TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JANNE MIKOLA

UTILIZING NORTHBOUND API OF THE SDN STACK IN WEB-BASED NETWORK MANAGEMENT

Master of Science Thesis

Examiner: Adjunct Professor Ossi Nykänen
Examiner and topic approved by the Council of the Faculty of Computing and Electrical Engineering on 5th of May 2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
MIKOLA, JANNE: Utilizing northbound API of the SDN stack in web-based network management
Master of Science Thesis, 96 pages, 0 Appendix pages
September 2014
Major: Hypermedia
Examiner: Adjunct Professor Ossi Nykänen
Keywords: Software Defined Networking, application development, application programming interfaces, REST, networking

The thesis is divided into two parts. In the literature study part, state of Software-defined Networking (SDN) standardization is walked through and compared to the current implementations of on-market vendors. In this part the technical possibilities enabled by SDN are mapped and SDN's probability of becoming the new way of building networks is assessed lightly. In the case study part, prototypes of such value adding services are designed and created. Showcase starts by comparing available SDN solutions and selecting two of them for closer inspection. Case study continues by making technological choices in API and development tool realms. Then, the process of creating a prototype of a management interface on one SDN controller is described. The designed and implemented solution has the capability of dynamically prioritizing the network flows and dynamically changing the route of a network flow from the shortest path (from source to destination) into forcing it to make a detour through intrusion prevention system before being allowed to reach the destination. As a part of the research, a web tool including user interface for achieving described functions is created, because such tools are widely adopted by ISPs as the visible interface for customers to interact with.

After the solution has been finalized on one SDN controller platform, the research shifts to analyze the consequences of changing one SDN controller to another. In this part, standardization situation of both south- and northbound interfaces are discussed more closely. Research finds out that because of the lack of standardization in northbound REST APIs, change invalidates most, or all, of the developed SDN applications. Thesis articulates if change with full functionality retained is possible, and analyses the amount of needed work for straightforward code conversion or other means.

The study indicates that SDN works as enabling technology and makes it possible to achieve functions in network management that have earlier been impossible through programming, mostly because the lack of relevant application programming interfaces. The thesis validates SDN as a functional technology with huge headroom for service development possibilities, however it finds the standardization of the northbound programming interfaces lacking and brings up questions about market penetration chances for the technology due to some major vendors having less than enthusiastic implementations of the technology.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
MIKOLA, JANNE: SDN-pinon ulkoisten rajapintojen hyödyntäminen web-pohjaisessa verkonhallinnassa
Diplomityö, 96 sivua, 0 liitesivua
Syyskuu 2014
Pääaine: Hypermedia
Tarkastaja: dosentti Ossi Nykänen
Avainsanat: Software-defined networking, ohjelmistokehitys, ohjelmistorajapinnat, REST, verkottaminen

Tämä diplomityö on jaettu kahteen osaan. Taustatutkimusosassa tarkastellaan Software-defined Networking (SDN) –konseptin standardoinnin tilaa ja vertaillaan markkinoilla olevia ratkaisuja sekä standardiin, että toisiinsa. Tässä osassa SDN:n luomia teknisiä mahdollisuuksia kartoitetaan ja analysoidaan mahdollisuuksia sille, että SDN teknologiana mullistaa tietoverkottamisen lähivuosina.

Tapaustutkimusosassa suunnitellaan, määritellään ja toteutetaan esimerkkiprototyyppi SDN-teknologian mahdollistamista uudenlaisista lisäarvopalveluista. Tapaustutkimus lähtee liikkeelle tekemällä analyyttisen hierarkiaprosessin mukaisen teknologiavalinnan saatavilla olevista SDN-ratkaisuista ja valitsemalla ja suunnittelemalla käytetyt kehitysympäristöt. Valituissa ympäristöissä ja valituilla alustoilla luodaan prototyyppi verkonhallintasovelluksesta pohjautuen SDN-teknologiaan. Luotu prototyyppi pystyy dynaamisesti hallitsemaan kytkinverkon tietoliikenteen reittejä sekä uudelleenmäärittelemään tiettyjen datavirtojen QoS-määrityksiä. Diplomityötä varten luodussa käyttötapauksessa kytkinverkossa havaitun tietovirran oletetaan sisältävän epäilyttävää liikennettä, jonka vuoksi se halutaan siirtää hallitusti lyhimmältä polulta pidemmälle reitille, jonka varrella toimiva IDPS-järjestelmä analysoi tietoliikenteen ja tekee päätöksen siitä, sallitaanko tietoliikenne verkossa. Osana toteutusta hallintatyökalulle luodaan web-käyttöliittymä, josta käyttäjä voi hallita kuvattuja toimintoja.

Kun prototyypin toteutus on viimeistelty yhdellä SDN-alustalla, tutkimus siirtyy analysoimaan SDN-kontrollerin vaihdon seurauksia luodun SDN-prototyypin kannalta. Tässä osuudessa SDN-pinon standardointiin kiinnitetään lisää huomiota. Tutkimus havaitsee, että heikon ulkopuolisten rajapintojen standardointitilanteen vuoksi alustojen tarjoamat REST-rajapinnat eroavat huomattavasti toisistaan. Käytännössä tämä saa aikaan sen, että yhdelle alustalle luotu toiminnallisuus on riippuvainen kyseisestä alustasta. Diplomityö käy läpi mahdollisisia tapoja toteuttaa laitteistotoimittajariippumattomia ulkoisia rajapintoja SDN-viitekehyksessä ja analysoi työn määrän siirrettäessä jo kerran toteutettua toiminnallisuutta toiselle SDN-alustalle.

Tutkimus vahvistaa SDN-teknologian olevan nykyisessä muodossaan käyttökelpoista ja uusia mahdollisuuksia luovaa tekniikkaa. Verkkojen ohjelmoitavuus ei työn mukaan ole aiemmin ollut mahdollista puuttuvien ohjelmointirajapintojen vuoksi. Diplomityön johtopäätelmä on, että SDN tarjoaa lupaavan kehitysympäristön ja alustan uudenlaiselle toiminnallisuudelle. Suurimmiksi SDN:n haasteiksi havaitaan joidenkin laitteistotoimittajien haluttomuus ja heikko standardointitilanne.

PREFACE

This thesis was written as a partial fulfilment of the requirements for a Master of Science's degree in Computer Science from Tampere University of Technology. It contains work done from March to September 2014. Both experimental prototyping work and writing were done as single employee of TeliaSonera Finland Oyj without any ties to ongoing in-house projects. The thesis has been made solely by the author; most of the text is based on earlier research however, and sources for said research are presented in-text and in the References-section of the thesis.

I would like to thank my examiner Ossi Nykänen for useful insight on the subject as well as flexibility with a full-time employed thesis writer like me. Thanks also go to whole TeliaSonera organization, especially my direct superior, Arto Urataipale for giving the chance to use some of my work hours for the thesis work, as well as my colleague Jarkko Suominen at TeliaSonera for introducing me to the Software-defined Networking. Pekka Töytäri kept encouraging me with the thesis project and Pekka Isomäki offered important insight on SDN and NFV realms many times, so I'd like to thank both these TeliaSonera employees as well. Thanks go also to Timo Ekholm at TeliaSonera for taking interest in my thesis work.

In Tampere, Finland, on 30th of September 2014.

Janne Mikola

TABLE OF CONTENTS

Abstract i
Tiivistelmä ii
Preface iii
Terms and definitions vi
1. Introduction 1
2. Programmability of networks 3
   2.1 Software-defined Networking 3
      2.1.1 Birth of OpenFlow 4
      2.1.2 SDN and data centers 5
      2.1.3 The definition of SDN 6
      2.1.4 Claimed SDN benefits 8
      2.1.5 SDN's road to existing networks 10
   2.2 Programmability and APIs in general 13
      2.2.1 REST 13
      2.2.2 XML and JSON 14
      2.2.3 Introduction to OpenFlow 14
      2.2.4 OpenFlow 1.0 15
      2.2.5 OpenFlow additions in versions 1.1, 1.2 and 1.3 15
   2.3 SDN programming interfaces 16
   2.4 Standardization 17
   2.5 Network Functions Virtualization (NFV) 18
3. SDN controllers as new functionality enablers 19
   3.1 Assessment of available SDN controllers 19
      3.1.1 The big three: Juniper, Cisco, Hewlett-Packard 19
      3.1.2 Cisco APIC 20
      3.1.3 Juniper Contrail and OpenContrail controllers 22
      3.1.4 HP VAN SDN Controller 25
      3.1.5 Other significant actors and open source controllers 26
      3.1.6 Project Floodlight 27
      3.1.7 OpenDaylight 29
      3.1.8 Big Switch Networks Big Network Controller 30
      3.1.9 POX and NOX controllers 31
      3.1.10 IBM SDN VE Unified Controller 31
      3.1.11 Other SDN controllers 32
   3.2 Selecting two SDN controllers as development platforms 34
      3.2.1 Applying analytic hierarchy process on the problem 35
      3.2.2 Comparing the SDN controllers on each attribute 39
      3.2.3 Controller descriptions 43
      3.2.4 HP VAN SDN Controller internals 44
      3.2.5 OpenDaylight internals 45

3.2.6 Northbound APIs in SDN controllers ... 46
3.2.7 HP VAN SDN Controller's Northbound API ... 46
3.2.8 OpenDaylight Northbound API ... 50
3.3 Development on HP VAN SDN Controller ... 51
3.3.1 SDK features ... 51
3.3.2 HP VAN SDN Controller Administrator Guide -document ... 51
3.3.3 SDN Controller Programming Guide -document ... 52
3.3.4 HP VAN SDN Controller REST API –document ... 53
3.3.5 Development environment specification ... 54
3.3.6 Initial physical development environment ... 54
3.3.7 Move to virtualized environment ... 55
4. Utilizing SDN functionality via HP VAN SDN Controller Northbound API ... 58
4.1 Description of new network management functionality ... 58
4.2 Technical design ... 59
4.2.1 Reroute -functionality ... 59
4.2.2 Quality-of-Service modification -functionality ... 63
4.3 Testing and validating the created service ... 64
4.4 Analysis of the development process ... 66
4.5 Analysis of the codebase created ... 67
4.6 Moving the solution to another SDN Controller platform ... 67
4.6.1 Direct comparison of HP and OpenDaylight REST API methods ... 68
4.6.2 Authentication and serialization formats ... 69
4.6.3 Flow creation and modification ... 71
4.6.4 Other differences ... 74
4.6.5 More robust solution through standardization or encapsulation ... 74
5. Discussion ... 76
5.1 The controller choice ... 76
5.2 Criticism and risks ... 77
5.3 Analysis of the created application ... 80
5.3.1 Benefits of SDN application –based network management ... 82
5.3.2 Significance of the developed web application ... 83
5.4 Controller interchangeability ... 83
5.5 SDN ecosystems ... 84
5.6 SDN roadmap ... 85
6. Conclusions ... 87
References ... 89

TERMS AND DEFINITIONS

AHP	Analytic Hierarchy Process (AHP) is a multi-criteria decision making method
API	Application Programming Interface (API) is a set of programmatically accessible methods and procedures that allow creation of applications relying in the functionality of another application
ASIC	Application-specific integrated circuit (ASIC), an integrated circuit (IC) customized for a particular use
Backbone	Refers to the Internet backbone, the principal data routes between large interconnected networks and core routers on the Internet
BGP	Border Gateway Protocol (BGP) is a protocol designed to exchange routing and reachability information between routers in a network
CAPEX	Capital expenditure (CAPEX), are expenditures used by a company to acquire or upgrade physical assets, such as equipment or property
CLI	Command Line Interface (CLI), refers to accessing a device or computer via terminal connection
DHCP	Dynamic Host Configuration Protocol (DHCP) is a protocol enabling automatic assigning of IP addresses to computer in a network
DNS	Domain Name Service (DNS) is a system for naming computers and systems in networks, hierarchically organized into domains
Flow	In SDN vocabulary, a flow refers to an identifiable data stream between two nodes in the network
GUI	Graphical User Interface (GUI) is a visual human-computer interface used to manipulate system functions via graphical representations of said functions and visual interaction with them
HTML	Hypertext Markup Language (HTML), a XML-based language for describing web pages
HTTP	Hypertext Transport Protocol (HTTP), the protocol on which WWW is based on
HTTP methods	Methods included in the HTTP protocol for manipulating information, such as GET, PUT, POST and DELETE
IETF	Internet Engineering Task Force (IETF), de facto organization defining Internet-related standards

