiaPssFlowRuleProgrammable* interface, particularly *addFlowRules* function. After "Flow Rule" is added to the device, it should be visible inside the ONOS terminal (invoke command *"flows"*).

In case we want to remove XC, a little bit different logic is implemented. We're getting input parameters and converting them again to the appropriate format in order to create a *dummy* "Flow Rule". After that, we read out all "Flow Rules" installed on the device and iterate over them in a loop. In each iteration we're comparing parameters of the "Flow Rule" installed on the device with the *dummy* "Flow Rule" we just composed. Once the full match is found, we read out the *FlowId* of matched "Flow Rule" and pass it to the appropriate function of *RoadmService* instance (*removeConnection*). It invokes *NokiaPssFlowRuleProgrammable* interface, particularly *removeFlowRules* function. After that the chosen "Flow Rule" should be deleted from the device.

Since all input parameters are global, they're accessible from any function. We don't need to pass them to the functions, except if we want to get their modified version.

Two help instances were created during the development of the *roadm-xc* command - *RoadmCrossConnectCompleter* and *OpticalChannelUtility*. First instance is responsible for completing feature. It offers the user to choose between *create* and *remove* options. *OpticalChannelUtility* is a shared utility instance, which was already described above. You can have a look on full code of mentioned instances here [85].

We're missing one last step to integrate this command inside the ONOS. To do this, we should add some dependencies into the *BUILD* file of the ROADM app. Here is a snippet of such file.

```
COMPILE_DEPS = CORE_DEPS + JACKSON + CLI + [ # Some changes here
    "//core/store/serializers:onos-core-serializers",
    "//apps/optical-model:onos-apps-optical-model",
]
TEST_DEPS = TEST_ADAPTERS + [
    "//utils/osgi:onlab-osgi-tests",
]
osgi_jar_with_tests(
    karaf_command_packages = ["org.onosproject.roadm.cli"], # Some
    ↪  changes here
    test_deps = TEST_DEPS,
    deps = COMPILE_DEPS,
)
```

Some additional changes are made in this file:

- Added "**karaf_command_package**" with link to the Java program implementing logic of the Cli command. It adds command into the *karaf* shell.
- Added "**JACKSON**" and "**CLI**" into the dependencies. It is important in consequence with previous point.

Now we can run clean installation of the ONOS and test the command.

This command was contributed to the ONOS project community. You can find this contribution here [85].

4.4 PowerConfig behavior integration

Next and probably the easiest step of the development is to integrate PowerConfig interface into the existing driver of Nokia 1830 PSI-2T device (Optical Transponder). This interface implements the power configuration capability for the optical transceiver devices. To enable the PowerConfig behavior, we should go to the already well-known file, *odtn-drivers.xml*, and add following lines to enable corresponding behavioral model on the device:

<behaviour api="org.onosproject.net.behaviour.PowerConfig"
impl="org.onosproject.drivers.odtn.NokiaTerminalDevicePowerConfig"/>

Now the driver will see which file it should execute, when the user wants to configure the power on the device. You may notice, that the file responsible for this functionality is called *NokiaTerminalDevicePowerConfig.java* [86].

As an inspiration for our implementation, existing power configuration solution of Cassini equipment, *CassiniTerminalDevicePowerConfig.java*, was taken.

You can find the workflow diagram of the PowerConfig behavior on Figure 4.8 with brief description of each function. Some values taken from the device datasheet were hardcoded. If you have a look on **getTargetPowerRange** and **getInputPowerRange**, you'll notice that they contain device power range, which is hardcoded. The main purpose is to limit the range of the values, which you can pass to the device, basically ensuring passing of the values in correct diapason.

The main challenge during the whole process consisted in the the way of parsing of obtained configuration through the internal Java libraries. Also, composing the RPC

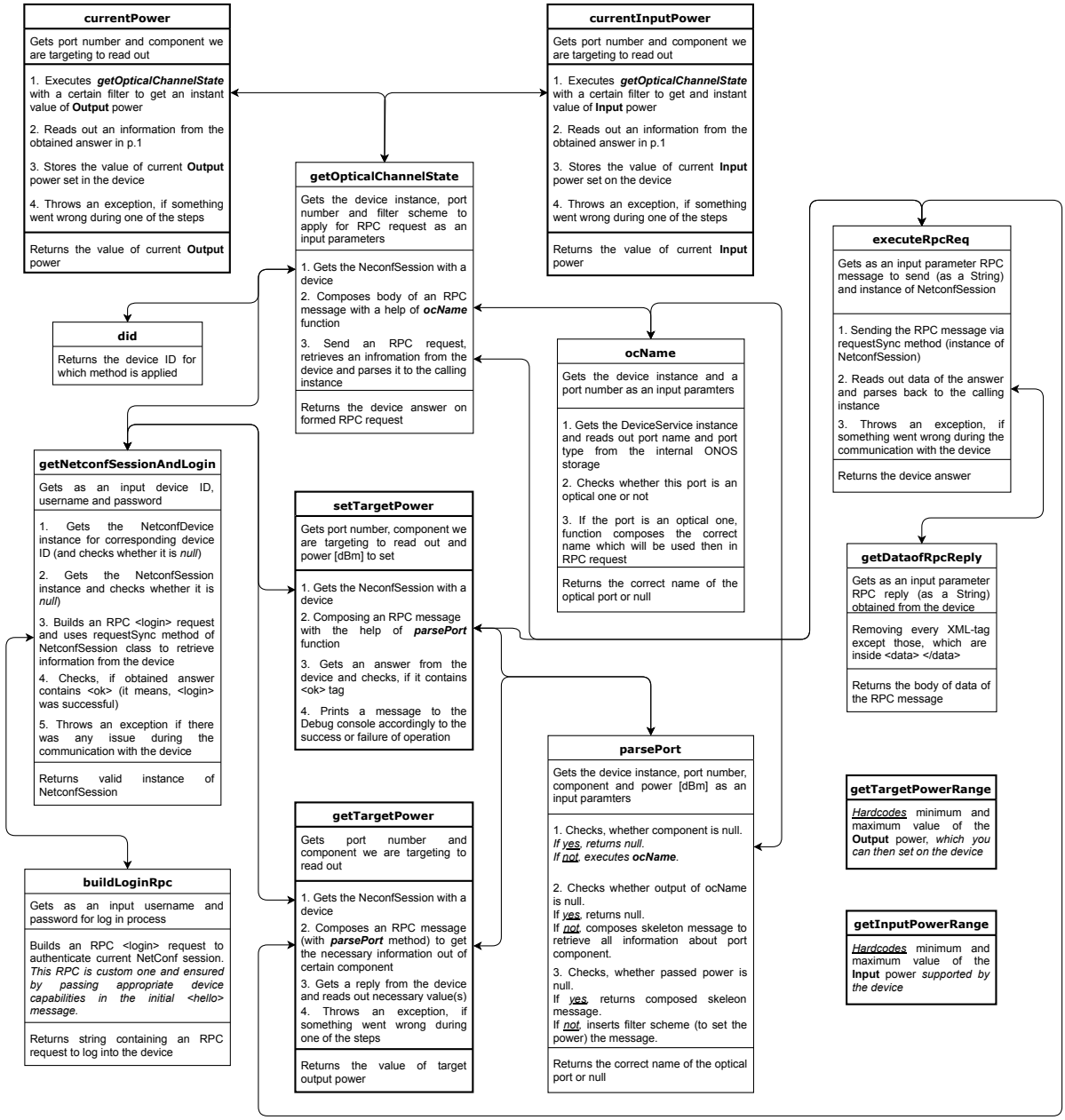

Figure 4.8: PowerConfig interface workflow

message to obtain required parameters is needed. For each device custom data model implementations could slightly differ.

Next challenging part during the deployment of this interface came from the device part. Nokia 1830 PSI-2T contains not only "Optical Channel" type of the ports, but also "Packet" type of the ports. In PowerConfig interface we're targeting to set the power only on the "Optical channel" ports, but not on the "Packet" type of the ports. This situation could provoke returning of a *null* values from some functions. It leads the interface implementation to behave in an incorrect way. For example, during the testing not all ports were able to simultaneously refresh their values. Moreover, some ports were showing the "null" value, even if the value was set and present in the device.

To avoid this, several enhancements on the PowerConfig behavior implementation were done:

- We check each port on it's type and continue processing only "Optical Channel" type of the ports, otherwise return we *null* state;
- It is necessary to catch all possible *null* states in order to reduce unnecessary computations by the controller.

After that was done, PowerConfig behavior worked more stable and displayed all values at the same time. This interface continuously reads out the data from the device. It is done due to the laser values fluctuation. Once per period, usually 4-5 seconds, ONOS controller sends a NETCONF **<get-config>** RPC request in order to read out power values from the specific port. Obtained from the device reply is then parsed and visualised through the *"Optical UI"* interface.

We can find implemented functionality under *"Optical UI"* tab by clicking on the port icon in the right top corner. To see how does it look like, please visit Appendix C. Implementation of PowerConfig interface also enables power configuration through the following Cli command:

power-config get/edit-config connectionPoint value

where:

- ***"get"*** option specifies *"request"* action. It tells to read out a target power value from the device's specific port.
- ***"edit-config"*** option specifies *"set"* action. It sets target power value on the device's specific port.
- ***"connectionPoint"*** specifies connection point, where we should set the power. Information must be passed here in format ***{DeviceId}/{PortNumber}***.
- ***"value"*** is a target power value in [dBm].

You can find full implementation contributed to the ONOS project community here [86].

4.5 AlarmConfig behavior integration

As a part of upcoming work on Alarm Correlation in Nokia Bell Labs, AlarmConfig behavior was required to be implemented. It is responsible for treating alarms coming from the device. It translates alarms from the OPENCONFIG format and stores them inside the ONOS.

You can find the workflow diagram of this interface implementation on Figure 4.9.

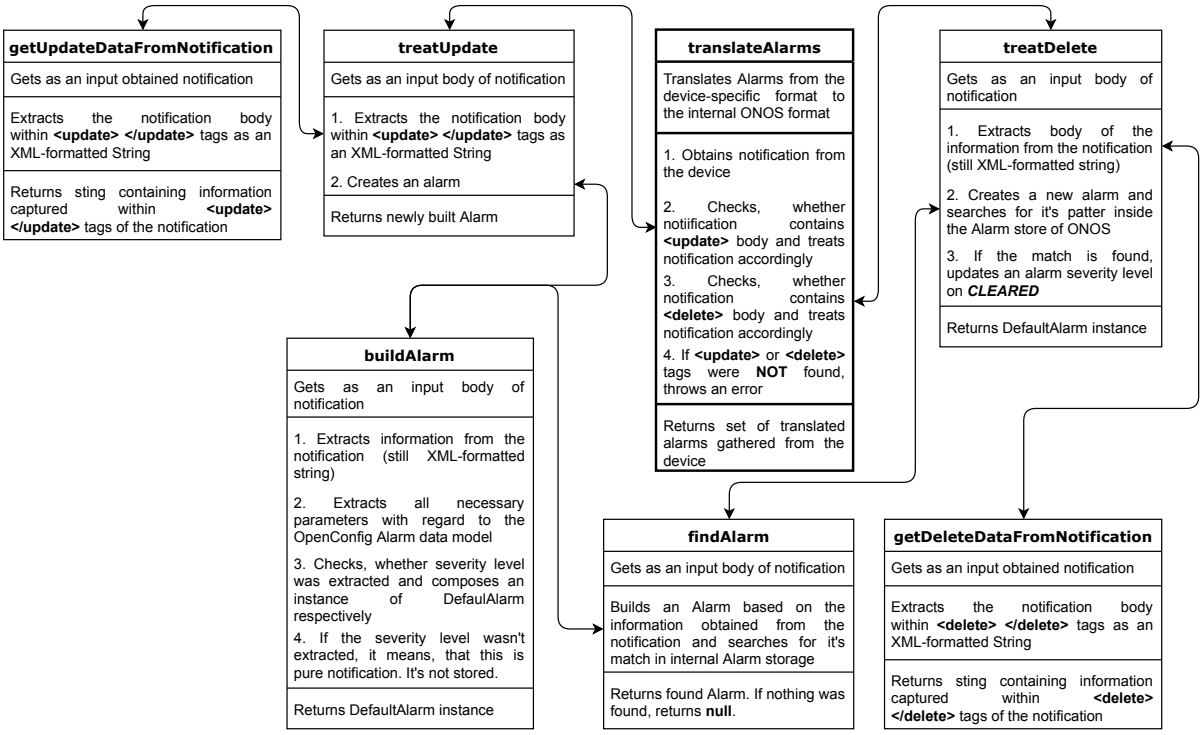

Figure 4.9: AlarmConfig interface workflow

According to the OPENCONFIG data model, alarms have following format:

```
alarms
  alarm
    id
    config
    state
      id
      resource
      text
      time-created
      severity
      type-id
```

This interface has only one function, **translateAlarms**, which does the whole routine. In order to ease the code and avoid duplicating, we decomposed it on a set of smaller functions. In **translateAlarms** we're treating upcoming notification from the device and checking whether it is:

- an ***<update>*** type of the event, what corresponds to the newly created alarm.
- ***<delete>*** type of the event, what indicates that alarm is expired (solved, not valid anymore).