IP	Internet Protocol (IP) is the method and protocol by which data is sent from one computer to another on the Internet
IP address	Internet Protocol address, a unique string of numbers separated by full stops identifying computers on networks using IP to communicate
JPEG	Joint Photographic Experts Group (JPEG), a lossy image compression standard
JSON	JavaScript Object Notation (JSON), a lightweight data interchange format and an alternative for XML
LAN	Local Area Network (LAN), a network segment encompassing a local area, such as an office building
MAC address	Media Access Control (MAC) address is a unique identifier for devices within physical network segments
MPLS	Multiprotocol Label Switching (MPLS), a scalable and protocol-independent transport method relying on labels instead of packet headers
Namespace	A class of elements in which each element is unique to that class, although they might be conflicting with other elements in other namespaces
NETCONF	A protocol by IETF to install, manipulate and delete the configuration of network devices released in 2006
NFV	Network Functions Virtualization (NFV), a technology aiming to virtualize network devices into software run on standard PC hardware
ONF	Open Networking Foundation (ONF), the organization that upkeeps the OpenFlow standard and is essential in defining SDN's developmental direction
OpenFlow	A protocol that configures network components via an API-like process
OPEX	Operational expenditure (OPEX) are the expenditures that company spends on an ongoing, day-to-day basis in order to run a business or a system
OSGi	Open Service Gateway initiative (OSGi) defines a dynamic module system for Java
QoS	Quality of Service (QoS) is the overall performance of a computer network, which can be affected with technical parameters
REST	Representational state transfer (REST), a stateless architectural style for designing and implementing APIs for Web Services
RESTful	A web service implemented with REST APIs and conforming to REST principles

RPC	Remote Procedure Call (RPC), a historical method of designing and implementing APIs
SDK	Software Development Kit (SDK) is a set of software development tools, documentation, code and examples that allow and help creation of applications for a certain software package
SDN	Software Defined Networking (SDN) provides programmable interfaces within network infrastructure to enable simplicity and automation in provisioning network services
SDN controller	The device that takes the control of an SDN-enabled network's decision making process
SOAP	Simple Object Access Protocol (SOAP), widely supported protocol for implementing APIs for Web Services
TCP	Transmission Control Protocol (TCP), a protocol used along with the IP protocol to send data and track packets through the Internet
TLS	Transport Layer Security (TLS), a security protocol from IETF based on SSL 3.0
UDP	User Datagram Protocol (UDP), a protocol used to send messages, or datagrams, to other hosts in the internet network without tracking the data packets
URI	Uniform Resource Identifier (URI), a mechanism used to identify resources in semantic environments
WAN	Wide Area Network (WAN), a network segment that covers a broad area, encompassing regional, national or even international areas
Web Service	Web service is a way of communicating between two applications over a network using HTTP as the transport protocol
WLAN	Wireless Local Area Network (WLAN) is a type of local-area network (LAN) which uses high-frequency radio waves rather than wires for transporting the data
Virtualization	Virtualization refers to the creation of virtual resource such as a server, operating system, file or storage
XML	Extensible Markup Language (XML), the basis on which most of the markup languages such as HTML are based on
XMPP	Extensible Messaging and Presence Protocol (XMPP) is a generalized, extensible framework for exchanging messages in XML format
YANG	A modeling language for defining the content carried via NETCONF protocol

1. INTRODUCTION

When compared to interleaving and interacting technologies, such as software development and IT hardware as whole, networking as a technology has remained relatively stable for a very long time. Some new protocols have been introduced, most of which impact has remained smaller than initially envisioned (such as IPv6), but the fundamental technology has remained the same for almost 15 years. Advances in ASIC manufacturing have made the networking devices more efficient, but the paradigm has been unshaken after the invention of VLANs is 1990s.

Software-defined networking (SDN) aims to bring programmability of networks first time available for mass deployments in production environments. This would create significant possibilities for network configuration, change management and dynamics of the whole technological field, especially when SDN makes it appearance conveniently with a sibling technology called Network Function Virtualization (NFV). The potential synergies of NFV and SDN could act as the force ensuring the success and adoption of SDN in the networking market.

Basically SDN is an enabling technology, which in itself doesn't do much. The value has to be created later in the value chain, by network operators, software developers and active community by developing functionality on top of the SDN by creating SDN applications. SDN applications are responsible for making changes in the behavior of the network, and bare SDN only enables to possibility for that.

SDN has been taken seriously on all levels of the industry, while at the same time the degree in which it truly will revolutionize networking remains unclear. There are some areas where SDN can already be deployed cost-effectively, such as datacenters, but the big question is if SDN can change networking outside of data centers, in LANs, WANs and carrier backbones. According to Matthew Palmer in SDN Market Sizing presented at SDNCentral (Palmer, 2013), SDN has already generated an industry of its own worth of hundreds of millions of dollars annually and is expected to grow into billions by 2018. The forecast assures that networking companies have noticed the possibility for a shift in networking to a model where network management wouldn't be any more as much about making configurations to network devices but programming dynamic functionality to agile and adaptive networks powered by devices called SDN controllers. SDN does present a new paradigm for networking, but generally it seems that network-

ing vendors are not yet capable of presenting ready-made use cases that would act as the "killer app" for SDN.

The purpose of this thesis is to chart the situation and possibilities of SDN as a technology, compare existing SDN platforms and create a proof-of-concept –level prototype on one of said platforms. After the prototype creation, a question whether it's effectively possible to change SDN platforms after one such platform has been chosen, is answered. The thesis is about technical validation of currently available SDN platforms in the context of an international carrier and telecommunications operator, TeliaSonera.

2. PROGRAMMABILITY OF NETWORKS

2.1 Software-defined Networking

The first precursors of what we today know as Software-defined Networking can be seen as early as around 2003, when IETF (Internet Engineering Task Force) was reviewing more efficient and functional ways of designing networking devices, such as switches and routers. Intel employees H. Khosravi and T. Anderson came to a conclusion in IETF's RFC 3654 that it would be advantageous to separate forwarding element from the control element in the network devices themselves, and they conceived a standard proposing this, called ForCES (Forwarding and Control Element Separation) (Khosravi & Anderson, 2003). The strengths in idea of separating these two elements in the devices wall relatively well-received by networking device vendors, and they started adapting it on hardware and software level as can be seen in the architecture of current generation networking devices, but work only happened within devices themselves. The Kohsravi's and Anderson's proposed ForCES standard went even further, specifying that it would theoretically be possible to even separate the control functionality off of the device, to altogether another device or even server.

The subject was revisited quickly, in 2005 when Stanford University initiated a program called "Clean Slate". The program's goal was to study how networks and networking equipment were invented today - without any legacy burdens – how would they be designed. (Stanford University, 2012) Clean Slate –program attempted to find out if there were any suboptimal ideas or even faults in any current day networking technology. Clean Slate arrived to a conclusion, that on like many other technologies, there should be a distinct controlling element in the networking also, containing the capability of making the decisions for the whole network or at least some segment of it, or even containing the intelligence of the network itself. Clean Slate saw distributed device configurations in extremely negative light and the program concluded that the network should be driven by network-level objectives, lowering the abstraction level of what kind of information in the networking devices' configurations is now-a-days stored. Clean Slate directly suggested a centralized system, which would be able to see the whole network, and make the traffic forwarding and routing decisions for that network.

The work on what we today know as Software-defined Networking kept being spearheaded at Stanford University, where the first SDN-type centralized controller was created by the name Ethane (Standord University, 2006). The development work was aided

by vendors such as NEC and HP, who added features to their existing networking devices, making it possible to subdue these devices' forwarding functionality under control of an external, centralized system. The implementation wasn't very generalized still, Ethane basically implemented a NAC-solution (Network Access Control) which are widely available today as non-SDN-based products. Ethane controlled how individual devices should be able to access network (e.g. network resources, subnets etc.) based on various information (policy information, topology, registration, bindings). The difference with NAC solutions that are common on market today was that it was achieved not by adding a RADIUS-enabled hardware appliance to the network, but by exposing existing networking devices' "flow tables" to a central controller. The primary researcher on Ethane, Martin Casado, later moved to found a company called Nicira, which was later acquired by VMware (Kerner, 2013).

2.1.1 Birth of OpenFlow

The protocol that was used with Ethane controller was matured into OpenFlow-protocol, which is a standardized protocol for centralized controllers to communicate with, and even control, networking devices. According to the information available on HP's OpenFlow Overview, OpenFlow has been implemented in various switches and routers from as early as the end of 2007. (Hewlett-Packard Development Company, L.P., 2014a) However the support had been experimental and only available for certain devices of many vendors' vast lineup of switches and routers. According to the timestamps on official OpenFlow specifications available in the Open Networking Foundation's website, the first official OpenFlow specification was released in December 2008. Since then the OpenFlow standard has evolved further, OpenFlow 1.0 being released in December 2009, and since that point in time, there have been versions of 1.1, 1.2, 1.3 and most recently, 1.4. (Open Networking Foundation, 2014a) OpenFlow 1.0 seems to still remain the most widely adopted protocol, while OpenFlow 1.3 seems to be the next version industry is willing to adopt.

While academia was relatively interested in Software-defined networking, vendors initially (and to some degree, still today) saw it as a threat, as pointed out by Julie Bort in her Business Insider article in October 2013. (Bort, 2013) Networking as a whole, as well as networking devices had remained as closed systems for a long time all the way from their inception, meaning that only the hardware vendors themselves had been able to write software and add features to them. Incumbent networking vendors also felt that the new technology, aimed to re-defining how networking works, was a threat and opened up possibilities for smaller companies to start taking up footholds in business that had traditionally been theirs. Therefore SDN is still today referred as a new thing, while at the same time the first standards on the subject are already almost seven years old.

2.1.2 SDN and data centers

Data centers are what made SDN more of a necessity. According to Chuck Black and Paul Goransson in chapter seven "SDN in the Data Center" of their e-book "Software Defined Networks", massive scale of data centers in 2010s threatened to break many networking technologies or at least introduce problems which must be solved by either re-designing network technologies or patching the issues one by one. (Black & Goransson, 2014) For example, server virtualization in data centers has increased compute nodes so massively, that MAC tables are overflowing in larger environments, despite the fact how heavy networking hardware has there been deployed. Spanning Tree's weaknesses have become intolerable in data centers; unused links by STP cause performance losses and convergence times are seconds instead of milliseconds, the scope where networks normally operate. Also the number of VLANs in multi-tenant/public cloud environments can exceed the limitation of 4096 currently in place not just by networking devices but also by the Ethernet standard's definition of having only 12 bits available for the VLAN Identifier in the 802.1Q tag inside an Ethernet header.

The issues are not only technical limitations in nature, however. In the bigger picture it looks like it is, and will be, impossible to bring any kind of real intelligence to the networks without having the understanding of the whole network's state. Because such solutions have not emerged during this time, it seems it is not possible with current independent, autonomous device model that is being deployed in networks. A centralized authority which has the view of the entire network and is able to make decisions which are appropriate to the situation is required, if traffic engineering is to be taken to the next level. In this scenario, traffic would not only be routed based on the shortest number of hops and possibly the maximum bandwidth, but the network's state, such as congested links along the shortest path in the upcoming hops.

Generally speaking, networking is the least advanced part of data centers today. Networking offers no agility, as it is not possible to quickly move networks from one physical location to another. There is no automation either in traditional networking, as it is impossible to make changes to the networks in programmatic methods. And also virtualization is not really part of the traditional networking, making it impossible to instantly create, destroy, and move network resources within the data center, where server virtualization allows such tasks for servers and services. This last weakness is being addressed by the SDN's sibling technology called Network Functions Virtualization (NFV) more than by pure SDN itself, however.

According to Black and Goransson, data centers were what really started pushing vendors towards the real implementations of SDN both academia and industry had been waiting for some time. Initially the issues described earlier were tried to be solved by

other means, such as increased data center management through orchestration solutions, virtual machine plug-in solutions, RADIUS-triggered automation on networks, using tunnels such as VXLAN and NVGRE to solve MAC address table and VLAN exhaustion issues, and creating new protocols to solve spanning tree issues such as Trill and Shortest Path Bridging (SPB), like listed by Rajesh Sundararajan in his presentation held at Interop, Las Vegas in 2012. (Sundararajan, 2012) None of these new fixes handled the whole problem field, many of their implementations were left in vendor specific state and some only fixed half of the problem in the first place. The need for SDN only strengthened, and while academia and open source projects had released some SDN environments and products during the years, it was not before 2013 when the larger vendors started getting ready to release their own platforms in a form or another.

2.1.3 The definition of SDN

SDN is not a strictly defined term. This is mainly because while OpenFlow-based SDN is the initial and currently most widely adopted approach, SDN's definition has always left space for other kind of approaches. Therefore Software-defined Networking's definition varies greatly on who is making the definition. Academia and vendors who have been part of the SDN creation process from the very start usually embrace the OpenFlow-style SDN, while established and incumbent vendors who are only trying to protect their current position on the market and usually see change as a threat would define SDN differently. For software-developers (such as the developers of an orchestration solution called OpenStack) or server vendors (such as VMware), SDN means very different things as well.

Despite different views on the subject, all definitions have a lot in common too. As a generalized SDN definition, the core idea is separating control and forwarding planes from each other, but not just within the devices themselves. The goal is to move control plane functionality to a centralized controller device (or software), outside the forwarding network nodes, which take care of data plane functionality. In the new working order, the network devices handle incoming packets using the hardware that has been programmed through a concept called "flows" to recognize packets and take actions appropriately. The controller on the other hand handles the more complex, compute-intensive functions that require more processing power and a network-wide view, and keeps installing these "flows" to the network devices as programmatic instructions on how single nodes in the network should function in each case. (Open Networking Foundation, 2014b) The SDN's idea is the fundamental change described by both ForCES and Clean Slate, and which was firstly implemented (in a specialized case) by the Ethane controller.

Because of this change in the paradigm, we can see that the network devices themselves are simpler, because they don't have all that control plane software running on them. This can be seen as a minor benefit also, possibly making the devices cheaper to manu-

facture after the full adoption of SDN, or at least making them more effective, freeing up hardware resources (memory, CPU, ASIC) from decision making to pure forwarding functionality. The centralized controller keeps configuring the network devices to make forwarding and routing decisions locally with no need to communicate with the controller, unless something unexpected in the network occurs, which has not been introduced and taught to the network devices yet.

Because of the fundamental way networking works (layered stacks, networking protocols running on top of other protocols etc.), it is clear that there has to be some kind of protocol for communication between the controller and network devices. In SDN realm, this protocol is a so-called southbound protocol, and the prime example of it currently is the protocol called OpenFlow.

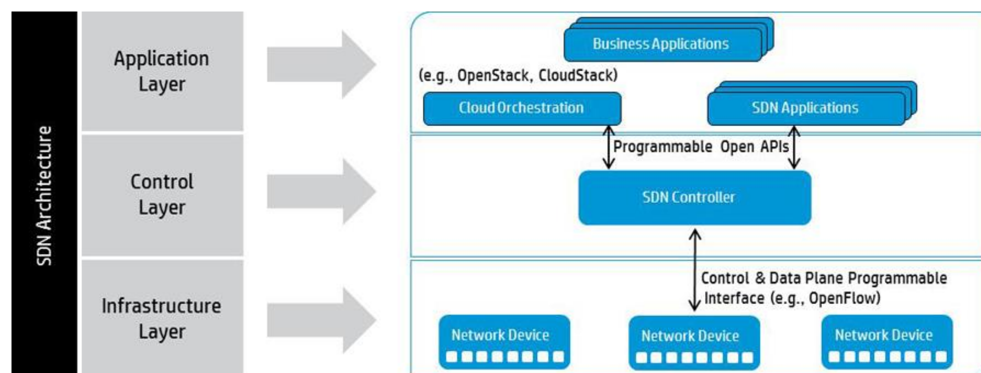

Figure 1. *Three-tiered SDN stack (Morgan, 2012).*

In the common SDN architecture, often expressed as a three-tiered stack as pictured in the Figure 1, the SDN controller, which in the SDN stack represents the Control Layer, also exposes itself to a “northbound” direction, offering programmable, and usually open application programming interfaces (APIs). These APIs are available in order for third party software developers to develop applications to the Application Layer, which will take responsibility in programming the rules to the network devices laying in the Infrastructure Layer. These rules are the basis on how the networking devices act in case of incoming packets with some detectable features, such as source address. These rules then are installed into the networking devices as flows. The applications which take advantage of the SDN controller’s northbound API can be newly developed external applications interacting with the controller via network connectivity and the exposed API. These APIs are usually formed in some standardized way, such as utilizing REST or SOAP architecture models, both of which have the upside of allowing the client application to be implemented in any programming language the developers are willing to use.

On the other hand, these applications could be integrated within the controller installation itself as well, depending on the controller implementation. Most of the controllers allow extending their functionality, features and even user interface by writing plug-ins,

add-ons or modules to the base controller. This work relies on the northbound API as well, but the API is offered in a different manner, allowing programmers to call internal native classes and methods of the base controller implementation. In this case, the programming language for the SDN controller extensions usually has to be the same that was used when developing the controller itself.

As a summary of the three-tiered SDN stack concept, the controller provides a set of APIs so that SDN applications can be written to the Application Layer, for controlling the behavior of the network. Because of the open APIs allowing various development approaches, existing business applications (such as network management tools and self-service tools, but also tools such as billing systems) could easily be integrated to the SDN stack by extending their functionality with features available in the SDN controller's northbound API.

The presented three-tiered architecture is the standard model of Software-defined Networking presented by the Open Networking Foundation. (Open Networking Foundation, 2012a) Vendors from different fields and approaches to the SDN have made small adjustments for their view on the SDN architecture. For instance HP has added a fourth layer to the stack above everything else, a Management Layer. This layer's point is to make clear distinction between SDN applications utilizing the SDN controller's API and thus making configuration changes to the network, in contrast with business applications and different kinds of orchestration tools such as CloudStack and OpenStack. Tools like those tend to live in a more abstract level and are handling a bigger picture than just adding, removing and modifying flows based on either external input from a third party system or the user, or newly gained information from the SDN controller (e.g. stats, counters, alarms). In the standard model all this would reside in the Application Layer, however.

Many other vendors seem to be keen in dividing the Control Layer into two separate layers, separating the parts of the SDN controller which offer functionality to southbound and correspondingly northbound directions. The naming scheme for this varies, but in the standard model these APIs reside in different sides of the Control Layer, which in itself contains the whole controller functionality from topology awareness to singular flow configuration mechanisms.

2.1.4 Claimed SDN benefits

A fundamental fact is that when SDN is deployed to an existing network, nothing changes instantly. SDN does not start making changes to the network by itself or by default. A SDN-enabled network, with an up-and-running SDN controller coupled with SDN-capable network devices functions exactly like a network that was not running SDN at all, if there is nothing special introduced to the application layer of the SDN stack. This is basically the state in which all SDN networks reside for some time during

the initial installation and configuration of the SDN network. This is because freshly installed SDN controllers don't have any kind of functionalities built-in on the application layer, and before anything on the application layer can be reasonably deployed, a SDN-capable network must be introduced to the SDN controller.

Generally speaking, SDN aims to solve multiple issues that have been appearing in the most complex and fastest-changing network environments first, such as datacenters, but which are not datacenter-related per se, and would eventually appear in other environments as well, like corporate LANs, WANs and backbones. While traits like agility, centralized management and programmability of the network describe SDN features very fundamentally, they are just that; features. These features lead to a certain set of benefits that is tried to address here. For this thesis, five key benefits for SDN are presented in the way Serdar Yegulalp summarized them in a Network Computing web publication about the original InformationWeek report in 2013 (Yegulalp, 2013).

According to Yegulalp, the main driver that is the key factor behind the success of SDN in data centers is the SDN's goal to enhance service provisioning speed and agility. The envisioned end-state of SDN's development as a technology is a state, where networks can be provisioned as quickly as virtual machine instances can be deployed with existing virtual hosts, such as VMWare ESXi and VMWare Workstation. This speed in service provisioning will not only benefit datacenters, but networking in a whole. In other words, the vision SDN is presenting is convincing, according to Yegulalp.

The second benefit Yegulalp claims is derived from the often-cited Stanford's and UC Berkeley's initial reasoning for the need for technology such as SDN; making it possible to experiment with networks and their configurations without impact in real-life production environments. In the current way of doing things, the protocol called SNMP makes it possible to do very limited dynamic experimentation on networks with scripted approach, but only with SDN's truly programmable nature network flexibility is high enough to allow holistic approach to network management.

The third identified benefit in the article for SDN is also very present in data centers, but not limited to them. Multi-tenant virtual clouds, where multiple virtual machines with highly different information security classifications and needs are run on the same physical hardware, behind the same physical connectivity, has made information security a huge issue with currently available tools and methods. SDN aims to offer more granular and fine-tunable ways to handle information security for applications, endpoints and BYOD (bring your own device) devices, than traditional hard-wired networks ever could.

While Network Functions Virtualization (NFV) is the technology which is traditionally claimed to cause savings in the networking business, according to Yegulalp's article,

SDN also aims to achieve some savings on operational expenses (OPEX) through increased efficiency via programmable management of the networks and replacement of manual work with automation. The methods and exact cost savings of SDN are still a little unclear, but it is clear that industry is expecting OPEX-saving from SDN too.

Last claim in the article, in addition to OPEX-savings, is that SDN aims to lower CAPEX (capital expenditure) as well. Here SDN has a multiple angles on cost savings. In the core ideology of moving the intelligence and decision making away from the network devices to a centralized, virtualized controllers installed on current-generation server hardware, the network devices would need to be able to do less in the future. This would mean that network devices wouldn't need to have as much hardware resources as today, resulting in cheaper network hardware. Furthermore, when SDN is coupled with its sibling, the other emerging technology NFV, virtualized networks both lessen dependencies on proprietary hardware and dedicated appliances, and make it easy to utilize the resources already owned and deployed to networks more effectively before the need for new hardware deployments are concretized.

2.1.5 SDN's road to existing networks

The fact that newly released technology has to be able to work with legacy technology has been taken into account in SDN design work. This is demonstrated by extensive controller support for use cases where SDN is deployed to existing networks in parts. The device at a time or a segment at a time –approach to SDN is very well supported by technology, but it even manages to create some new opportunities as well, as analyzed by Stefano Vissicchio et al. in their study for "Opportunities and Research Challenges of Hybrid Software Defined Networks" (Vissichio, et al., 2014). This deployment model leads to so-called hybrid networks, where only some parts of the network is working in agile SDN mode, and the rest of the network runs in legacy mode. Naturally, the SDN controller only gets visibility to the parts of the network that have been enabled in its domain for SDN functionality, but at the same time it understands the situation, where it doesn't have the full visibility to all parts of the network. Therefore intelligent decision making and network optimizing is still possible, even with such topologies.

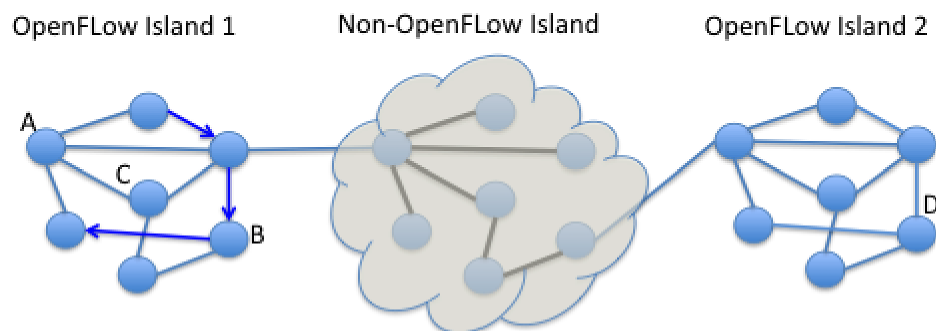

Figure 2. *A supported hybrid topology of one SDN platform (Wang, 2012).*

In the Figure 2 a use case of a hybrid network topology is presented. The presented topology would be fully supported by the open source SDN controller, Floodlight. Here the network can be seen as consisting of three so-called islands, of which two support OpenFlow-protocol, but can only access each other through an island in the network that contains multiple hops and devices, but doesn’t support OpenFlow. Handling scenarios like this paves much easier road for SDN’s introduction to the networking business, because in real environments replacing all the devices at once to enable even a possibly game-changing technology such as SDN just is not usually possible. Without approach like this, it would of course be possible to start deploying SDN in newly built environments, like in new office buildings which only get the latest hardware installed in them, but situations like these are rare.