Based on the type of the event corresponding set of functions is being triggered.

In case we want to create a new alarm, we go to the **treatUpdate** function. Body within the <update></update> tags is extracted with the help of the **getUpdate-DataFromNotification** function (it cuts off everything outside the body of <update> tags).

Once it's been done, **buildAlarm** function is invoked. It extracts all necessary information from the XML String, creates a new *Alarm* instance and stores it in ONOS.

In case we want to clear an alarm, we go to the **treatDelete** function. With the help of **getDeleteDataFromNotification** function, body within the <delete></delete> tags is extracted. After that this information is passed to the **findAlarm** function. Main purpose of this function is to find in the related to the device set of alarms the one with the same parameters. To do this, we create a pattern of an alarm with the **buildAlarm** function and try to match it with alarms collected from the device. Once the match is found, we update an alarm by putting a ***clear*** flag.

In order to start using this interface we should slightly modify the existing driver of Nokia 1830 PSI-2T and a new driver of Nokia 1830 PSS. In the body of **discoverPortDetails** function we should insert following lines:

```
try {
    ns.startSubscription();
    log.info("Started subscription");
} catch (NetconfException e) {
    log.error("NETCONF exception caught on {} when the subscription
     ↪  started \n {}",
            data().deviceId(), e);
}
```

In that way we start NETCONF subscription. Client (SDN controller in our case) collects device notifications within the current NETCONF session. By default, NETCONF subscription puts Client into the listener mode and restricts it from sending any further NETCONF requests, what makes not possible to manage the device within the same NETCONF session. Fortunately, NETCONF subscription in ONOS is implemented with interleaving feature. It means that we would be able to manage the device even if there is a subscription running on the current session.

This interface was contributed to the ONOS project community. You can find it here [87]. Developed interface is universal. It is valid for both devices, Nokia's ROADM and Nokia's Optical Transponder.

4.6 Sample Alarm Correlation application development

After implementing AlarmConfig behavior and collecting alarms from the devices, we could put some logic on top of it. Next few sub-sections are dedicated to the implementation of an application. It is capable of alarm analysis and signalizing, if there is any issue in the network. Developed application is targeting to be a part of a SDN controller. Communication with the devices is done over the NETCONF protocol.

4.6.1 Analyzed scenarios

Before diving into the application development we concentrated on the analysis of a *"fiber cut"* scenario. It is necessary to understand, how we could detect and localize failure based on the alarms obtained from the different devices. After that we could start putting logic into application.

We worked with the same device bench in the lab. Topology is depicted on Figure 4.10.

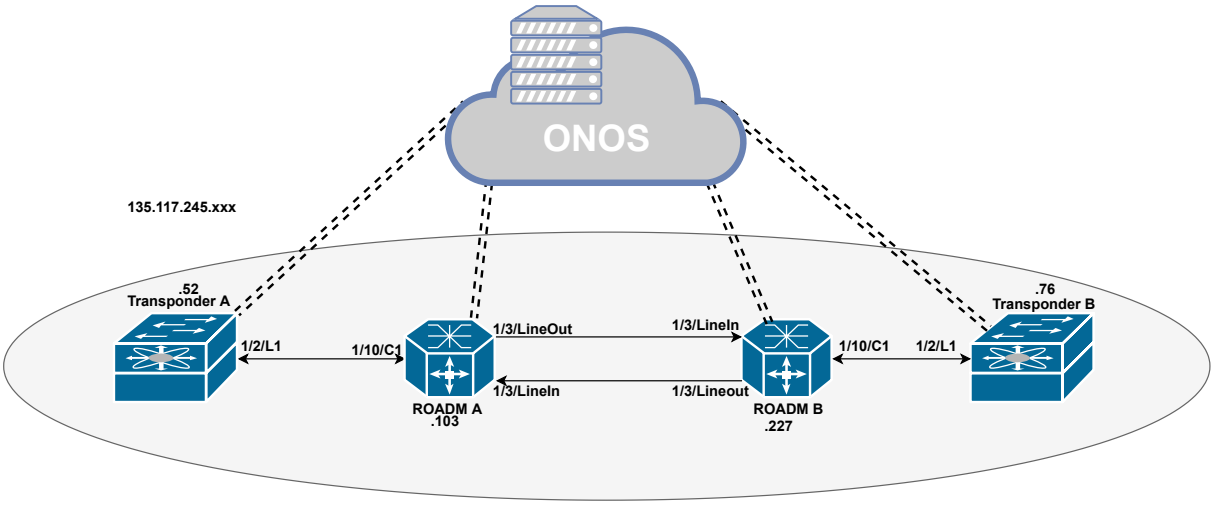

Figure 4.10: Topology in the lab

Several tests simulating *"fiber cut"* scenario were performed. We unplugged optical fiber between each pair of the devices. Following scenarios were covered:

- Optical Transponder - ROADM
  - Unplug one fiber on the Transponder side.
  - Unplug both fibers on the Transponder side.
  - Unplug one fiber on the ROADM side.
  - Unplug both fibers on the ROADM side.
- ROADM - ROADM
  - Unplug one fiber (on the degree port) at a time.
  - Unplug both fibers (on the same degree port) at the same time.

Later on, collected over NETCONF subscription alarms were analyzed. We detected two following scenarios, covering *"fiber cut"* problematic.

First scenario is Optical Transponder - ROADM case. It is depicted on Figure 4.11.

Second scenario is ROADM - ROADM case. It is depicted on Figure 4.12.

In both scenarios we catch two *"Loss Of Signal"*-type of the alarms on interconnected ports. It indicates to us that there is a *"fiber cut"*. With regard to this rule we can make a detection.

Other types of the alarms were caught as well. Most of them were stating that there is a *"Loss Of Frame"*, *"External Interface Failure"* or something else happening in the network. These types of the alarms are not that useful for a failure localization.

Generalizing this approach, we need to correlate two parameters:

- Alarms should be *"Loss Of Signal"*-type.
- These alarms should be caught on two interconnected ports within the same link.

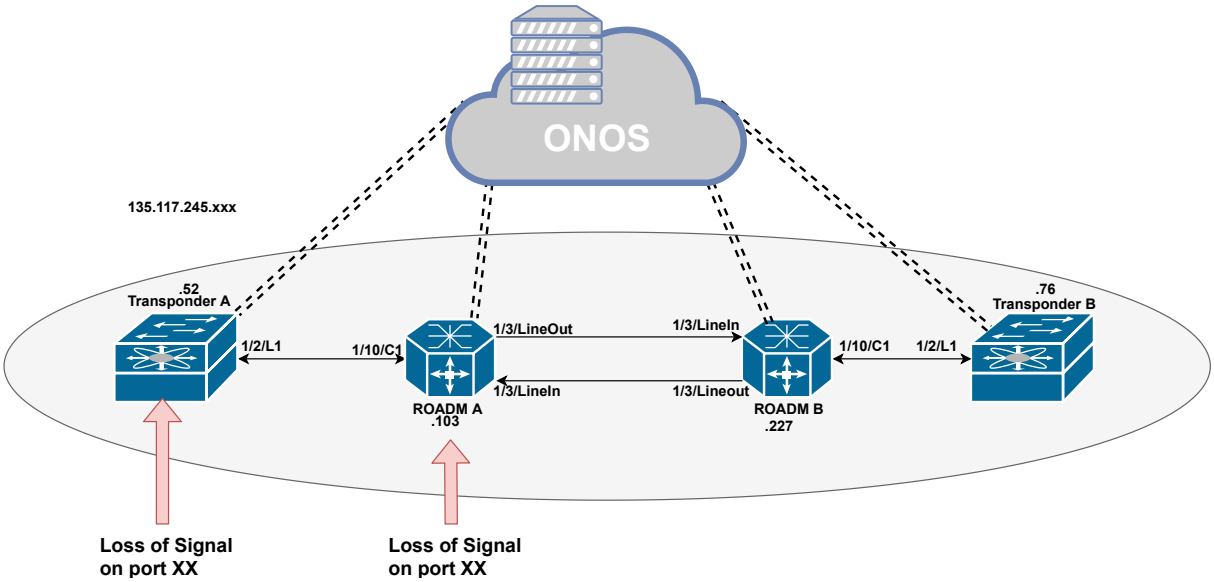

Figure 4.11: Alarm Correlation. Scenario 1

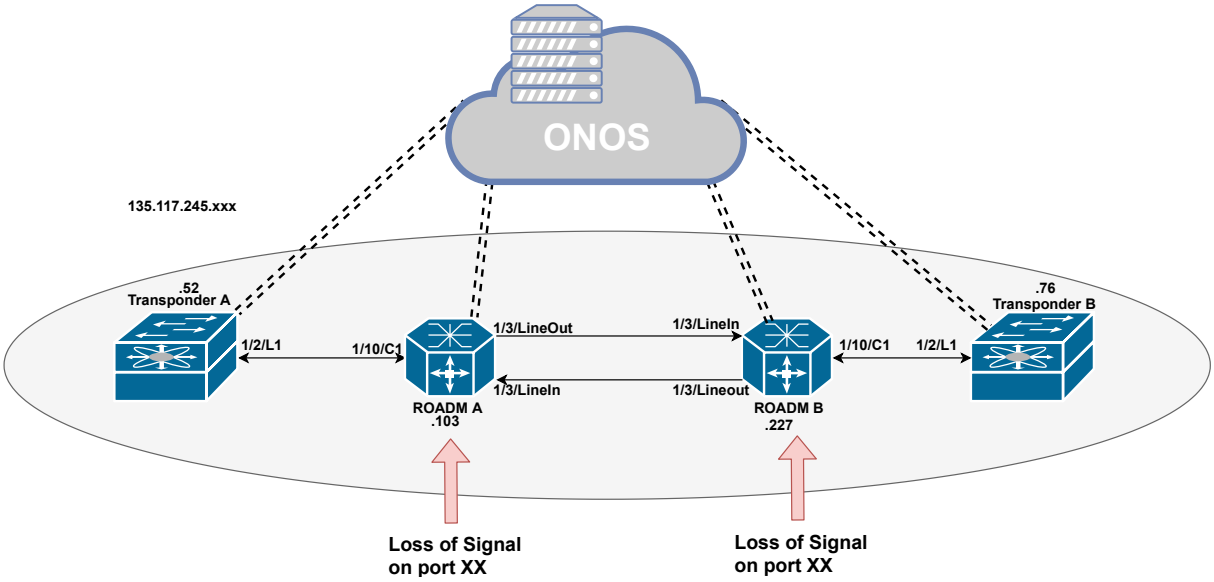

Figure 4.12: Alarm Correlation. Scenario 2

4.6.2 Application development

Now, after target approach is defined, we can start thinking on it's implementation as a SDN application.

Network topology is stored in ONOS as a set of links. Each link contains information about source Connection Point, destination Connection Point and link metrics. Connection point (CP) is composed from the Device ID and Port Number.

Also, ONOS already contains Path Computation Element (PCE) function. We propose to utilize it for failure localization and path restoration purposes. PCE function is based by default on Dijkstra[5] algorithm and interacts with a Topology subsystem of the controller.

PCE is capable of searching for a path between two nodes (devices). Found path between two devices contains set of links. Search could be based on different algorithms.

Utilizing these tools could ease the development process. General algorithm for *"fiber cut"* detection could look like:

- Get the path between two nodes;
- Iterate over each link:
  - Check alarm store of **source** CP on presence of *"Loss of Signal"*-type of the alarm on current CP's port.
  - Check alarm store of **destination** CP on presence of *"Loss of Signal"*-type of the alarm on current CP's port.
    - If both ends of the link contain *"Loss of Signal"*-type of the alarm, then "fiber cut" was detected and localized.
      - In that case we should disable the link to exclude it from further path computations.
    - Otherwise, continue iteration over the links in the path and check each link on the presence of two *"Loss of Signal"* alarms.