As explained by James Sherry, according to Grand View Research’s market analysis the size of the SDN market in 2013 was $326.5 million. The same report suggests that in 2020 the size of the market will be $4909.8 million, corresponding to a growth of CAGR of 44.2%. (Sherry, 2014) Financially speaking SDN is definitely still in its starting phase, and there is still time to join the early adopters in the industry and start capitalizing on the technology relatively fast. It has to be noted that the SDN business does not only consist of developing and selling SDN-controller solutions. Some companies have not entered the SDN controller market at all with a product of their own, especially due to the fact how good the availability of open source SDN controllers is, but they are still supporting in their active network devices via OpenFlow-support or other means. The cited figures also include development of SDN-based services and applications, which is the area where carriers and telecommunication and other IT-companies operators are traditionally placed.

SDN’s adoption forecasts are backed up by the fact that real, known implementations of SDN, except for unknown number of datacenter deployments, are rare. This leaves room for implementations, as market is not even close to saturation point.

Google has developed their own inter-datacenter networks to run on a proprietary SDN, called Andromeda (Clark, 2014). The Andromeda is a pure OpenFlow-based SDN solution, but is not, and probably never will be available for general public. According to Timothy Morgan in an article posted at EnterpriseTech, Google has noticed significant benefits in network performance measured in data throughput of normalized bytes per second, where the SDN-enabled network was capable to work even four to five times faster than traditional network, when server CPUs were fully utilized. When normalized for a single CPU cycle, Google’s Andromeda-based network managed to throughput six times more bytes per second than traditional network when working with one stream, but benefits metered in this way showed signs of diminishing returns when increasing the amount of streams running in the network. With 200 streams SDN-enabled network was only two times more efficient than traditional network. (Morgan, 2014a)

As a summary, SDN is technologically viable for gradual deployment, it has claimed many conceptual benefits and can also show quantitative enhancements over traditional networks. Despite a number of promised and measured benefits of SDN, it seems SDN is still waiting for its "killer app" to really start driving its adoption in exponential speed. In this context the "app" doesn't necessarily refer to a SDN application per se, but could as well be the discovery of the most easily exploitable and capitalizable use case for SDN deployments. It could be a fusion of these two as well, like the scenario where Microsoft Lync calls are managed via SDN API in customer LANs with an SDN application designed for that purpose.

TeliaSonera has been looking for use cases for SDN development projects as well. The most promising lead in TeliaSonera's context is related to the company's Intelligent Business Network –program and more precisely to its feature, where network's functionality is managed based on mapping of recognized and pre-prioritized application groups and user groups, leading to differentiated treatment of different users in the network, regardless of their connection location or media. (TeliaSonera Finland Oyj, 2014) The Intelligent Business Network (IBN) –concept is technologically so advanced, that its vision's full implementation has not been possible before emergence of technologies such as SDN, but it has nevertheless already led to a pending patent based on its tool that dynamically creates and manages the user and application mappings, called Profile Manager. The program has been assessing earlier technologies such as SNMP, traps, syslogs and scripting as means to achieve dynamic network configuration, but these earlier technologies have not proven to be valid for large scale deployments. They merely allow some simple tasks such as shutting down and enabling single switch ports, but really changing the behavior of the network is on wholly another complexity level.

The next step within Intelligent Business Network –program and its approach to SDN would be to include dynamic network configuration to the differentiated treatment of users in the network. The current solution is based on pre-configured network segments, or VLANs, but in the future with the aid from SDN, the treatment of these aforementioned groups could vary depending on current users on the network and current loads in the links, as well as the currently used applications. As stated earlier, functionality like this has not been feasible to implement with traditionally available techniques, such as SNMP or CLI. With SDN however, it would essentially be possible to make the currently available Profile Manager tool an SDN application, making it use the SDN controller's northbound API to propagate changes to large segments of customers' networks.

2.2 Programmability and APIs in general

In the SDN model, programmability of networks is achieved via a set of application programming interfaces, or APIs, which reside in the SDN controllers. These APIs make it possible to both read and write information from and to the SDN controller, which then implements the changes based on this information to the underlying network.

The currently prevalent trend by far of implementing external APIs to applications in general is the REST architectural model. It is clear that most of the SDN controllers have adopted this mechanism also, sometimes in addition to some other kind of APIs, such as SOAP, RPC or the option of directly exposing Java methods and classes to third party developers. Some of the most significant techniques are explained in the following chapters.

2.2.1 REST

REST is an architectural style for programming applications and web services. REST was initially presented by Roy Fielding in his dissertation in the year 2000 (Fielding, 2000). As a summarization, the key principles of REST are the following:

- Every resource has its unique identified, such as a URI
- Resources are linked together, forming a web of inter-resource relationships
- Standardized methods have to be used (e.g. HTTP, media types, XML)
- Resources can have multiple representations, which represent the states of the application
- Messaging must occur in a stateless fashion, using HTTP

From here it can be gathered that one key concept in REST architecture is a resource. Servers contain resources, which are accessible and manipulatable by client applications. Any pieces of information that can be named, are a resources. Depending on the subject of the implemented API, a few examples of resources could be for instance a switch, a data flow and the amount of data throughput between points A and B in last minute.

Resources like these always have unique identifiers, which in practice usually is URI. This means that every piece of information has its own URI – both the idea of a switch within the system, as well as a single dataflow between points A and B during certain timeframe. Usually these resources are coupled with the information about the media type the resources should be accessible in. In its most simple form, this means that an image could be JPEG-binary data (JPEG: Joint Photographic Experts Group, a lossy image compression method), while a document could be an HTML document, and so on.

REST architecture is also based on client/service –model, in which both requests by the client as well as the responses by the server are built as the representations of aforementioned resources. Here the representation of a resource means that a single resource at a given point of time corresponds to the state of the resource; for instance the current data flow counter's value in bytes for a certain data flow.

2.2.2 XML and JSON

As was defined in the description of REST, a RESTful web services must rely on standardized data interchange formats. Most prevalent examples of these are XML and JSON.

XML (Extensible Markup Language) is a text-based markup language that has been designed to describe data in a way that would be both human-readable and machine-readable. It has fast become the de facto standard for data interchange on the web. In XML, data is identified using tags, which are identifiers enclosed in angle brackets "<" and ">". Collectively, tags are known as markup, and inside tags can reside whatever data the markup has been designed to represent. (Bray, et al., 2008)

JSON (JavaScript Object Notation), much like XML, is a way to store information in both human-readable and machine-readable collections of data. In JSON, any number of properties can be declared using name/value –pairs, separated by commas. In most implementations of JSON, the value corresponding to any given name is accessible by simply referring to the name of the property in question. (Atif & Scott, 2007)

The difference between JSON and XML is in most cases pure preference, but as a generalization JSON can sometimes have less overhead in its markup than XML, and with most commonly used tools single elements' values are sometimes easier to access compared to the XML markup.

2.2.3 Introduction to OpenFlow

Unlike REST, XML and JSON, which are essential in the SDN stack's northbound direction, OpenFlow is the standardized protocol which implements the southbound functionality in the pure SDN defined by Open Networking Foundation. OpenFlow is used to carry messages between network devices, such as switches, and their SDN controllers. Communication is bi-directional and can be initiated by either end of the stack. OpenFlow supports both open and TLS-encrypted communication. (Open Networking Foundation, 2012a)

2.2.4 OpenFlow 1.0

In OpenFlow 1.0, and OpenFlow in general (with minor additions to the scheme in the later versions), stores information on the switch in the form of Flow Entries, which together constitute a Flow Table. Flow entries consist of three major parts: match fields, statistics and actions.

As described in the Open Networking Foundation's whitepaper OpenFlow Switch Specification (Open Networking Foundation, 2009), match fields are like conditions, which are used to match against incoming packets on the switch. If a condition matches, that flow's actions will then occur for that data flow. Match fields in OpenFlow 1.0 consist of a 12-tuple, including the ingress port of the packet on the switch and eleven fields which come directly from the Ethernet header: e.g. MAC source and destination, IP source and destination, TCP/UDP ports, and so on.

Statistics of a flow entry keep track of counters for each flow, while a flow entry's actions are the instructions which are performed if the incoming packet matched the specific flow entry. Some examples of flow actions would be outputting the packet to a specific port, dropping the packet, or giving the instruction to treat the packet as "normal", meaning that the network device should treat the packet as it would if whole SDN scheme was not deployed at all in the network, and flooding the packet to all ports (for use cases such as multicast or broadcast in the network).

Flow tables are ordered list of flow entries, against which every incoming packet is matched in the device's ingress port. This setup resembles a lot the configuration of a firewall, where rules are read from top to bottom. If any flow entry is successfully matched on the packet, rest of the entries in the flow table are never validated against that packet. If no matches are found, in OpenFlow 1.0 the packet will be forwarded to the controller for decision making on what to do with the packet.

2.2.5 OpenFlow additions in versions 1.1, 1.2 and 1.3

OpenFlow development progressed in such a way that OpenFlow 1.1 and OpenFlow 1.2 were never really implemented or supported any of the real-life SDN controller generally available. It was not until OpenFlow version 1.3 when the controller developers and rest of the SDN community recognized the need for adoption of a new OpenFlow version.

Each version incrementally enhanced the protocol in significant ways, however. In OpenFlow 1.1, support for multiple flow tables within single network device was added, in contrast to the model explained for OpenFlow 1.0. This made it possible to create conditional matches in an effective way without flooding the flow tables with flow entries with only minor differences in multi-part match fields. Ironically, while SDN was

designed to solve MAC table overflow issues, OpenFlow's version 1.0 was very easy to get to overflowing with flow entries in quickly and very dynamically managed laboratory networks. This change in flow table model led to some other insignificant, but necessary changes in flow actions and match fields, as well, breaking backwards-compatibility for the most part at the same time. One significant addition was the support for MPLS protocol, however, which is used in critical enterprise connectivity instead of IP. (Open Networking Foundation, 2011a) The usage of MPLS is common in enterprise-sized companies in Finland, also.

OpenFlow 1.2 added two key features to the protocol; the support for IPv6 and the built-in support for extensibility within the OpenFlow standard. This means that developers and vendors would be able to extend OpenFlow with the attributes of vendors' choosing without breaking the standard, both for matching and applying actions. (Open Networking Foundation, 2011b)

OpenFlow 1.3 was probably the least significant of the upgrades, adding only support for Provider Backbone Bridging (PBB). (Open Networking Foundation, 2012b) PBB is needed by service providers who have packets which traverse provider edges and must get encapsulated and tagged as they pass through the provider network. The significance of this addition is small, and OpenFlow 1.3 did not end up being the version adopted by industry because of that, but most likely because of timing and cumulative list of development becoming significant enough for actors in the field.

2.3 SDN programming interfaces

In the three-tiered SDN stack, and in available SDN controllers in practice, two tiers of APIs are used. (Ferro, 2012) The so-called southbound API is the less interesting from this thesis' point of view, implementing the interactions between network devices and the SDN controller. While OpenFlow is an open standard and open protocol, in real-life implementations it is not accessible by a programmer, but built as a closed and secure channel between these two end-nodes.

OpenFlow is not the only allowed protocol for southbound functionality, either. Some other, sometimes a lot older protocols which initially were designed some different thing in mind, are used for implementing the southbound API of an SDN controller. Examples of protocols like these are NETCONF often coupled with YANG and XMPP. Also CLI, meaning simply having a terminal connectivity via SSH or telnet –connection to the devices, is used in some SDN controllers (such as Cisco System's SDN controller). (Gourlay, 2014)

SDN controller's northbound APIs are the ones which are more interesting from this thesis' point of view. They are the part in the SDN stack that truly makes it possible to

programmatically control the network, offering open interfaces for third party developers or anyone owning the SDN controllers in question, in order to develop their own functionalities, behavior and features to their SDN network. The market situation seems to be surprisingly uniform in the implementation of these northbound APIs, most of the key vendors relying in the REST-styled API for their northbound interfaces. Details vary, such as the data interchange formats and the scopes of the APIs, and some vendors do offer some additional ways of interacting with the controller with program code, but REST seems always to be present as a choice.

The idea of controllers' northbound APIs is to expose basically all of the OpenFlow's features to the programmers. This means that within the scope of OpenFlow features, programmer gets almost unlimited access for manipulating the network's behavior based on matches, statistics and actions, limited only by the OpenFlow versions limitations (e.g. one cannot control IPv6 networks with OpenFlow 1.0 –capable SDN controller). The benefit of using REST for the northbound API implementations is the fact that SDN applications can then be written in any language capable of accessing URIs with HTTP methods GET, PUT, DELETE and POST.

2.4 Standardization

The southbound interface, regardless of solution implemented in a specific controller, is very well standardized. OpenFlow is the most commonly used standard, but all the other encountered techniques in southbound direction rely on standards as well. This leads to a situation where underlying network devices in the SDN-controller network should be interchangeable and non-vendor specific.

Northbound direction is more problematic. There is a lack of standardization in the area, despite the fact that Open Network Foundation has a working group called Northbound APIs. (TechTarget, 2014) The working group has no releases, however, and the situation has escalated into one where every vendor has their own proprietary northbound API, with varying feature sets, documentation and usability.

The situation is nothing new for web service developers, however. Most of the time web services are developed against non-standard APIs, as situation-specific solutions with help from API-specific documentation and code examples. Vendor lock-in is a constant issue in physical networking business, where huge CAPEX spending is directed towards acquiring and owning physical devices, and therefore the issue may be larger from the network operator's point of view than it normally is for web developer or software development company, which mostly work with virtual assets (such as program code).

2.5 Network Functions Virtualization (NFV)

Network Functions Virtualization (NFV) is a technology that is forecasted to have great synergies with SDN. NFV is not within the scope of this thesis however. These technologies are separate in such a terms that either one can be deployed in any environment without the other one, but together they're expected to add significant value to the value chain.

The core idea of NFV is to bring commonly implemented virtualization techniques to network equipment, such as routers and switches. Virtualization would allow more efficient use of installed resources and more flexible allocation of these resources.

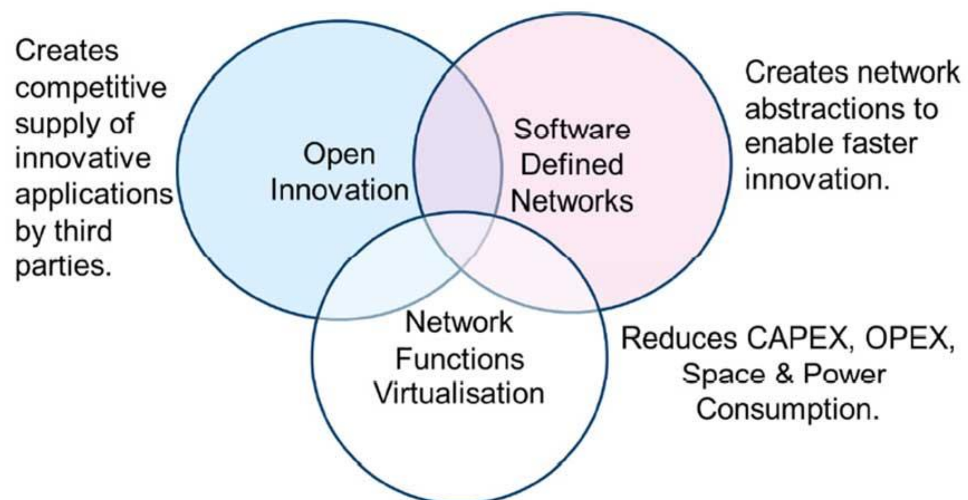

Figure 3. *Relationship of SDN, NFV and developer community (Salisbury, 2012).*

Basically, as depicted in the Figure 3, SDN can be thought to allow network functionality development accelerate to a level of speed and agility never before possible, when programmability of the network makes networks manipulatable through abstractions in northbound APIs. In its most crudest form, NFV can be seen as expense reducer, but virtualized network equipment are also more agile to deploy and manage, making the change management speed to have another accelerator on it. One way of abstracting the situation is stating that while SDN brings automation, NFV brings up-to-date provisioning, and only the combination of SDN and NFV can bring automation and provisioning together, leading to a world where networks behave in a manner we've used to seeing from virtualized servers.

3. SDN CONTROLLERS AS NEW FUNCTIONALITY ENABLERS

3.1 Assessment of available SDN controllers

Assessment of SDN controllers is especially interesting because as a generalization on the idea level, SDN controllers are designed to be interoperable and interchangeable. Open Network Foundation has designed the SDN architecture in such a way that so-called vendor lock-in should never occur when building, extending or modifying SDN networks. Many vendors even use this as a marketing point in their SDN solution marketing materials (Juniper Networks, Inc., 2014a). This is achieved through extensive and clear standardization situation in the southbound application programming interfaces. This means that one SDN controller can be replaced with another, and in theory this should be possible without any kind of changes in the configurations of the existing network devices that are being controlled by the SDN controller.

In practice there may be some minor work needed, which might be related to for instance different OpenFlow versions (from 1.0 to 1.3 and forth) implemented by the controllers or supported by the network devices. It is also important to always remember that SDN is not synonymous, nor is it limited to only OpenFlow standard. Even though OpenFlow is a standard created by Open Network Foundation, ONF has at the same time designed the whole SDN architecture in such a way that OpenFlow is not a forced component in SDN stack, but rather an option among others. As was discussed in chapter 2.3, there are other competing ways in addition to OpenFlow to implement the southbound interfaces in SDN architecture stack. Some existing methods are XMPP, NETCONF, YANG and even CLI, or some kind of combination of these technologies. More competing standards might very well rise in the future.

3.1.1 The big three: Juniper, Cisco, Hewlett-Packard

The interchangeability situation is far worse when the SDN network is designed from the beginning to be using one of the mentioned competing technologies. Especially the CLI-solution is very unsophisticated and regressed. Basically the CLI-solution only works with a certain set of network devices and can even stop functioning after software updates in network devices, if these updates remove, add or modify some command line commands within the network device’s CLI, command line interface. XMPP’s and NETCONF’s problem is that even though they are well standardized messaging protocols, in practice the support for these technologies vary greatly in network devices.

3.1.2 Cisco APIC

There is at least one example of a vendor who is not pursuing OpenFlow standard with interest but rather relying on older standards, proprietary definitions of protocol implementations, and the mentioned CLI-solution: Cisco Systems, Inc.. Cisco's SDN-architecture is called Application Policy Infrastructure Controller (APIC), which can be generalized as a solution that works with Cisco devices only. APIC is a part of the Cisco ONE –platform, which is a pre-existing network management and configuration platform Cisco has built their features on for the last few years.

Especially the usage of CLI (command line interface) as a method of achieving machine-to-machine interactivity can be seen very controversial and even unreliable and hack-like. Basically this means that APIC uses telnet/SSH sessions to connect to network devices one by one and issuing commands automatically to their command line structures with vendor, model and software version number specific commands. As of September 2014 Cisco APIC only supports a part of Cisco's own product portfolio and does not support any of competitors' network devices. Building a proprietary and/or closed source solution is not a problem from the SDN application developer's viewpoint, but when such a SDN platform is restricted to function with only one vendor's networks, problems arise.

In addition to CLI-solution, Cisco's APIC includes a Cisco's self-designed protocol for southbound interactivity: OpFlex. OpFlex is basically a Cisco-proprietary counterpart of OpenFlow (Duffy, 2014). However OpFlex is followed by the same vulnerability that the CLI-solution presented; no other vendors' network devices support it. It's possible that this changes over time, but a strategy of creating own standards usually only leads to a situation where there exist a numerous standards to achieve the same thing, as can be seen for instance with network monitoring technologies such as NetFlow (Cisco alternative), jFlow (Juniper alternative) and sFlow (sFlow.org Consortium's free of charge and royalty free alternative driven by actors such as HP, Extreme Networks and Hitachi).

There are some signs, and also to some extent contradictory information, that Cisco might introduce some sort of partial support for certain OpenFlow functions into its platform, but it is still reserved for future releases, and thus its inclusion cannot be yet treated as a fact. For instance Cisco Systems' own press release poster on January 2014 claims that OpenFlow is among the supported southbound API's (Cisco Systems, Inc., 2014) and same claim is brought up by Zeus Kerravala in a research paper published by ZK Research in 2014, both in-text and in a diagram on the page five (Kerravala, 2014). The contradictory part is that Cisco's own whitepaper on The Cisco Application Policy Infrastructure Controller has only a single mention of OpenFlow, where it compares APIC to OpenFlow-based SDN controllers with the phrase "unlike an OpenFlow con-

troller". (Cisco Systems, Inc., 2013) Most likely scenario is that Cisco has borrowed some elements from the OpenFlow technology, but on a practical level they do not support any referable OpenFlow version. The generalized take on the situation is however, that Cisco Systems is against the usage of OpenFlow in SDN solutions due to performance issues (Morgan, 2014b).

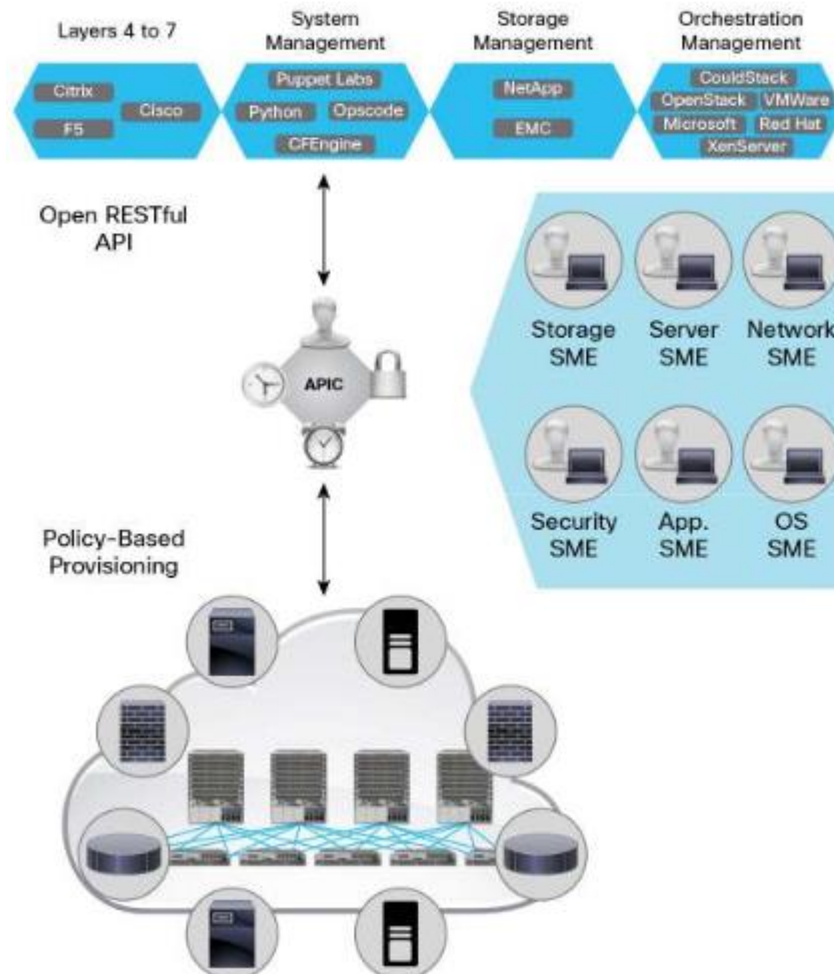

Figure 4. *Cisco APIC architecture (Cisco Systems, Inc., 2013).*

As shown in the Figure 4, Cisco's APIC architecture offers Open RESTful API in to the northbound direction, which Cisco has named as ONE DevKit. Architecturally, Cisco's APIC solution resides on top of a system called Cisco Application Centric Infrastructure (ACI), which itself is an architecture that offers a common policy management framework. By this Cisco means that ACI is the underlying structure that enables APIC to manage, orchestrate and even virtualize (especially in the case of NFV) the network. Practically this means that ACI enables APIC to call certain API methods that translate into configuration changes in the network or that ACI is the underlying structure that enables programmability of the network.

Also another weakness of the Cisco solution and a part of the reason why APIC's relationship with OpenFlow is still an open question, is the fact that they are late to the market. Cisco's solution will be available for first customer shipment in the end of third

quarter of 2014. (Matsumoto, 2013) For the purposes of this Master of Science thesis, an EFT (early field trial) version of Cisco APIC-EM was acquired from Cisco for testing purposes in laboratory environment.

However, the explained proprietary and closed approach to software-defined networking that Cisco Systems' solution implements makes APIC-EM not very interesting within the scope of this thesis. It can be said that Cisco's SDN breaks, or does not even try to achieve, one of the fundamental ideas of the SDN: the controller interchangeability and cross-vendor functionality. It can be claimed that this poses one of the main threats to SDN as a technology, too. If the big vendors do not respect the open concept of SDN, it can cripple the whole technology for years to come.