Once failed link is identified, we should reconfigure optical network to restore impacted optical channels. To fulfill this task we can rely again on PCE function. Since cut link was disabled and excluded from the network topology used by PCE, we could easily find an alternative path between two nodes. Afterwards we should reconfigure each device in this new path. In context of optical transport networks we mostly have two types of the devices:

- Optical Transponder, where we need to configure:
  - Wavelength on the port;
  - Power on the port.
- ROADM, where we need to configure:
  - Cross-connectivity between certain pairs of ports.

Each device requires specific interaction and configuration of specific parameters. In order to automate (re)configuration of the device, we can create a shared utility containing specific functions capable of (re)configuring the device. In multi-vendor environment we have to have more of such utilities, where each would implement functions responsible for different vendor device reconfiguration.

Since we studied only *"fiber cut"* scenario, different cases could bring more complexity inside the structure of this Alarm Correlation application. Different cases could introduce different parameters for correlation. In our prospective a good solution could be to wrap functionality related to the detection of a specific failure state in an utility. Later on each next failure scenario could be implemented as a separated shared utility. It makes the structure more modular-like and easy to track.

With regard to the aforesaid architecture of the Alarm Correlation application was developed. It is depicted on Figure 4.13.

[5]ONOS implements a lot of different algorithms for path search. You can always swap Dijkstra algorithm with the one which fits your requirements the best.

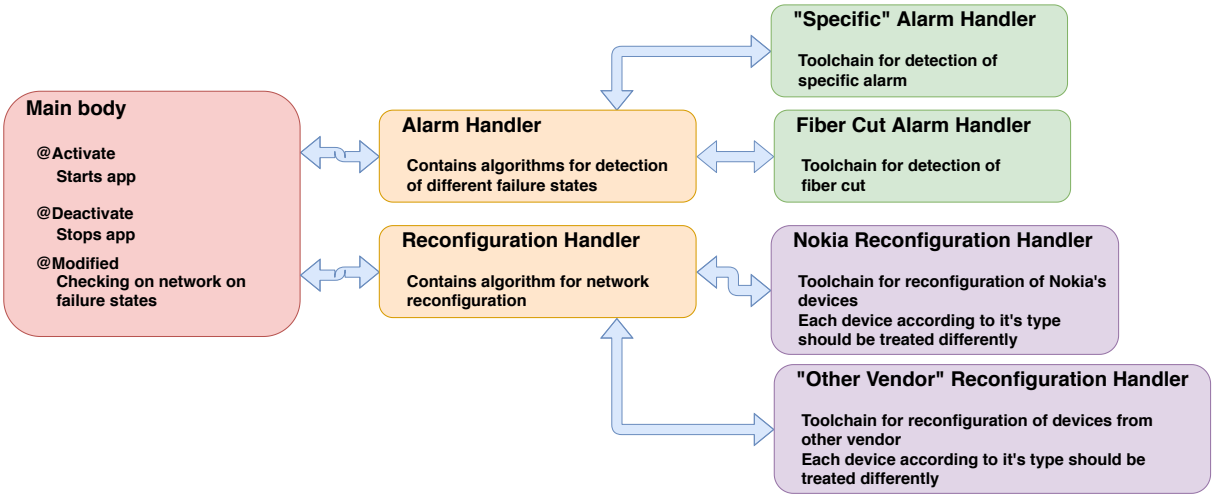

Figure 4.13: Alarm Correlation application architecture

Within this architecture we tried to introduce following modular approach:

- **Main body** of the program have three states:
  - **@Activate** - starts the application.
  - **@Deactivate** - ends application correspondingly.
  - **@Modified** - method, which reacts on dynamic changes of the network parameters (i.e. network state). From there we would like to trigger routine for checking the network on different failure states and then corresponding reconfiguration of the network.
- **Alarm Handler** is an instance capable of detecting different failure states in the network. Contains various algorithms for detection of different failures. It is linked to the following instances:
  - **Fiber Cut Alarm Handler** contains set of functions necessary for *"fiber cut"* scenario detection. Based on these functions algorithm in **Alarm Handler** for *"fiber cut"* detection is built.
  - **"Specific" Alarm Handler** contains set of functions for other failure scenario detection. **Alarm Handler** later implements algorithm for this specific case detection based on the functions provided by this utility.
- **Reconfiguration Handler** contains algorithm (and some supportive functions) for (re)configuring path in the network. It is linked to following instances:
  - **Nokia Reconfiguration Handler** contains set of functions capable of (re)configuring different types of Nokia's optical devices.
  - **"Other Vendor" Reconfiguration Handler** contains set of functions capable of (re)configuring different types of optical devices from different vendor.

4.6.3 Implementing sample application

Once an approach is defined, we could start implementing this application. First of all we should create a new directory under ***$ONOS_ROOT/apps/***. It would contain files

of application. Full path to our application now looks like:

$ONOS_ROOT/apps/alarmcorrelation/

In ***/alarmcorrelation*** directory we should put a BUILD file. It contains instructions for application building. Here you could also bind pieces of functionality from different subsystems of the controller. Skeleton BUILD file should look like this:

```
COMPILE_DEPS = CORE_DEPS + [
    "//core/net:onos-core-net",  # enables ONOS core net packages for
    ↪ managing links
    "//apps/faultmanagement/fmcli:onos-apps-faultmanagement-fmcli",  #
    ↪ Enables AlarmManager stuff
    ↪ "//apps/faultmanagement/fmmgr:onos-apps-faultmanagement-fmmgr-native",
    "//apps/roadm/app:onos-apps-roadm-app",  # Enables RoadmService
    ↪ instance
    "//apps/optical-model:onos-apps-optical-model-native",  # Enables the
    ↪ use of OpticalChannelUtility
    ...
    # And many other dependencies
]

osgi_jar_with_tests(
    deps = COMPILE_DEPS,
)

onos_app(
    app_name = "org.onosproject.alarmcorrelation",
    category = "Traffic Engineering",
    description = "This is proposition of application, which is capable
    ↪ of " +
                  "detecting failure states in the optical network. " +
                  "By now, only fiber cut scenario is implemented.",
    title = "Alarm Correlation",
    url = "http://onosproject.org",
)
```

After that, in order to include our application into the full build of the ONOS, we should add one more dependency inside the file ***tools/build/bazel/modules.bzl***.

```
ONOS_APPS = [
    # Apps
    "//apps/alarmcorrelation:onos-apps-alarmcorrelation-oar",
    ...
]
```

Now we can implement functionality with regard to the proposed architecture. All files responsible for it's functionality must be stored in:

../alarmcorrelation/src/main/java/org/onosproject/alarmcorrelation/

Inside this folder we created following files:

- **AlarmCorrelation.java** - main body of the program;
- **AlarmHandler.java** - instance containing algorithm for *"fiber cut"* detection, respectively for any other scenario.
- **FiberCutAlarmHandler.java** - instance containing set of functions helping to detect *"fiber cut"* scenario.
- **package-info.java** - contains license information and short information about this package.

As you may notice, we didn't specify here (re)configuration part, since it wasn't tested. Implemented algorithm you can find here [88]. Workflow of this application was tested on Cassini emulators (please refer to the section 5.2).

4.6.4 In conclusion

During the development of the application we made following assumptions:

- PCE function is perfect and knows everything about the network.
  - Default metric consists in computing number of *"hops"*. It is possible to change this metrics on custom one.
- Shortest path is computed and configured between each pair of the nodes in the network.
- Detecting two *"Loss of Signal"* alarms on interconnected ports is enough for stating, that the fiber has been cut.
  - *"Loss of Signal"* alarm could have various causes. It is also necessary to keep track of history of different parameters (like transmitting/receiving power and etc.) on interconnected ports. State of the laser (active/inactive) should be also correlated.

These assumptions are limiting. In practice, the shortest path between two nodes is not necessary the best one and used metrics could differ from number of *"hops"* between two nodes.

This application is a proposition of a concept, which, for sure, must be enhanced in the future. It was developed for single-vendor environment. Multi-vendor environment could bring additional complexity to this problematic. We should know operating principles of each device in order to state about the origin of the failure, even if types of the alarm are standardized by OPENCONFIG [89], [90]. This understanding is also necessary, when it goes to (re)configuration part. Moreover, it is necessary to have an utility which is capable of (re)configuring each specific device in multi-vendor environment. More investigation in multi-vendor case should be done.

Following modular approach, application could be extended on other utilities. For example, such module could implement specific Machine Learning (ML) technique for predicting specific failure state in the network. Other utility could implement other ML algorithm for predicting other failure state in the network. Implementing of ML techniques is always complex. We should clearly understand how we can benefit from certain algorithm and which data we should pass as an input. This is a big field for further research.

It would be also beneficial to measure delay between the time, when actual failure state

has happened, and a moment, when this state was detected by the application. Within several tests performed on better understanding of *"fiber cut"* scenario, difference between first captured alarm and the last captured alarm was few seconds.

In conclusion, this application is a proposal of how does alarm correlation function could look like. A lot of work still needed to be done in order to improve this application and make it able to handle other network failure cases, not only one specific failure scenario.

Chapter 5

Validating results

This chapter presents practical application of developed features. In the first part of this chapter steps for end-to-end optical channel connectivity validation are described. In the second part of this chapter, I tried to test Alarm Correlation application on emulated optical devices.

5.1 Configuring end-to-end optical channel connectivity

This section provides description of an end-to-end optical channel connectivity validation scenario. Device's bench, network setup and configuration steps are described as well.

5.1.1 Scenario description

For validation of end-to-end optical channel connectivity simple topology setup depicted on Figure 5.1 was used.

Devices used in this topology are following[1]:

- Transponder A - Nokia 1830 PSI-2T;
- ROADM A - Nokia 1830 PSS-16 II;
- ROADM B - Nokia 1830 PSS-16 II;
- Transponder B - Nokia 1830 PSI-2T;

Firstly, we should create cross-connection corresponding to the interconnected ports on the ROADMs. After that, we should configure the Power (and frequency) on Optical Transponder.

Once it's done, we connect via the SSH to both switches and execute ping command to validate establishing of end-to-end optical channel connectivity. Both switches are in the same sub-network - ***10.2.0.x***.

ONOS instance is running on a *localhost* of a laptop.

[1]These devices were introduced in Section 4.1.2 and Section 4.1.3 of chapter four.

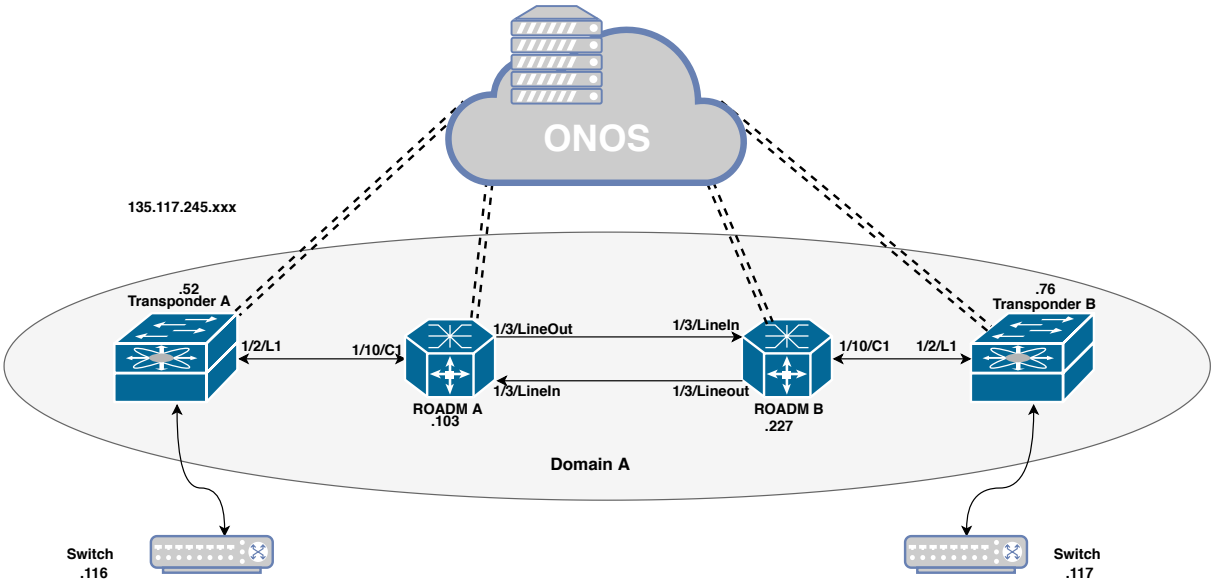

Figure 5.1: Testing topology

5.1.2 Configuration steps

This section briefly describes steps for running ONOS and configuring network devices via the Cli (and WebUI) of ONOS.