Like some other products too later in this thesis, Cisco's APIC codebase is based on the open source SDN controller called OpenDaylight. Cisco has however heavily modified the OpenDaylight code, added their own features (such as support for onePK southbound protocol) and removed some others (such as most other southbound API protocols).

In April 2014, Cisco announced it has designed and implemented a competitive standard for OpenFlow titled OpFlex. OpFlex's difference with OpenFlow is the fact that OpFlex only concentrates on pushing policies (basically new network rules) into hardware devices, which after that make the decisions themselves, while one of the OpenFlow's core ideas is to decouple control logic away from the hardware device, into the intelligent controller. OpFlex is not yet supported by Cisco APIC, however. (Ramel, 2014)

3.1.3 Juniper Contrail and OpenContrail controllers

While it is clear that Cisco is the major incumbent networking company in the world with revenue of US$ 48.6 billion in 2013 (U.S. Securities and Exchange Commission, 2013), Cisco obviously has some formidable competition as well. According to Jim Duffy in NetworkWorld, the ten most significant competitors for Cisco include companies listed as HP, Alcatel-Lucent, IBM, Juniper, Aruba, Polycom, Avaya, Microsoft, Check Point and Brocade (Duffy, 2010). The competitor field is quite fragmented, some of the mentioned companies working only in limited areas (such as Aruba in wireless networking) and some others doing networking only as a side business (such as Microsoft and IBM). Most commonly two names pop up when speaking about most significant networking companies besides Cisco; Juniper Networks and Hewlett-Packard. No other company can compete with Cisco in all networking fields, but Juniper with the revenue of US$ 4.7 billion in 2013 (Google Inc., 2014a) is very formidable number two in core Internet routing with its 30% market share, while HP (revenue of US$ 112.3 billion in 2013 (Google Inc., 2014b)) is taking larger and larger role in enterprise switching in Local Area Networks (LANs), being number two after Cisco in that market

with global share of 10%. The small percentage only highlights Cisco Systems' incumbent position. Regarding SDN, it can be said that the competition is not imitating Cisco's approach very closely at all.

Juniper Network's SDN controller solution is called the Contrail Controller. Unlike Cisco Systems, Juniper embraces the open source ideology with their controller by continuously co-releasing an open source version of their controller, called OpenContrail. Even though an open source version of a controller is not a requirement for embracing Open Network Foundation's SDN-standard fully, it is a promising sign about the direction where the company is headed with their SDN development.

However, unfortunately just like Cisco Systems, Juniper Network is not supporting OpenFlow at all with their controller's current versions. Juniper's main method of achieving southbound interface functionality is based on XMPP. Juniper representatives have given comments in media which tell that they are looking into incorporating OpenFlow into their controllers in the future. (Morgan, 2013) If and when this will happen, is still very much unknown.

If we look at Juniper's OpenContrail more closely, we can see that the northbound direction is implemented via a REST API. The API at this stage is pretty extensive, even though a big portion of the methods are more related to Network Functions Virtualization rather than plain SDN (e.g. controlling the traditional, physical and existing, current networks with software, rather than building virtualized networks via software). For instance the OpenContrail's REST Tutorial discusses the creation of a virtual network as the first and only use case presented for using the REST API, and the REST API has most of its methods touching subjects such as virtual routers, virtual machines and virtual networks (Juniper Networks, Inc., 2013). As we described in chapter two, NFV is a technology and a principle which can be enabled and enhanced with SDN, but neither one is a requirement for each other. For organizations and/or people looking for a plain SDN solution with no NFV needs in mind, the Juniper's approach where NFV is fully integrated all around their SDN architecture can be confusing and even impractical.

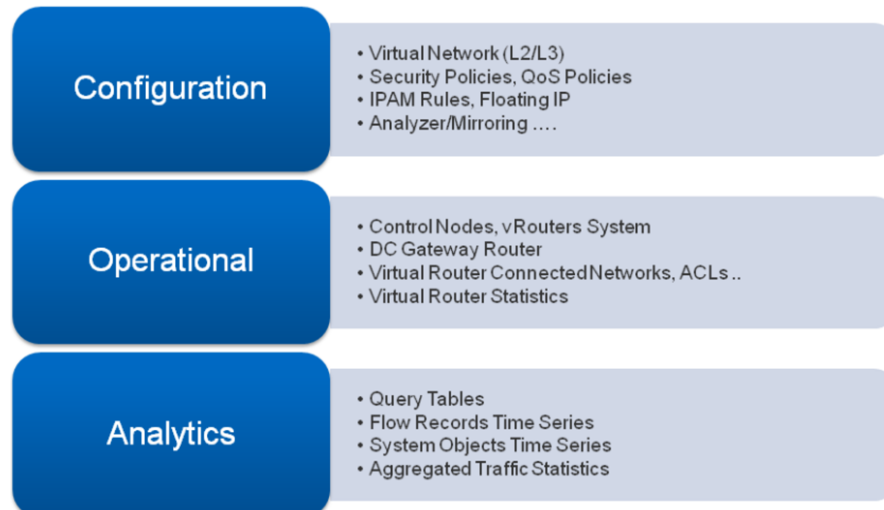

Figure 5. *OpenContrail's Northbound API focus areas (Juniper Networks, Inc., 2014b).*

According to the API documentation, the northbound REST API does contain a moderate amount of methods for use cases around generic SDN functionality, though. As described in the Figure 5, OpenContrail’s API is clearly divided into three sections: configuration, operational and analytics parts. The configuration sub-API contains ways to achieve at least Quality of Service –functionality and some degree of IP address management (IPAM). Operational part then again is mainly aimed for virtual routers, but it has methods to control any attached southbound nodes, such as physical network devices which support XMPP. Analytics–part of the Juniper’s northbound API offers at least ways to read aggregated traffic statistics and flow records in time series.

It is clear that Juniper’s Contrail–environment has a usable Northbound API which would allow the creation of applications of some kind, but its focus area is seriously different what is expected from the API in the use cases this thesis is seeking to solve.

The lack of clear differentiation between SDN and NFV can be easily seen in Juniper’s official OpenContrail Architecture Documentation too, where Juniper presents only two use cases as the drivers for Software Defined Networking: cloud networking and NFV. (Singla & Rijsman, 2014) Even though Juniper mentions in their documentation that OpenContrail is an extensible system that can be used for multiple networking use cases, it is clear Juniper’s primary motivation for SDN is not in the area of enabling the development of un-predetermined network service functionality, but rather creating effective tools for cloud and NFV realms, which are closely tied together.

OpenContrail.org, the official home for the open source implementation of Contrail Controller goes even as far as hosting a descriptive HTML slideshow on its site, which is titled as “OpenContrail Network Virtualization Architecture – Deep Dive”. (Juniper Networks, Inc., 2014b) The fact that they even call their SDN solution a Network Virtu-

alization Architecture, rather than a SDN controller, implies alone that their focus area is a lot towards NFV.

3.1.4 HP VAN SDN Controller

The situation where one or two (or all) of the big three networking companies are acting as a brake for progress is not unique for SDN, but rather a very many times experienced situation where companies with the leading market position are trying to uphold their positions by trying to resist change on market. This phenomenon has been witnessed in business regardless of the industry, and has been closely analyzed in e.g. European airline industry by Viellechner and Wolf in their HHL Working Paper “Incumbent Inertia Upon Disruptive Change in the Airline Industry: Causal Factors for Routine Rigidity and Top Management Moderators” (Viellechner & Wulf, 2010).

Hewlett-Packard’s solution in the SDN area is the controller product called HP Virtual Application Networks (VAN) SDN Controller. Among the trio of Juniper, Cisco and HP, the HP solution has been on-market for the longest time, it being released for general availability in November 2013.

HP VAN SDN Controller differs from Cisco and Juniper solutions especially by its direct and extensive support of OpenFlow protocol towards its southbound API, as shown in the Figure 6. This is not surprising, because HP has been heavily invested in the Open Networks Foundation’s work in designing and specifying both SDN and OpenFlow from the early years. According to Hewlett-Packard technical whitepaper titled “HP Virtual Application Networks SDN Controller”, HP was the first company in the world to demonstrate hardware-based OpenFlow-enabled switch at ACM SIGCOMM in 2008, within a year of the original publication of the SDN ideology at Stanford and Berkeley Universities (Hewlett-Packard Development Company, L.P., 2013a).

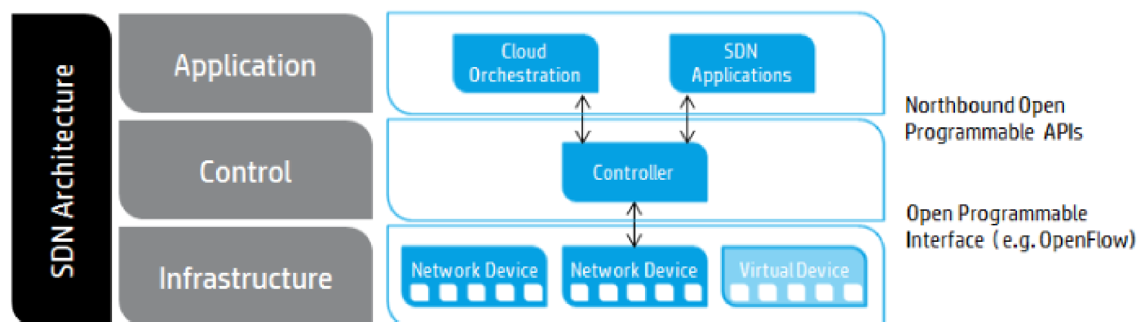

Figure 6. *HP SDN controller's interaction principles to southbound and northbound directions (Hewlett-Packard Development Company, L.P., 2013b).*

Unlike many or most of the OpenFlow-supporting SDN controllers on the market, HP’s OpenFlow-support is not based on OpenFlowJ-library, which is essentially an open source community supported version of OpenFlow in Java and was the de facto choice

for OpenFlow implementations in SDN controllers during OpenFlow's early versions. OpenFlowJ's weakness is that it only supports OpenFlow versions up to version 1.0, which is a limitation that HP VAN SDN controller has overcome by developing their own messaging library. As of time of writing, HP's VAN SDN Controller supports OpenFlow versions 1.0, 1.1, 1.2 and 1.3, and promises easy extensibility for future versions of OpenFlow, of which OpenFlow 1.4 is already available.

HP's SDN controller's northbound API offering is based on both Java and REST API technologies. Java API offers deeper access to time-critical functions of the SDN controller, while the REST API offers higher abstraction level and allows more rapid and flexible development. This dual offering of two APIs on different abstraction levels is not very common among SDN controllers, but can be counted as a definitive benefit for HP's platform.

HP's take on SDN architecture is also a lot clearer than for instance Juniper's mix of SDN and NFV principles. While this could be counted as a minor weakness if the solution would be straightforwardly used for NFV, as a general platform HP's internal build makes more sense. Most clearly this can be seen in the HP SDN VAN Controller's REST API's methods, which are very generalized and useful in all kinds of use cases. Also the API is larger in scope than either one of the larger competitors, Cisco or Juniper.

3.1.5 Other significant actors and open source controllers

The on-market SDN solutions are not limited to solutions created by the big three, however. There are at least three significant and continuously developed SDN controllers which are released as open source: Project Floodlight, OpenDaylight and the duo of NOX and POX controllers. Of these both Project Floodlight and OpenDaylight are also released as paid, closed source versions by major vendors participating in their open source development; IBM's proprietary SDN controller is based on OpenDaylight and Project Floodlight acts as a base for Big Switch Networks' Big Network Controller.

3.1.6 Project Floodlight

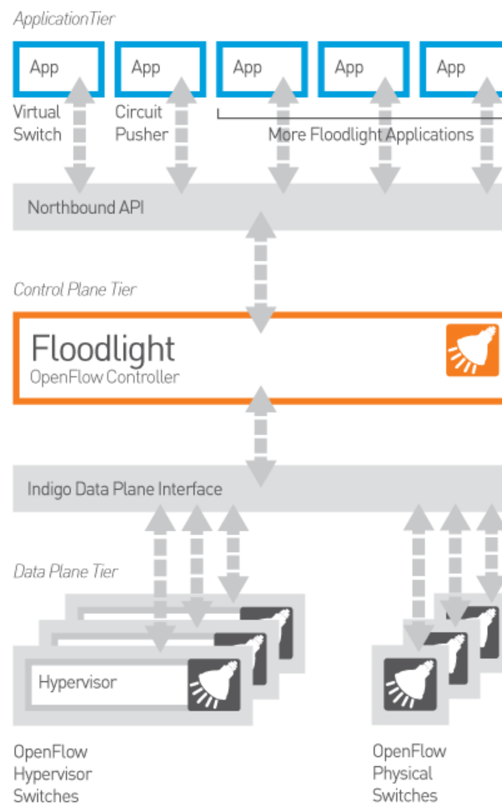

Figure 7. *Floodlight SDN controller's architecture (Project Floodlight, 2014a).*

As shown in the Figure 7, Floodlight Controller supports OpenFlow for both virtualized and physical switches. In reality this shouldn't matter much to the controller: a virtualized switch could in theory be used in a transparent mode where the controller would not even know it was interacting with a virtualized device instead of a physical hardware. In this case, however, the controller wouldn't obviously be able to use more advanced virtualization features and move the SDN realm closer to the NFV ideology.