- Start ONOS with ***ok clean*** command.
- Connect to the Karaf (ONOS command line interface) with ***onos localhost*** command.
- Set ONOS Flow Rule Subsystem with following commands:

  cfg set org.onosproject.net.flow.impl.FlowRuleManager allowExtraneousRules true

  cfg set org.onosproject.net.flow.impl.FlowRuleManager importExtraneousRules true

  cfg get org.onosproject.net.flow.impl.FlowRuleManager

  It would force ONOS to store Flow Rules already installed on the device. You can verify it later, once devices are registered, with the command ***flows***.

- Register devices and links via REST API of ONOS (***onos-netcfg*** *localhost devices.json* command).
- Check from the Karaf shell, that devices and links are registered. Use approptiate commands ***devices*** and ***links***.
- Check Flow Rules installed on the devices by typing ***flows*** in a command line.

Now we can start configuration of the ROADMs.

- With help of a Cli command developed during this work, we can easily manage cross-connections on the ROADM side.
  - Configure unidirectional XCs between ports 1/10/C1 and 1/4/LineOut and 1/3/LineIn and 1/10/C1 on both ROADMs. ***Wavelength*** and ***Slot Width*** parameters should be the same in all link.
- Configure wavelength on the transponder side with following command:

  wavelength-config edit-config *DeviceID/PortNumber wavelength*
  - Somehow, there is one disadvantage. After executing this command, corresponding port of the device would be deactivated. You need to go to the WebUI of the device and activate this port again. After that you will be able to set transmitting power.
- You can set the transmitting power through ***Optical UI*** tab inside the ONOS WebGUI. Go to the ***Optical UI***, choose device you wish to set the power (Optical Transponder in our case) and click on the ***Ports*** icon. In a while you'll get an information about the power value on each port. You can dynamically change transmitting power via this tab.
  - In alternative way you can set power via following Cli command:

    power-config edit-config connectionPoint value

5.1.3 Validation

Once steps described in previous sub-section are done, login to the routers and validate end-to-end optical channel connectivity with a ping. You can also go to the WebUI of both Optical Transponders and validate that an optical link was established by checking amount of receiving energy on the corresponding Line ports of each transponder.

```
*A:SR#1# ping 10.2.0.2 count 150
PING 10.2.0.2 56 data bytes
64 bytes from 10.2.0.2: icmp_seq=1 ttl=64 time=0.385ms.
64 bytes from 10.2.0.2: icmp_seq=2 ttl=64 time=0.339ms.
64 bytes from 10.2.0.2: icmp_seq=3 ttl=64 time=0.382ms.
64 bytes from 10.2.0.2: icmp_seq=4 ttl=64 time=0.381ms.
```

Figure 5.2: Ping from first router

```
*A:SR#2# ping 10.2.0.1 count 150
PING 10.2.0.1 56 data bytes
64 bytes from 10.2.0.1: icmp_seq=1 ttl=64 time=0.623ms.
64 bytes from 10.2.0.1: icmp_seq=2 ttl=64 time=0.625ms.
64 bytes from 10.2.0.1: icmp_seq=3 ttl=64 time=0.613ms.
64 bytes from 10.2.0.1: icmp_seq=4 ttl=64 time=0.631ms.
64 bytes from 10.2.0.1: icmp_seq=5 ttl=64 time=0.589ms.
64 bytes from 10.2.0.1: icmp_seq=6 ttl=64 time=1.20ms.
```

Figure 5.3: Ping from second router

On Figure 5.2 and Figure 5.3 are represented successful ping between two devices. On Figure 5.4 you can see presence of the power in the channel after all configuration steps were done. Also, you can find how does the process of setting power on optical port through ***Optical UI*** look like on Figure C.1.

5.2 Alarm correlation scenario

This section presents validation of Alarm Correlation function developed as a SDN application.

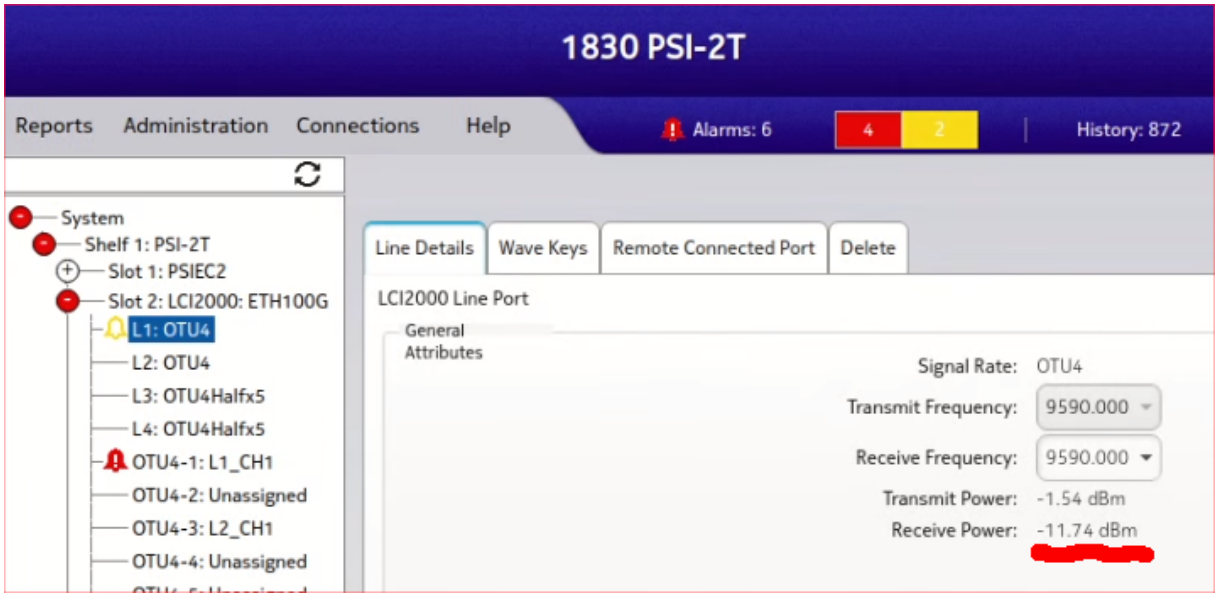

Figure 5.4: Power presence in the channel

5.2.1 Scenario description

Workflow of the developed application was tested on Cassini emulator. It is emulated optical Cassini equipment wrapped in a docker container. You can find some guidance on how to operate with it here [91]. With the help of this emulator we created testing topology depicted on Figure 5.5.

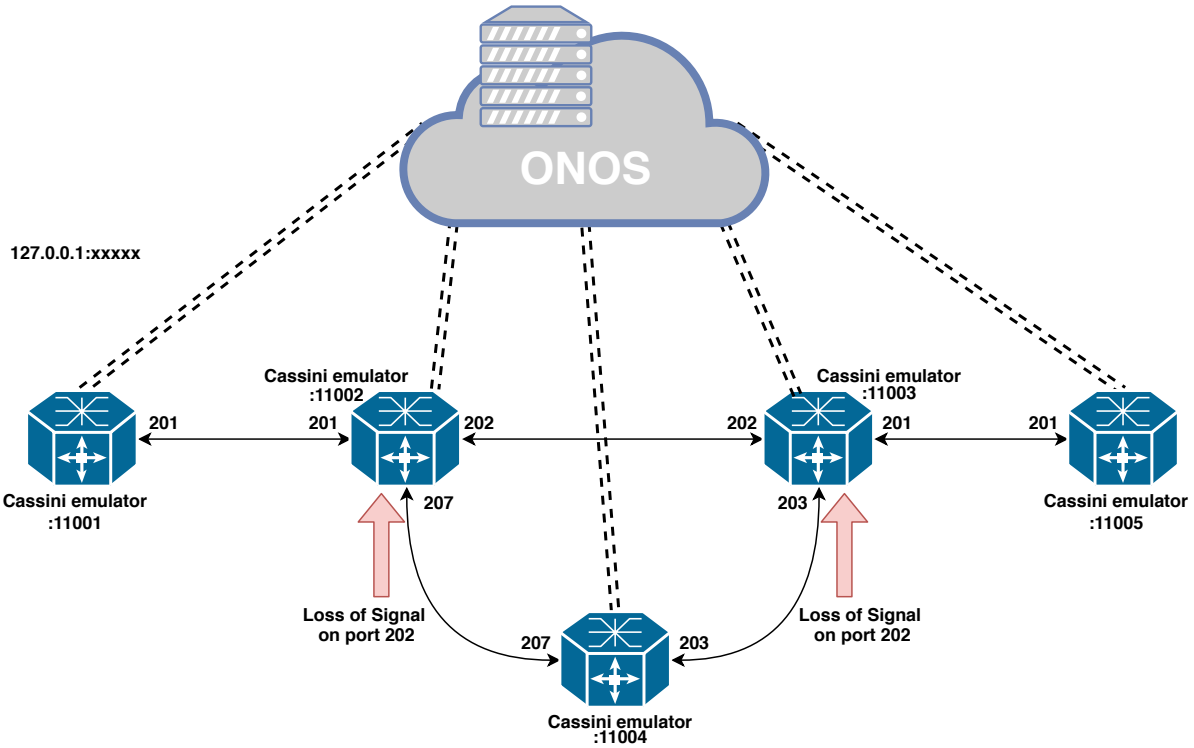

Figure 5.5: Testing topology for Alarm Correlation application

To test the algorithm, I slightly changed the **Main body** of developed application. Before doing check on *"fiber cut"* scenario I inserted in the alarm store two *"Loss of Signal"*-type of the alarms on ports 202 of devices **:11002** and **:11003**. Once it's done, alarm correlation function should detect *"fiber cut"*.

5.2.2 Configuration steps

In order to fake algorithm, we must insert Loss of Signal type of the alarms. It was done with the help of following two functions implemented as a part of *FiberCutAlarmHandler.java*.

```
public static void insertLosAlarms() {
    AlarmStore alarmStore = AbstractShellCommand.get(AlarmStore.class);
    Alarm alarm1 = generateLosAlarm("1",
     ↪  DeviceId.deviceId("netconf:127.0.0.1:11002"),
                                    "202", 12);
    Alarm alarm2 = generateLosAlarm("2",
     ↪  DeviceId.deviceId("netconf:127.0.0.1:11003"),
                                    "202", 15);
    alarmStore.createOrUpdateAlarm(alarm1);
    alarmStore.createOrUpdateAlarm(alarm2);
}
public static Alarm generateLosAlarm(String id, DeviceId deviceId,
                                    String source, long timeStamp) {
    return (new DefaultAlarm.Builder(AlarmId.alarmId(id),
                                    deviceId, "Loss of Signal:" + source,
                                    Alarm.SeverityLevel.CRITICAL,
                                     ↪  timeStamp).
            withServiceAffecting(true).build());
}
```

First one insert generated alarms in the alarm store, second on genrates *"Loss of Signal"*-type of the alarm. After that we inserted in the main body of the application following lines:

```
private static final DeviceId DEVICE_ID_SOURCE =
 ↪  DeviceId.deviceId("netconf:127.0.0.1:11001");
private static final DeviceId DEVICE_ID_DESTINATION =
 ↪  DeviceId.deviceId("netconf:127.0.0.1:11005");
    ...
//Faking (inserting LoS alarms inside the storage)
FiberCutAlarmHandler.insertLosAlarms();
delay(5000);
Link l =
    AlarmHandler.detectFiberCut(ReconfigurationHandler.getPath(DEVICE_ID_SOURCE,
 ↪  DEVICE_ID_DESTINATION));
// Once alarm is detected, disabling the link
disableLink(l.src(), l.dst());
```

This piece of code initializes two nodes, source and destination, between which the path would be computed. **insertLosAlarms()** function inserts alarms inside the alarm store of the ONOS. **detectFiberCut()** method triggers routine to detect failure state. As a

parameter you need to pass the network path to check. Function **getPath()** there returns network path between two nodes in the network. In the end, we disable link which was returned as the one with fiber cut. Piece of code disabling the link is provided below.

```
private void disableLink(ConnectPoint cpSrc, ConnectPoint cpDst) {
    log.info("\n\n [disableLink] Deleting link between {} and {} \n",
     ↪  cpSrc, cpDst);
    linkProviderService.linksVanished(cpSrc);
    linkProviderService.linksVanished(cpDst);
}
```

Application is modified and ready to be tested. Now we need to set up the network topology and register it inside the ONOS. To do this, simple bash script was created.

```
#!/bin/bash
sudo docker run -it -d --name odtn-emulator_openconfig_cassini_1_1 -p
 ↪  11001:830 onosproject/oc-cassini:0.21
echo "First device is created"
sudo docker run -it -d --name odtn-emulator_openconfig_cassini_2_1 -p
 ↪  11002:830 onosproject/oc-cassini:0.21
echo "Second device is created"
sudo docker run -it -d --name odtn-emulator_openconfig_cassini_3_1 -p
 ↪  11003:830 onosproject/oc-cassini:0.21
echo "Third device is created"
sudo docker run -it -d --name odtn-emulator_openconfig_cassini_4_1 -p
 ↪  11004:830 onosproject/oc-cassini:0.21
echo "Fourth device is created"
sudo docker run -it -d --name odtn-emulator_openconfig_cassini_5_1 -p
 ↪  11005:830 onosproject/oc-cassini:0.21
echo "Fifth device is created"

onos-netcfg localhost devices.json
echo "Devices are registered"
sleep 3
onos-netcfg localhost link.json
echo "Links are registered"
```

As you may notice, last two commands are linked to files ***devices.json*** and ***link.json***. These files contain information about the devices and links correspondingly. They were composed with regard to our target topology. To give a better idea following snippets provide a piece of these files.

```
{
  "devices" : {
    "netconf:127.0.0.1:11001" : {
      "basic" : {
        "name":"cassini1",
        "driver":"cassini-openconfig"
      },
      "netconf" : {
        "ip" : "127.0.0.1",
        "port" : "11001",
        "username" : "root",
        "password" : "root",
```

```
            "idle-timeout" : "0"
                }
            },
    }
}

    ...

{
  "links": {
    "netconf:127.0.0.1:11001/201-netconf:127.0.0.1:11002/201": {
      "basic": {
        "type": "OPTICAL",
        "metric": 1,
        "durable": true,
        "bidirectional": true
              }
        },
    }
}
```

All what is left to do is just to start ONOS, register topology with a script provided above and execute application with following command:

app activate org.onosproject.alarmcorrelation

After that checkout logs of the ONOS.

5.2.3 Validation

Results of the test are represented below. With the help of bash script, registered topology looks like on Figure 5.6.

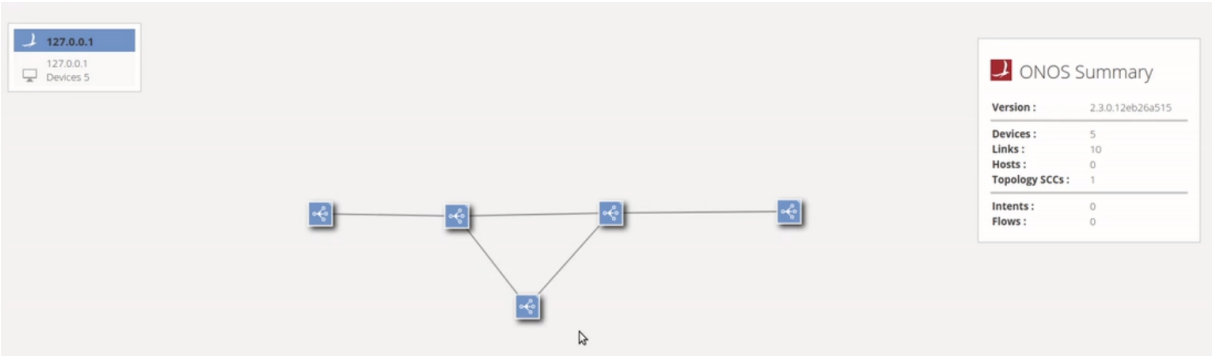

Figure 5.6: Testing topology for Alarm Correlation application. ONOS GUI.

In the ONOS logs on Figure 5.7 we can see that algorithm successfully detected failure state and deactivated failed link.

In the ONOS command line we can observe changes in state of the link represented on Figure 5.8. Same information we can observe in ONOS WebGUI (see Figure 5.9).

```
2019-12-20T13:50:53,834 | INFO  | features-3-thread-1 | AlarmHandler                    | 251 - org.onosproject.onos-apps-alarmcorrelation - 2.3
.0.SNAPSHOT |

 [AlarmHandler - detectFiberCut] Fiber cut was detected!

2019-12-20T13:50:53,834 | INFO  | features-3-thread-1 | AlarmCorrelation                | 251 - org.onosproject.onos-apps-alarmcorrelation - 2.3
.0.SNAPSHOT |

 [disableLink] Deleting link between netconf:127.0.0.1:11002/202 and netconf:127.0.0.1:11003/202

2019-12-20T13:50:53,835 | INFO  | features-3-thread-1 | LinkManager                     | 190 - org.onosproject.onos-core-net - 2.3.0.SNAPSHOT |
 Link DefaultLink{src=netconf:127.0.0.1:11002/202, dst=netconf:127.0.0.1:11003/202, type=OPTICAL, state=INACTIVE, expected=true} removed/vanished
2019-12-20T13:50:53,835 | INFO  | features-3-thread-1 | LinkManager                     | 190 - org.onosproject.onos-core-net - 2.3.0.SNAPSHOT |
 Link DefaultLink{src=netconf:127.0.0.1:11003/202, dst=netconf:127.0.0.1:11002/202, type=OPTICAL, state=INACTIVE, expected=true} removed/vanished
2019-12-20T13:50:53,836 | INFO  | features-3-thread-1 | AlarmCorrelation                | 251 - org.onosproject.onos-apps-alarmcorrelation - 2.3
.0.SNAPSHOT |
```

Figure 5.7: Logs of the ONOS after Alarm Correlation application was executed

```
eroshkin@root > links
src=netconf:127.0.0.1:11001/201, dst=netconf:127.0.0.1:11002/201, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11002/201, dst=netconf:127.0.0.1:11001/201, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11002/202, dst=netconf:127.0.0.1:11003/202, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11002/207, dst=netconf:127.0.0.1:11004/207, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11003/201, dst=netconf:127.0.0.1:11005/201, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11003/202, dst=netconf:127.0.0.1:11002/202, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11003/203, dst=netconf:127.0.0.1:11004/203, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11004/203, dst=netconf:127.0.0.1:11003/203, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11004/207, dst=netconf:127.0.0.1:11002/207, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11005/201, dst=netconf:127.0.0.1:11003/201, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
eroshkin@root > app activate org.onosproject.alarmcorrelation
Activated org.onosproject.alarmcorrelation
eroshkin@root > links
src=netconf:127.0.0.1:11001/201, dst=netconf:127.0.0.1:11002/201, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11002/201, dst=netconf:127.0.0.1:11001/201, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11002/202, dst=netconf:127.0.0.1:11003/202, type=OPTICAL, state=INACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11002/207, dst=netconf:127.0.0.1:11004/207, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11003/201, dst=netconf:127.0.0.1:11005/201, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11003/202, dst=netconf:127.0.0.1:11002/202, type=OPTICAL, state=INACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11003/203, dst=netconf:127.0.0.1:11004/203, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11004/203, dst=netconf:127.0.0.1:11003/203, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11004/207, dst=netconf:127.0.0.1:11002/207, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
src=netconf:127.0.0.1:11005/201, dst=netconf:127.0.0.1:11003/201, type=OPTICAL, state=ACTIVE, durable=true, metric=1.0, expected=true
```

Figure 5.8: Link states after Alarm Correlation application was executed

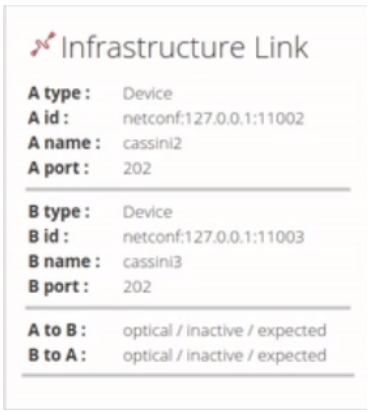

Figure 5.9: Link states after Alarm Correlation application was executed. ONOS GUI.

Chapter 6

Conclusion

6.1 Work evaluation

In this work was studied the problematic of Software-Defined Networking (SDN). I dived inside the principles of SDN and had a look on some protocols used in SDN, mainly on NETCONF. It helped to understand the communication flow between the SDN controller and the device.

Brief overview and comparison of two most popular open-source solutions for SDN controller was provided as well. Each platform has it's own pros and cons. Somehow, industry is currently taking advantage of advanced functionality of OpenDaylight. Probably, after some years of deployment, industry would go back to ONOS again. Upcoming changes in it's architecture are very promising.

In practical part I developed a driver for Nokia's ROADM and extended functionality of existed solution for Nokia's Optical Transponder. Extensions for Optical Transponder driver were contributed to the ONOS community. ROADM driver stayed proprietary for future evolution and future usage.

Architecture of the Alarm Correlation function as a SDN application was proposed. Several tests simulating *"fiber cut"* scenario were performed. Based on the obtained information I was able to track the failure state in the network with regard to the obtained alarms. Based on this information, I came out with a proposal for such an application.

Practical part introduced several enhancements in ONOS controller. Some of these enhancements were contributed to the ONOS community. You can find full list of contributions here [92].

6.1.1 Encountered issues

During the development process several issues were encountered. Brief description is provided within this sub-section.

6.1.1.1 NetconfSessionMinaImpl issue

In the beginning of the ROADM driver development an issue in communication between ONOS and ROADM occurred. SDN controller and device were not able to understand each other. Root cause of this problem was in exchanged NETCONF message format.

There are currently two versions of NETCONF protocol (v1.0 and v1.1), which have some specific changes. One of them is message format. In v1.1 it is different from v1.0. In the beginning of the NETCONF communication Client and Server should exchange *<hello>* messages, which carry set of capabilities that each party implements. Version of NETCONF protocol is also specified within this initial message.

Both parties, Nokia's ROADM and ONOS, are able to communicate with both versions of NETCONF protocol. For some reasons, ONOS always communicated in NETCONF v1.1, when the device expected[1] communication in NETCONF v1.0. This was an obstacle. It made the device misunderstood the communication flow.

In order to fix this, some lines in *NetconfSessionMinaImpl.java*, implementing possibility to communicate in NETCONF v1.1 were commented. It forced ONOS to communicate with NETCONF protocol v1.0. After that communication between the SDN controller and Nokia's ROADM (and Optical Transponder) was always successful.

This is a temporary solution, not something what could be contributed to the ONOS community.

6.1.1.2 RoadmDeviceMessageHandlerView issue

With the development of the ROADM driver, there was a need to display the device under *"Optical UI"* tab, introduced by ONOS for managing optical devices. On the first try Nokia's ROADM didn't show up. Later on minor fixes were done in order to make the ROADM visible inside the *"Optical UI"* tab. These minor changes were contributed as a patch. You can find it here [93].

6.1.1.3 Issue with AlarmConfig behavior

When I started to work with alarms, there was a need to implement AlarmConfig behavioral model, which translates alarms coming from the device and stores them inside the ONOS. There was an issue with incorrect passing of the NETCONF notification through some internal ONOS data structures. With the help of Andrea Campanella, small patch fixing this issue was created, tested and contributed to the community. You can find this contribution here [94].

6.1.1.4 Issue with storing of existing Flow Rules

By default ONOS doesn't store existing configuration installed on the device. Each time device is connected to the ONOS, it tries to clean up the device's configuration. It means that each time you should set a specific flag in ONOS in order to prevent erasing of the device configuration.

In order to ease it and extend this API, thanks to Andrea Campanella, some small fixes were proposed and contributed as a patch. You can find it here [84]. Main contribution is in embedding second flag to the *FlowRuleManager*. Now, you have two flags you need to set. First one, *allowExtraneousRules*, tells ONOS to **not** delete device configuration. Second flag, *importExtraneousRules*, tells ONOS to import this configuration inside the ONOS. By default both flags are set on false, what means that device configuration wouldn't be kept, once the device is connected to ONOS.