As explained before when discussing the Hewlett-Packard solution, most of the OpenFlow controllers on the market haven't yet evolved past the OpenFlow version 1.0. This is the case with Project Floodlight; it only supports OpenFlow version 1.0 and according to Floodlight's wiki-based FAQ, the timeline for support of OpenFlow 1.2/1.3 or beyond is currently unknown (Project Floodlight, 2013). This doesn't pose a significant problem however. As described in the chapter 2.2.3, many early implementations of OpenFlow-based networks deemed already OpenFlow version 1.0 suitable for even a large scale use, if a bit lacking in advanced features.

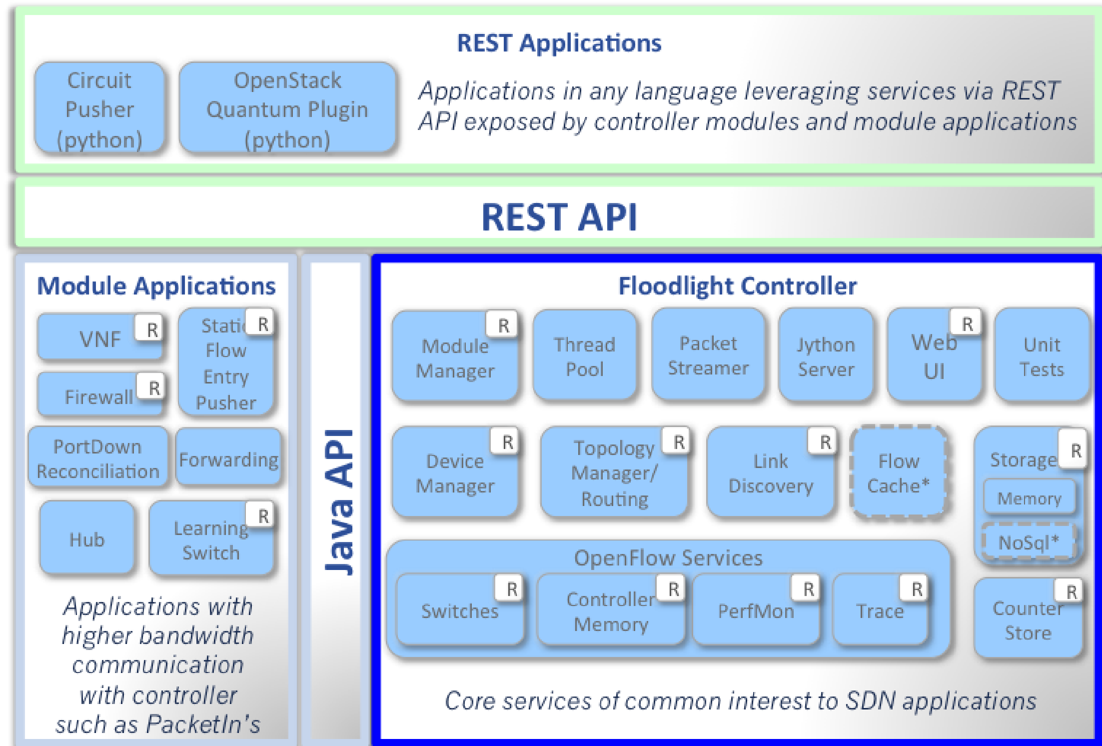

Figure 8. *Floodlight Controller's API offering (Project Floodlight, 2012).*

As described in the Figure 8, Floodlight controller's northbound API structure is very similar to Hewlett-Packard's architecture. Floodlight offers both Java and REST APIs with very similar distinction that HP uses. Floodlights documentation points out that Java API should be used for applications with higher bandwidth needs, unlike HP which pointed out the low-latency requirement as the primary factor, but both are performance related limitations. REST API then offers the possibility of development of applications in any language, leveraging the high-abstraction methods of the public and open interface.

While the Floodlight Controller's REST API is not even close to as extensive as HP's implementation is, it is documented very well in Project Floodlight's Wiki, and it contains relevant methods for basic network management; a number of different statistics, a flow pusher API in order to control data flow directions (for the sought use case where a traffic is rerouted towards intrusion prevention system in case of an anomaly) and for instance possibility to edit flow priorities. So in theory and based on the documentation, Floodlight's API would be sufficient for the use cases in this thesis.

As a showcase of Floodlight's usability in real-world heavy use scenarios, Floodlight was the controller of choice by Caltech for the SDN implementation at CERN's Large Hadron Collider (LHC). (Project Floodlight, 2014b)

3.1.7 OpenDaylight

While Project Floodlight development is mainly steered and supported only by a single company, Big Switch Networks, OpenDaylight project has significantly broader company support, including all the aforementioned three big networking companies who also have their own SDN controller solutions, of which none is based on the OpenDaylight directly or indirectly.

Possibly due to the large base of company support, OpenDaylight development seems to progress many times more rapidly than Floodlight's, and OpenDaylight seems clearly to be the most ambitious project of all the open source SDN controllers available, which should be clear from the scope of the controller entity depicted in the Figure 9.

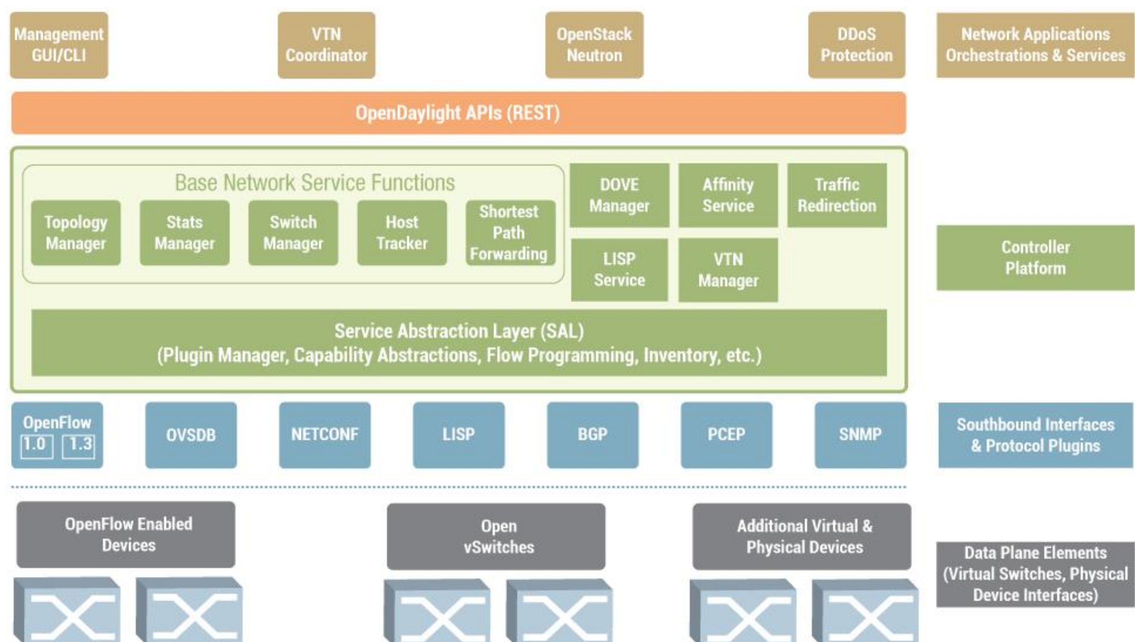

Figure 9. *OpenDaylight controller architecture (The OpenDaylight Project, Inc., 2014a).*

As shown in the, OpenDaylight SDN controller offers RESTful APIs in the northbound direction, like all the other SDN controllers have done too. What differentiates OpenDaylight from Floodlight is the extent of the REST API, which is easily comparable to the Hewlett-Packard implementation. This can be seen from the logical division of the REST API into different modules, and the plain amount of methods available. Also the state of the online documentation is at least as good as is with HP solution, and clearly better than any of the other competitors.

Another area that brings OpenDaylight on par with HP solution is the support of OpenFlow versions. Both versions 1.0 and 1.3 are supported, which is a logical and suitable limitation: OpenDaylight allows the use of either initial version for legacy environments or the latest version for developmental environments and state-of-the-art production

networks. It is easy to admit that the OpenFlow version support for versions in-between is not as interesting as exactly the version 1.0 and 1.3 are.

In addition to OpenFlow, OpenDaylight offers other southbound interfaces and protocol plugins too, such as NETCONF and SNMP. The support for different technologies is vast, but the situation is not unlike HP's solution, which in addition to OpenFlow, offers L2 and L3 Agent and DNS/DHCP APIs implemented with REST architecture and a device abstraction API implemented via proprietary device drivers towards southbound direction as well.

Even though the architectural figure doesn't implicate it directly, OpenDaylight controller also offers what OpenDaylight Project calls a level 2 API, which is essentially the equivalent of HP's and Floodlight's Java APIs. The difference in philosophy exists between HP/Floodlight –style Java APIs and OpenDaylight's implementation: while HP and Floodlight aim to raise the abstraction level for REST API, OpenDaylight claims that in their Java API, the set of accessible functions should be same as in the REST API (The OpenDaylight Project, Inc., 2014b). It is expected this will cause differences in SDN application development between the platforms.

While Juniper has its own SDN solution in both paid and open source formats, it has also, at least on paper, been a participant in a Linux Foundation Collaborative Project open source SDN controller project entitled OpenDaylight. While news agencies have reported that during the years Juniper has been a member organization of the project, it has refused to participate in the development in the extent Juniper's been expected to, and Juniper has seen OpenDaylight as more of a threat than a project they want to invest to, as summed up by Mitch Wagner in article posted at LightReading in 2014, in April of 2014 Juniper submitted a OpenContrail plugin to OpenDaylight project (Wagner, 2014). The idea of the plugin is to allow co-existence and coordination of these two controllers. According to the architectural diagram on OpenDaylight Wiki, the plugin adds an OpenContrail plugin to the southbound interface of the OpenDaylight controller, meaning that the plugin allows OpenDaylight to control OpenContrail, while the opposite would be mostly not true.

3.1.8 Big Switch Networks Big Network Controller

As explained earlier, Floodlight controller is available for download free of charge and is an open source project, but the company driving the project, Big Switch Networks, has packaged a proprietary version of it under the name Big Network Controller. Because the solution is closed-source, it is hard to verify how much of the Floodlight's code base is kept intact in the Big Network Controller, but judging from the features and demonstrations of the Big Network Controller, at least no new functionality or support has been implemented (Big Switch Networks, Inc., 2014). The main benefit of the existence of the mentioned product is the fact that Big Switch Networks can offer a part-

ner for enterprise actors and internet service providers, who would not dare to use an unsupported and free of charge solution in their production environments. For ISPs and other enterprises, in case something goes wrong, there always has to be some entity to share the blame and help fix the situation.

3.1.9 POX and NOX controllers

POX and NOX are two SDN controllers developed and released by open source community at NOXrepo.org. NOX is the SDN controller a company called Nicira was developing in the beginning of the SDN concept, and it goes as far as claiming to be the first SDN controller ever developed. Nicira (later bought by VMware) released NOX as open source after shifting their focus to other subjects, and the community at NOXrepo.org continued its development from that point.

NOX and POX are however mostly research-oriented SDN controllers. Therefore it is clear that they can't act as solutions in capitalized internet service provider or network operator areas. This reason alone puts them outside the scope of this thesis.