[1] Most probably such a choice of NETCONF protocol version was due to the old version of the software running on the device.

6.1.2 Contributions

Except aforesaid patches some other contributions to the ONOS community were done during this work:

- ***PowerConfig*** *behavior* for Nokia's Optical Transponder was contributed [86] to the ONOS community. It allows to set transmitting power values on Line ports of the device.
- ***AlarmConfig*** *behavior* for Nokia's Optical Transponder was contributed [87] as well. It translates OPENCONFIG-based alarms to internal ONOS representation and stores them inside the controller. Alarms could be later visualised through the WebGUI or command line. This implementation is compatible with Nokia's ROADM driver.
- ***Cli command*** for creating *cross-connections* [85] on ROADMs. This command due to the tight bundling with *RoadmManager* instance and *Flow Rule Subsystem* is capable of configuring ROADM of **any** vendor.

6.1.3 Future Work

This work already provides solid basis for applying SDN concepts in Open Optical Transport networks. Somehow, several aspects could be improved:

- ***ROADM driver*** could be extended on further functionality. For example, embed power adjustment capability or enable possibility to set output power on amplifier card.
- ***Alarm Correlation app*** could be potentially reworked. There could be implemented next features:
  - Keep track of wider range of parameters in order to detect *"fiber cut"* scenario. History of output power values as well as laser state (active/inactive) could be taken into account for example.
  - Extend application on further failure scenarios. This could potentially lead to existing algorithm rework.
  - More investigation should be done on interaction with PCE function of ONOS.

6.2 Future networks

Software-Defined Networking concept was introduced already a long time ago. Since that time it is still not fully deployed in the real networks. Rising demand of operators on full automation of the network and desire to do not be attached to only one vendor pushed the operators to create various initiatives. Some of them, like OPENCONFIG and OpenROADM, standardize device functions. Vendor later must implement these data models inside their devices. Configuration of network in multi-vendor environment becomes more universal. It leads to appearance of multi-vendor environment in real networks.

SDN by separating control and data plane introduces programmable approach in the networks. It lets the operator to save costs on network maintenance and network administration, centralizes control of the network and eases network provisioning and troubleshooting. From this prospective SDN is a next logical step of network evolution.

Big companies, like AT&T, Orange, Nippon Telegraph and Telephone (NTT) and Telefonica, are betting on SDN as a future solution. Moreover, development of some promising technologies, like ONAP or OpenDaylight, is lead by these big players, who invested a lot of money and time into these initiatives.

Wrapping aforesaid, future network evolution could be summarized with following pillars:

- ***Deeper integrity of SDN*** into the networks.
- Network will move towards ***Cloud-Native environment***.
  - Server dominating environment would provide more freedom for introduction of NFV/VNF.
  - Microservice-oriented environment would be prevalent.
  - Network orchestration would be done with ONAP, NFV-MANO, Kubernetes or any other network orchestrator.
- Rise of the ***Machine Learning (ML) techniques*** in networking is already triggered. Some big standardization institutions, like ITU, already propose some recommendations. A good example could be ITU-T Y.3172 [95] recommendation, which standardizes ML framework for networking purposes.
- Maybe, one day ***Artificial Intelligence (AI)*** embedded inside the network would be introduced. Somehow, working ML techniques should be introduced in the network first. Only after this industry could start move towards AI.
- ***Multi-vendor*** environment inside the network.

6.2.1 Possible architecture

Until now we were talking about SDN in a small scale of the lab. When it goes to the reality, network consists from thousands of elements. One huge instance of the SDN controller running on top of a global network is quite heavy. Right now SDN is purely used for network configuration, but we also need to create services on top of the network. For this purpose network orchestrator, like ONAP, could be beneficial. On the Figure 6.1 is represented possible architecture of future multi-domain network environment.

As you may notice, SDN controller now covers and manages only certain domain. Domain could be whatever - data center, city (or city agglomeration) or even whole region. In multi-domain case SDN architecture deployed in the network becomes distributed. For this purpose, we need a network orchestrator, which would allocate enough resources for each SDN controller and take care of services running on top of the network.

Another huge milestone for deploying SDN architecture in real networks would be coexistence with legacy networks. Telecom operators already invested a lot of money into their networks. Changing all architecture from the scratch is risky and costly. In this case, there is a need in a transition solution. On Figure 6.2 is represented such architecture, where SDN cooperates with legacy networks.

Network Monitoring System (NMS) is already capable of managing legacy network. It should be enhanced on some additional features in order to cooperate with a SDN controllers or directly with a network orchestrator. One of the possible solutions could be embedding universal interface, like T-API or any other, which would provide state information about the network on higher control levels. Network orchestrator again facilitates work of whole infrastructure. It is one of the possible solutions. A lot of research in this area still needed to be done.

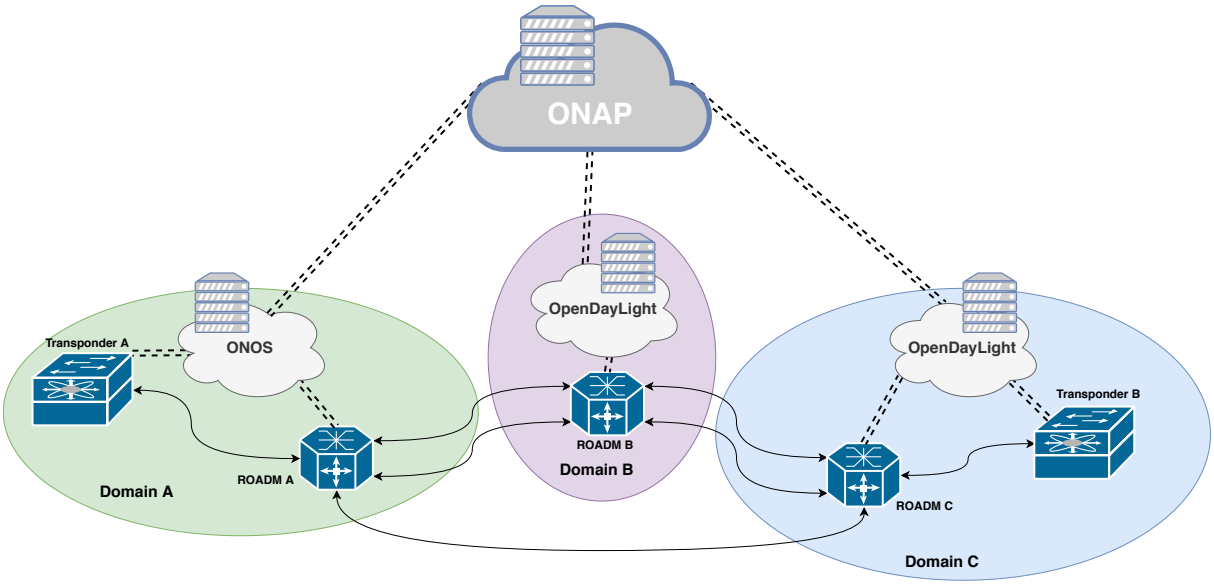

Figure 6.1: Future network architecture

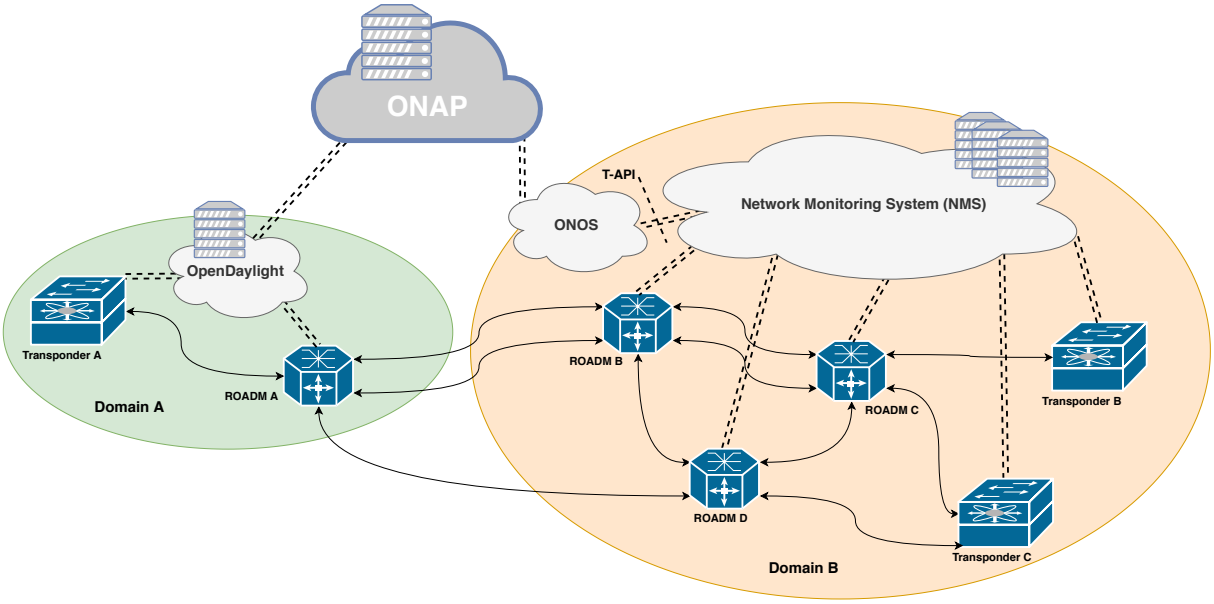

Figure 6.2: Future network **transit** architecture

For the industry SDN seems to be a promising solution as a control platform for the network. It could ease maintenance, troubleshooting and save huge costs on network administration. Standardization activities, like development of data models, are currently running. It should give to the operators more freedom in choice of the device between different vendors, bringing a possibility of multi-vendor environment inside the network.

Raise of virtualization technologies and complementary to the SDN concepts, like NFV/VNF, open wider possibilities for service providing. It also brings opportunity to embed various Machine Learning techniques on top of the network. ML would not provide a greater speed advantage, but it would be extremely beneficial for more precise estimation of network parameters and, mainly, for prediction of events inside the network. It should result in providing a better quality services to the end consumer.

Appendix A

OpenConfig data models for optical communication

Let's have a look on some of the OpenConfig data models and try to understand, what they're providing to us. In terms of this work our main focus is data models for optical transport and some other models which optical devices should support as well.

Platform data module

- *openconfig-platform* – defines a data model for representing a system component inventory, which can include hardware or software information. Each element in the inventory is termed "component" and contains at least a unique name and unique type. Each component has also list of sub-components, like an interface has ports. This is the generic schema for the device. [96]
- *openconfig-platform-types* – defines data types which are supported by the OpenConfig component inventory model. Includes information about hardware, software, chassis, linecard, port, controller card and many more components [97].
- *openconfig-platform-linecard* – this module defines data related to the Linecard component in the *openconfig-platform* model. It describes configuration data and operational state for all linecard components. [98]
- *openconfig-platform-port* – defines data related to the Port components in the *openconfig-platform* model [99]. It includes information about port state, channel speed configuration and similar.
- *openconfig-platform-transceiver* – defines configuration and operational state data for transceivers. Transceiver is expected to be a sub-component of a Port component [100]. Provides operability with input/output parameters of the optical channel like power and frequency.
- *openconfig-platform-ext* – defines optional extensions (i.e. operational state) for the existing OpenConfig data model [101].

Optical transport module

- *openconfig-terminal-device* – describes an optical terminal device data model for managing terminal systems on client and line side in a Dense Wavelength Division Multiplex (DWDM) transport network [102]. This model includes definitions of:
  - *physical port* – physical pluggable client port. Has one or more physical channels;
  - *physical channel* – physical lane or channel in the physical client port;

  - *logical-channel* – logical grouping of logical grooming elements that may be assigned to multiplexing/de-multiplexing, or to an optical channel for the line side transmission. It can represent an ODU/OTU logical packing of the client data onto the line side;
  - *optical channel* – corresponds to an optical carrier and assigns a frequency (wavelength). Can also have power, Bit Error Rate (BER) and operational mode properties. [102]

- *openconfig-transport-types* – contains general type definitions and identities for the optical transport models [103], like frequency value, administrative state mode, base mapping protocols and even some statistics.

- *openconfig-transport-line-common* – describes common data elements for the optical transport line system elements, such as amplifiers and ROADMs (will be described in following chapter) [104]. Includes types of the optical port (if it's ingress or egress), input/output power parameters.

- *openconfig-transport-line-connectivity* – defines the device-level connectivity in terms of internal port-to-port connection for the optical transport system elements like ROADMs and amplifiers [105]. All connections are unidirectional and are only internal.

- *openconfig-optical-amplifier* – contains configuration and operational state data for the optical amplifiers as a part of a transport line system [106]. Defines different types of the optical amplifiers (EDFA, Raman and etc.) with bounds of gain range, amplifier mode, power, fiber type profile and many more parameters.

- *openconfig-wavelength-router* – defines configuration and operational state data for the optical transport line system node, different types of Reconfigurable Optical Add-Drop Multiplexer (ROADM), Wavelength Selective Switch (WSS) [107]. Nodes are modeled as configurable switching elements with ingress and egress ports, add/drop ports which could be splitted between different degrees (group of interface cards ensuring fully functionality) of the device. Also, enables to configure Power Spectrum Density.

- *openconfig-channel-monitor* – describes operational state data for the optical transport system elements such as wavelength router (i.e. ROADM) and amplifier. Specified set of data (operational state, PSD, frequency) which could be monitored [108].

Appendix B

Brief introduction to the ITU-T 694.1

This standard specifies DWDM frequency grid by introducing frequency slots and specifying them. Each frequency slot can be represented by "*Central Frequency*" and "*Slot Width*". Also, to simplify numbering of frequency slots, two parameters, "m" and "n", were introduced. You can compute central frequency according to the following equation [78]:

$$f_{central} = 193.1 + n \times 0.00625, \tag{B.1}$$

where n is a positive or negative integer including 0, and 0.00625 is the nominal central frequency granularity in [THz].

Slot width could be computed like [78]:

$$SlotWidth = 12.5 \times m, \tag{B.2}$$

where m is a positive integer, and 12.5 is the slot width granularity in GHz.

For better understanding let's have a look on Figure B.1, which represent usage of such conception.

Any combination of frequency slots is allowed as long as there is no overlap between any two frequency slots [78].

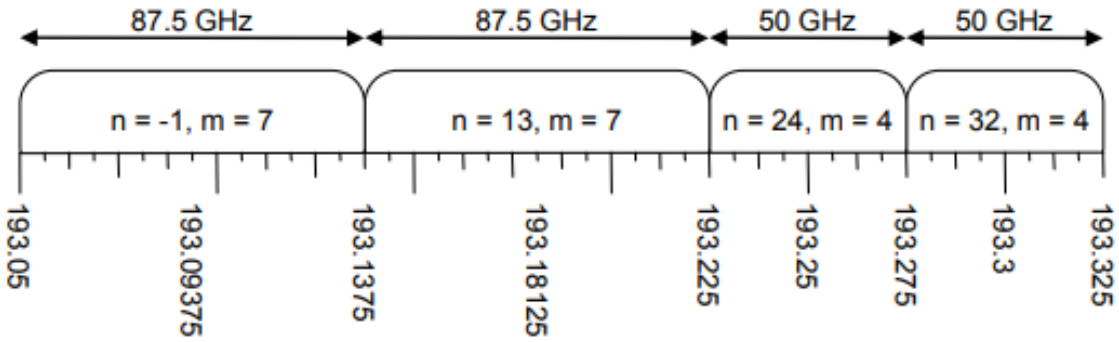

Figure B.1: Flexible DWDM grid

Appendix C

Here are the screenshots of the WebGUI of ONOS, particularly *"Optical UI"* tab.

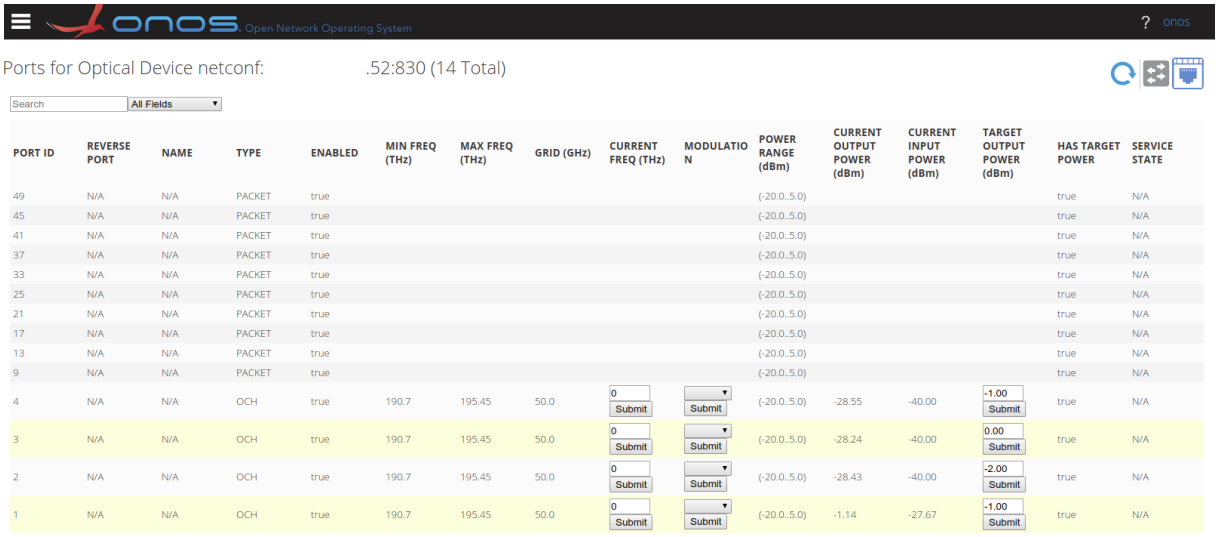

Figure C.1: Optical UI interface. Setting the power on optical port

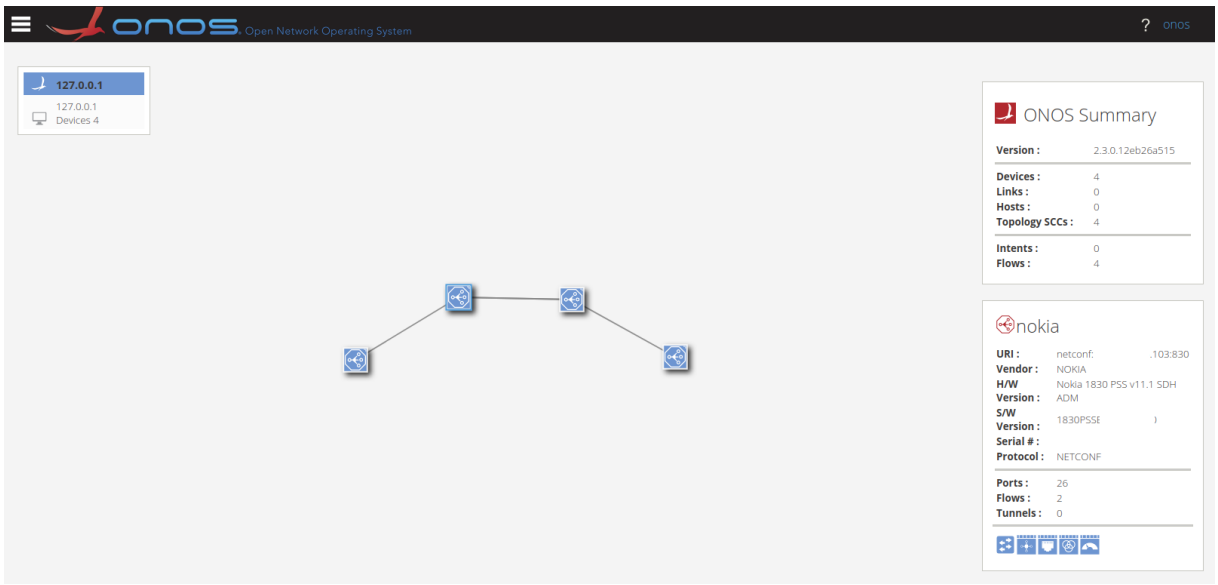

Figure C.2: Topology representation in ONOS

Bibliography

[1] Software-defined networking, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Software-defined_networking.

[2] M. Lessing and S. Staff, What is Software-defined networking (SDN)? Definition. SDxCentral, Sep. 2019. [Online]. Available: https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/.

[3] M. Rouse, L. Rosencance, J. English, and J. Burke, Software-defined Networking (SDN), TechTarget.com, Aug. 2019. [Online]. Available: https://searchnetworking.techtarget.com/definition/software-defined-networking-SDN.

[4] SDxCentral Staff, What is SDN orchestration (SDN Policy orchestration)?, SDxCentral, Aug. 2015. [Online]. Available: https://www.sdxcentral.com/networking/sdn/definitions/what-is-sdn-orchestration/.

[5] M. Rouse and J. DerGurahian, WhatIs. [Online]. Available: https://searchnetworking.techtarget.com/definition/network-orchestration.

[6] Home Page, ONAP. [Online]. Available: https://www.onap.org/.

[7] ONAP. [Online]. Available: https://www.onap.org/architecture.

[8] H. Adams, Open optical networks: what, why and next steps, IHS Inc., Aug. 2017. [Online]. Available: https://technology.ihs.com/594552/open-optical-networks-what-why-and-next-steps.

[9] Open and Disaggregated Transport Network, Open Networking Foundation. [Online]. Available: https://www.opennetworking.org/odtn/.

[10] S. Staff, What is White Box Switching White Box Switches (are they SDN Switches)?, SDxCentral, Jan. 2015. [Online]. Available: https://www.sdxcentral.com/data-center/bare-metal/white-box/definitions/what-is-white-box-networking/.

[11] Network function virtualizetion, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Network_function_virtualization.

[12] SDxCentral Staff, An Overview of NFV Elements, SDxCentral, Jul. 2015. [Online]. Available: https://www.sdxcentral.com/networking/nfv/definitions/nfv-elements-overview/.

[13] OASIS TOSCA, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/OASIS_TOSCA.

[14] S. Raynovich, What is Network Service Chaining? Definition, SDxCentral, Feb. 2016. [Online]. Available: https://www.sdxcentral.com/networking/virtualization/definitions/what-is-network-service-chaining/.

[15] OpenFlow, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/OpenFlow.

[16] SDxCentral Staff, What is OpenFlow? Definition and How it Relates to SDN, SDxCentral, Aug. 2013. [Online]. Available: https://www.sdxcentral.com/networking/sdn/definitions/what-is-openflow/.

[17] P4 project overview, Open Networking Foundation. [Online]. Available: https://www.opennetworking.org/p4/?utm_referrer=https://www.google.fr/.

[18] P4 (programming language), Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/P4_(programming_language).

[19] B. Butler, What P4 programming is and why it's such a big deal for Software Defined Networking, Network World from IDG, Jan. 2017. [Online]. Available: https://www.networkworld.com/article/3163496/what-p4-programming-is-and-why-it-s-such-a-big-deal-for-software-defined-networking.html.

[20] B. Pfaff and B. Davie, The Open vSwitch Database Management Protocol, RFC 7047, Internet Engineering Task Force, Dec. 2013. [Online]. Available: https://tools.ietf.org/html/rfc7047.

[21] SDxCentral Staff, What is Open vSwitch Database or OVSDB?, SDxCentral, Sep. 2014. [Online]. Available: https://www.sdxcentral.com/open-source/definitions/what-is-ovsdb/.

[22] Understanding the OVSDB Protocol Running on Juniper Network Devices, Juniper Networks, Jun. 2018. [Online]. Available: https://www.juniper.net/documentation/en_US/junos/topics/concept/sdn-ovsdb-junos.html.

[23] M. Bjorklund, YANG - A Data Modeling Language for the Network Configuration Protocol (NETC RFC 6020 (Proposed Standard), Internet Engineering Task Force, Oct. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc6020.txt.

[24] The YANG Data Modeling Language, Tail-f Systems. [Online]. Available: http://66.218.245.39/doc/html/ch03.html.

[25] M. Bjoerklund, DHCP Tutorial, YANG Central. [Online]. Available: http://www.yang-central.org/twiki/bin/view/Main/DhcpTutorial.

[26] OpenConfig Overview, Juniper Networks. [Online]. Available: https://www.juniper.net/documentation/en_US/junos/topics/concept/openconfig-overview.html.

[27] M. Rouse and M. Haughn, OpenConfig, WhatIs.com. [Online]. Available: https://whatis.techtarget.com/definition/OpenConfig.

[28] J. Edelman, OpenConfig, Data Models, and APIs. [Online]. Available: http://jedelman.com/home/openconfig-data-models-and-apis/.

[29] OpenConfig - Home, OpenConfig Working Group. [Online]. Available: http://www.openconfig.net/.

[30] Openconfig data models and apis, OpenConfig Working Group. [Online]. Available: http://www.openconfig.net/projects/models/.

[31] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, Network Configuration Protocol (NETC RFC 6241 (Proposed Standard), Internet Engineering Task Force, Jun. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6241.txt.

[32] The NETCONF Server, Tail-f Systems. [Online]. Available: http://66.218.245.39/doc/html/ch15.html.

[33] Network Configuration Protocol, NetConf Central. [Online]. Available: http://www.netconfcentral.org/netconf_docs.

[34] S. Chisholm and H. Trevino, NETCONF Event Notifications, RFC 5277 (Proposed Standard), Internet Engineering Task Force, Jul. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5277.txt.

[35] M. Foschiano, Cisco Systems UniDirectional Link Detection (UDLD) Protocol, RFC 5171 (Informational), Internet Engineering Task Force, Apr. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5171.txt.

[36] XPATH Cover Page, W3C. [Online]. Available: http://www.w3.org/TR/xpath.

[37] A. Bierman, M. Bjorklund, and K. Watsen, RESTCONF Protocol, RFC 8040, Internet Engineering Task Force, Jan. 2017. [Online]. Available: https://tools.ietf.org/html/rfc8040.

[38] A. Dainese, NETCONF and RESTCONF, ipSpace.net. [Online]. Available: https://www.ipspace.net/kb/CiscoAutomation/070-netconf.html.

[39] Kanika, What Is RESTCONF?, SDN Tutorials. [Online]. Available: http://sdntutorials.com/what-is-restconf/.

[40] K. Lelonek, A brief introduction to gRPC in Go. [Online]. Available: https://blog.lelonek.me/a-brief-introduction-to-grpc-in-go-e66e596fe244.

[41] J. Smith, Introduction to gRPC, Container Solutions, Mar. 2017. [Online]. Available: https://blog.container-solutions.com/introduction-to-grpc.

[42] gRPC Concepts Overview, chromium.googlesource.com. [Online]. Available: https://chromium.googlesource.com/external/github.com/grpc/grpc/+/HEAD/CONCEPTS.md.

[43] S. Rao, SDN Series Part Seven: ONOS, thenewstack.io, 2015. [Online]. Available: https://thenewstack.io/open-source-sdn-controllers-part-vii-onos/.

[44] A. Koshibe, Intent Framework, wiki.onosproject.org, May 2016. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Intent+Framework.

[45] Architecture and Internals Guide, ONOS wiki. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Architecture+and+Internals+Guide.

[46] TAPI Overview, Open Networking Foundation. [Online]. Available: https://wiki.opennetworking.org/pages/viewpage.action?pageId=317292566.

[47] OSGi, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/OSGi.

[48] S. Rao, SDN Series Part Six: OpenDaylight, The Most Documented Controller, thenewstack.io, Jan. 2015. [Online]. Available: https://thenewstack.io/sdn-series-part-vi-opendaylight/.

[49] OpenDaylight Architecture - Carbon release, The Linux Foundation. [Online]. Available: https://www.opendaylight.org/what-we-do/current-release/carbon.

[50] Cisco Open SDN Controller, Cisco Systems Inc. [Online]. Available: https://www.cisco.com/c/en/us/products/cloud-systems-management/open-sdn-controller/index.html.

[51] A. Vahdat, Enter the Andromeda zone - Google Cloud Platform's latest networking stack, Google, Apr. 2014. [Online]. Available: https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html.

[52] L. Hardesty, Google Brings SDN to the Public Internet, SDxCentral, Apr. 2017. [Online]. Available: https://www.sdxcentral.com/articles/news/google-brings-sdn-public-internet/2017/04/.

[53] ONOS, ONF. [Online]. Available: https://docs.onosproject.org/.

[54] ONOS configuration overview, ONF. [Online]. Available: https://docs.onosproject.org/onos-config/docs/.

[55] [Online]. Available: https://www.opennetworking.org/stratum/.

[56] Scott M. Fulton III, OpenDaylight and ONOS: Does SDN Really Need Two Controllers?, thenewstack.io, Mar. 2016. [Online]. Available: https://thenewstack.io/opendaylight-onos-sdn-really-need-two-controllers.

[57] S. Rao, SDN Series Part Eight: Comparison Of Open Source SDN Controllers, thenewstack.io, Mar. 2015. [Online]. Available: https://thenewstack.io/sdn-series-part-eight-comparison-of-open-source-sdn-controllers/.

[58] C. Lam, Passive Optical Networks: principles and practice, 2007. [Online]. Available: https://books.google.fr/books?hl=en&lr=&id=DSO5CVBuhKEC&oi=fnd&pg=PP1&dq=passive+optical+network+overview&ots=rQnTEOYxFa&sig=-TSbsXwLpO8UqTVT5C_x1bkp8ko#v=onepage&q=passive%20optical%20network%20overview&f=false.

[59] Passive Optical Networks, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Passive_optical_network.

[60] Passive Optical Networks, ScienceDirect. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/passive-optical-network.

[61] FTTH Access Networks - AON vs. PON, fiberopticshare.com, 2015. [Online]. Available: http://www.fiberopticshare.com/ftth-access-networks-aon-vs-pon.html.

[62] Basic Knowledge About GPON SFP Trancievers, fs.com, Jan. 2018. [Online]. Available: https://community.fs.com/blog/basic-knowledge-about-gpon-sfp-transceivers.html.

[63] Wavelength-division multiplexing, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Wavelength-division_multiplexing.

[64] [Online]. Available: https://www.springer.com/gp/book/9783030162498#aboutAuthors.

[65] Tutorials of Fiber Optic Products, Difference between Pre-Amplifier, Booster Ampllifier and In-line fiber-optic-tutorial.com. [Online]. Available: http://www.fiber-optic-tutorial.com/differences-between-pre-amplifier-booster-amplifier-line-amplifier.html.

[66] FiberLabs Inc., Erbium-Dopped Fiber Amplifier (EDFA), FiberLabs Inc. [Online]. Available: https://www.fiberlabs.com/glossary/erbium-doped-fiber-amplifier/.

[67] Dr. Rüdiger Paschotta, Fiber Amplifiers, RP Photonics Encyclopedia. [Online]. Available: https://www.rp-photonics.com/fiber_amplifiers.html.

[68] D. R. Paschotta, Raman Amplifiers, RP Photonics Encyclopedia. [Online]. Available: https://www.rp-photonics.com/raman_amplifiers.html.

[69] Semiconductor Optical Amplifier (SOA) Introduction, Fiber Optic Solutions. [Online]. Available: http://www.fiber-optic-solutions.com/introduction-semiconductor-optical-amplifier-soa.html.

[70] Optical Switch, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Optical_switch.

[71] Optical Add-Drop Multiplexer, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Optical_add-drop_multiplexer.

[72] Reconfigurable Optical Add-Drop Multiplexer, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Reconfigurable_optical_add-drop_multiplexer.

[73] A Primer on ROADM Architecture, ADVA. [Online]. Available: https://oristel.com.sg/wp-content/uploads/2015/03/A-Primer-on-ROADM-Architectures.pdf.

[74] Optical Networks - ROADM, Tutorials Point. [Online]. Available: https://www.tutorialspoint.com/optical_networks/optical_networks_roadm.htm.

[75] Wavelength Selective Switch, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Wavelength_selective_switching.

[76] Principles of 2-D MEMS switching, ResearchGate. [Online]. Available: https://www.researchgate.net/figure/Principle-of-the-2D-MEMS-switching_fig2_309168203.

[77] Next-Generation ROADM Architecture Benefits, Fujitsu. [Online]. Available: https://www.fujitsu.com/us/Images/Fujitsu-NG-ROADM.pdf.

[78] Spectral grids for WDM applications: DWDM frequency grid, Recommendation ITU-T G.694.1, International Telecommunication Union, Feb. 2012. [Online]. Available: https://www.itu.int/rec/T-REC-G.694.1-201202-I/en.

[79] 1830 Photonic Service Interconnect (PSI), Nokia. [Online]. Available: https://www.nokia.com/networks/products/1830-photonic-service-interconnect/#overview.

[80] 1830 Photonic Service Switch, Nokia. [Online]. Available: https://www.nokia.com/networks/products/1830-photonic-service-switch/#overview.

[81] S. Desai, ONOS Driver Consolidation, PALC Networks. [Online]. Available: https://gerrit.onosproject.org/#/q/owner:dsudeep%2540palcnetworks.com+status:open.

[82] odtn-driver.xml, ONOS project. [Online]. Available: https://github.com/opennetworkingla
onos/blob/master/drivers/odtn-driver/src/main/resources/odtn-drivers.xml.

[83] ONOS project, Flow Rule Subsystem, ONF. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Flow+Rule+Subsystem.

[84] I. Eroshkin, Save Flow Rules patch, Nov. 2019. [Online]. Available: https://gerrit.onosproject.org/#/c/22816/.

[85] ——, Cli command to manage XCs on ROADM, Nokia Bell Labs, Oct. 2019. [Online]. Available: https://gerrit.onosproject.org/#/c/22750/.

[86] ——, (Contribution) PowerConfig capability for Nokia's Optical Transponder, Nokia Bell Labs, Oct. 2019. [Online]. Available: https://gerrit.onosproject.org/#/c/22722/.

[87] ——, NokiaAlarmConfig interface, Nokia Bell Labs, Dec. 2019. [Online]. Available: https://gerrit.onosproject.org/#/c/22870/.

[88] I. Eroshkin and D. Verchere, Alarm Correlation application, Nokia Bell Labs, Nov. 2019. [Online]. Available: https://github.com/eroshiva/Alarm-Correlation-App-Sample.

[89] Openconfig-alarms.yang. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/system/openconfig-alarms.yang.

[90] Openconfig-alarm-types.yang. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/system/openconfig-alarm-types.yang.

[91] H. Okui, ODTN Phase 1.0 Demo (at NTTCom Lab), ONOS project, May 2019. [Online]. Available: https://wiki.onosproject.org/pages/viewpage.action?pageId=23335851.

[92] I. Eroshkin, Contributions, Nokia Bell Labs, 2019. [Online]. Available: https://gerrit.onosproject.org/#/q/owner:Ivan+Eroshkin.

[93] ——, Optical UI patch, Nokia Bell Labs, Aug. 2019. [Online]. Available: https://gerrit.onosproject.org/#/c/22501/.

[94] ——, AlarmConfig patch, Nokia Bell Labs, Nov. 2019. [Online]. Available: https://gerrit.onosproject.org/#/c/22806/.

[95] Architectural framework for machine learning in future networks including IMT-2020, International Telecommunication Union, Jun. 2019. [Online]. Available: https://www.itu.int/rec/T-REC-Y.3172/en.

[96] openconfig-platform.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/platform/openconfig-platform.yang.

[97] openconfig-platform-types.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/platform/openconfig-platform-types.yang.

[98] openconfig-platform-linecard.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/platform/openconfig-platform-linecard.yang.

[99] openconfig-platform-port.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/platform/openconfig-platform-port.yang.

[100] openconfig-platform-transceiver.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/platform/openconfig-platform-transceiver.yang.

[101] openconfig-platform-ext.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/platform/openconfig-platform-ext.yang.

[102] openconfig-terminal-device.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/optical-transport/openconfig-terminal-device.yang.

[103] openconfig-transport-types.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/optical-transport/openconfig-transport-types.yang.

[104] openconfig-transport-line-common.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/optical-transport/openconfig-transport-line-common.yang.

[105] openconfig-transport-line-connectivity.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/optical-transport/openconfig-transport-line-connectivity.yang.

[106] openconfig-optical-amplifier.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/optical-transport/openconfig-optical-amplifier.yang.

[107] openconfig-wavelength-router.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/optical-transport/openconfig-wavelength-router.yang.

[108] openconfig-channel-monitor.yang, OpenConfig Working Group. [Online]. Available: https://github.com/openconfig/public/blob/master/release/models/optical-transport/openconfig-channel-monitor.yang.