While Java-developed POX and its Python-oriented sister product NOX support OpenFlow and are very interesting from the SDN controller development perspective, there is another matter that makes them totally unusable in this thesis; they don't have any kind of support for northbound APIs. (NOXRepo.org, 2014)

3.1.10 IBM SDN VE Unified Controller

While IBM is obviously a major player in ICT research and development sector, in the networking area their presence is not very extensive, and because they cannot offer end-to-end solutions in the networking realm, they cannot be considered a direct competitor or a same class actor as the aforementioned Juniper, HP and Cisco. But because IBM has a big role in defining the direction of the whole IT industry, it is interesting to follow their developments too.

Like Big Switch Networks in the case of the Floodlight SDN controller, IBM has been a supporter and a co-developer of the OpenDaylight controller during the course of its development cycle. IBM was also participant in the SDN controller market with a product of their own, IBM Programmable Network Controller, from the early years of the SDN technology. Its development however slowed down during 2013 and 2014 (e.g. the controller's OpenFlow support remained on version 1.0 level (IBM Corporation, 2014a)) and IBM was even rumored to be exiting SDN business (Hesseldahl, 2014).

Just two weeks after information about IBM seeking to sell their SDN business surfaced, IBM released a new SDN controller and an ecosystem, this time a product based on the open source controller OpenDaylight, which IBM had been supporting for years.

The controller is titled IBM SDN VE Unified Controller, and it is a fully OpenFlow 1.3 compliant, and has also added (compared to the OpenDaylight) the support for controlling Linux-based KVM hypervisors and VMware environments (Little, 2014).

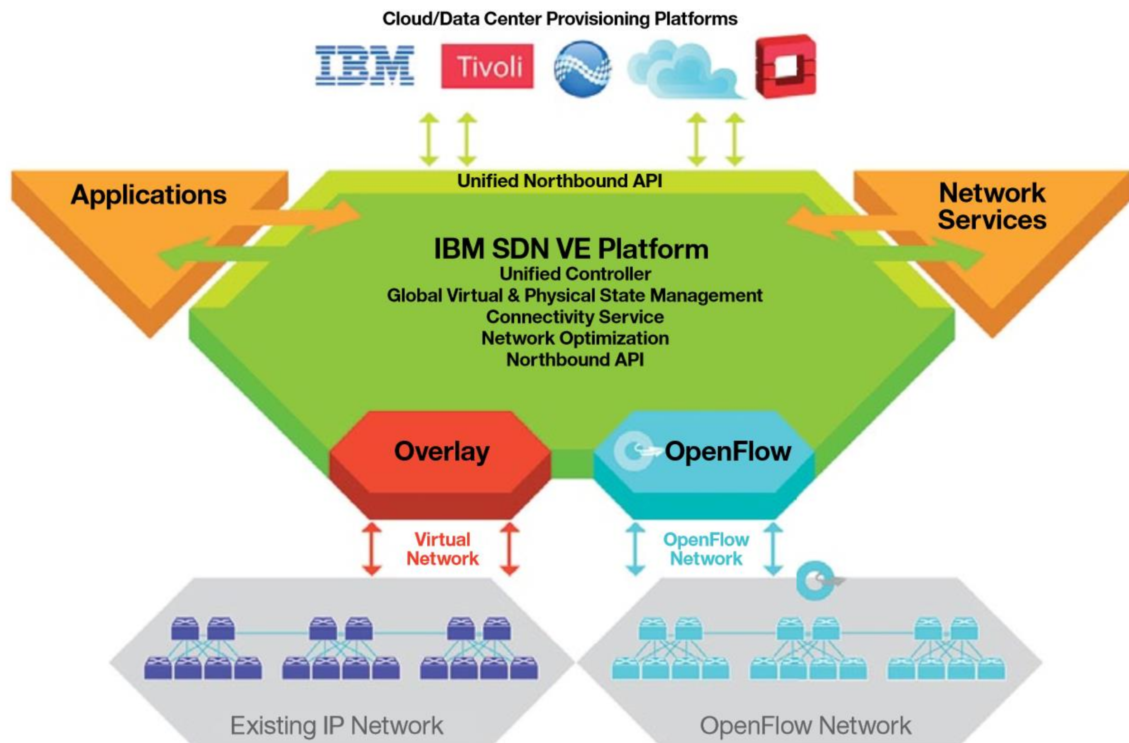

Figure 10. *IBM SDN VE Platform's architecture (IBM Corporation, 2014b).*

As described in the Figure 10, IBM’s solution offers a northbound API, much like the all the other controllers as well as the OpenDaylight controller the IBM’s solution is based on. According to the same source, the northbound API is RESTful and offers access to network functions, but not much else is publicly available about the IBM’s product. Because the controller is based on OpenDaylight codebase, it is a safe assumption that the products’ northbound APIs have much in common, however. The IBN SDN VE Platform could not be acquired for testing during the creation of this Master of Science’s thesis.

3.1.11 Other SDN controllers

It is important to understand that SDN controllers on market are not limited to the options listed and discussed here. For instance the Open Networking Foundation maintains a public list of SDN and/or OpenFlow -related products on market (as submitted by their members, which might cause some bias or possible exclusions) (Open Networking Foundation, 2014c), and there are several products listed there which have not been mentioned here.

Examples of such products are Plexxi Control, NEC’s ProgrammableFlow Controller, Nicira Network Virtualization Platform and NTT Virtual Network Controller. All of

these are interesting products, but all in one or more ways non-optimal or even unsuitable for the scope this thesis has. Plexxi Control, for instance, is a complex product involving two tiers of controller functionality; the more common controller is accompanied by a number of co-controller residing in, or among, the switch hardware. Like all the controllers, Plexxi Control has an API for third party -developed functionality, but the API is heavily emphasized towards VMware's vCloud and vCenter products. (Murray, 2012) This practically means the Plexxi Control has been designed only data centers in mind. While being built upon a non-standard representation of SDN architecture wouldn't be a problem in itself, being a non-generalized solution renders Plexxi Control uninteresting, at the very least.

NEC's product is interesting due to the fact that they are a significant actor in ICT research and development field, but their solution seems to be very closed, and more detailed information is available to paid customers only. Blind dive into NEC's technology was not within the realm of possibility during the making of this thesis.

From TeliaSonera's point of view definitely the most interesting of these excluded SDN controllers would be the NTT's Virtual Network Controller. This is due to the fact that in addition to being a SDN controller developer, NTT (Nippon Telegraph and Telephone Corporation) happens to be the most direct and largest international competitor of TeliaSonera in the Internet backbone market. According to Earl Zmijewski in Renesys' research article "A Baker's Dozen, 2013 Edition" published in January 2014, NTT was the second largest IP backbone operator in the world, while TeliaSonera was the third largest in December 2013. While Renesys doesn't release the exact figures (only relative graphs) to the public, it has to be noted that the difference between the two operators is such a small that in the end of the December it can hardly be seen in the graphs. It does seem that NTT just got ahead TeliaSonera in the end of the year, while TeliaSonera was the dominant one during the rest of the year. (Zmijewski, 2014)

A competitive situation like this practically excludes the NTT SDN controller device from the possible product choices by TeliaSonera, and thus renders it uninteresting from the angle the thesis is written. Nicira on the other hand was one of the first pioneers attempting to capitalize on SDN market, but they were acquisitioned by VMware in 2012, which resulted in some of Nicira technology being blended into existing VMware virtualization products (Butler, 2013), but no clear product launches or reveals have been made in the SDN area by VMware. Nicira Controller still exists as a product, but it has not been actively developed in two years.

Probably the most interesting product from TeliaSonera's perspective, not dissected in this thesis, is Alcatel-Lucent's SDN platform called Nuage. Nuage was not available during the writing of the thesis, but from all the vendors Alcatel-Lucent seems to have most complete vision of capitalizable use cases for SDN. Their SDN approach is, how-

ever, not very general, as it employs VXLAN overlays as part of the solution, and never dwells in the hardware layer itself. Alcatel-Lucent’s solution could easily lead to quickly generated income, if adopted fast.

3.2 Selecting two SDN controllers as development platforms

Next step in this thesis is to assess the SDN controller market and make a choice of two platforms for hands on development. The candidates for the selection were introduced in the chapters 3.1.2 to 3.1.11.

The selection was made based on the analytic hierarchy process –technique, which is a method for multi-attribute analysis and decision-making, that organizes complex systems or unstructured problems with many elements into comprehensible and a mathematical form (Zilinskas, et al., 2012). AHP (analytic hierarchy process) was developed by Thomas L. Saaty in 1970s and has been extensively analyzed and refined since then.

The analytic hierarchy process was chosen as the decision-making method over other multi-criteria decision analysis methods (MCDA) for three reasons. Firstly, James McCaffrey conducts in MSDN Magazine’s article “Test Run: The Analytic Hierarchy Process” in June 2005 that AHP is not only validated to be both technically valid and practically useful, but it according to the article, it is also well-suited for measuring overall software quality. And because SDN controllers are software products, AHP would suit very well for the needs of this thesis. (McCaffrey, 2005)

Secondly, another scientific study carried out by Ramatullah Khondoker et al., which includes a feature-based SDN controller selection, was released by Fraunhofer Institute for Secure Information Technology in 2014 (Khondoker, et al., 2014). This study relied on Analytic Hierarchy Process as well. Using the same method here makes it easier to compare the results obtained and grounds used for the decision-making process. Khondoker, et al.’s study differs greatly from this thesis in many aspects, however. The candidates used in the article are all open source projects, so paid vendors have not been taken into account at all. In addition to that, over half of the SDN controllers in the article are not traditional controller products, but either research-oriented projects (POX) or more like a frameworks for developing SDN controllers (Trema, Ryu). Trema and Ryu are not suitable for direct deployment in networks, but destined more for developing functionality on top of them. Lastly, the Fraunhofer Institute’s article doesn’t clearly form any kind of special angle in making the controller selection, while for the purposes of this thesis a great weight is put on the interfaces in both southbound, but especially in northbound directions.

Thirdly, the author of this thesis has former experience with AHP method. Therefore it’s an effective solution in a workflow-sense.

The intention on adapting AHP method for selecting the best possible SDN controllers is to apply extremely heavy importance on the northbound API scope, functionality, documentation and support. All this enables the developmental possibilities that will be used later in the thesis. From this thesis' point of view, based on the ultimate goals wished to be achieved, things such ease of deploying the SDN network underneath the controller, or the feature list of the southbound interfaces are not weighed as important. The question studied in this thesis revolves around building value adding services on top of the SDN stack, and more specifically developing SDN applications based on the northbound APIs.

3.2.1 Applying analytic hierarchy process on the problem

Based on the initial introductory research, for this thesis five SDN controllers were chosen for comparison with the AHP method. The final list of these controllers is (in alphabetical order) is Cisco APIC-EM, HP VAN SDN Controller, Juniper Contrail, OpenDaylight and Project Floodlight. Even though other SDN controllers were introduced in the earlier chapters, only five most promising ones were chosen for the AHP analysis. The reason for this is that despite the fact that AHP supports basically unlimited number of candidates for selection, according to McCaffrey in the MSDN Magazine article referred earlier, three or four candidates would be optimal number for evaluation purposes (McCaffrey, 2005), and therefore the amount of controllers in the AHP method is not desired to grow too high. Due to the high availability of SDN controllers in the market, the number of candidates was grown to one higher than the optimal three or four.

After choosing the candidates, the first step in the AHP process was to select the attributes in which the SDN controllers would be evaluated. Because the chosen platform will be used for software development, clear and descriptive documentation is very important. Therefore "documentation" was chosen as one of the comparison attributes. In addition to this, the functionalities of the controller dictate what is possible, what the limitations are, and how easy it is to achieve specific features or behavior with the controller, so "functionality" was chosen as the other top level attribute.

According to McCaffrey, as a general rule, well created AHP most often has any number necessary of top-level attributes, which already were chosen to be the "documentation" and "functionality", and between two to five sub-attributes. This given number was taken into consideration as the desired limitation in the number of sub-attributes.

In the documentation -area a sub-attribute called "installation and administration" was chosen to represent the general documentation for the SDN controller's administration, such as setting up the network for the controller, setting the controller to handle the

network, installing the controller, taking care of pre-requisites and using and maintaining the controller from the default user interface (either graphical or command line).

In addition to the generalized sub-attribute described above, extra interest was placed on API documentation in other sub-attributes. "Method descriptions" were chosen as another sub-attribute in the documentation realm. This attribute includes the clarity and depth of method per method –styled documentation of APIs available. "Code examples" were chosen as another sub-attribute. This attribute contains all the officially available code snippets and examples for API usage. Lastly, the availability of "machine-readable documentation" (e.g. Javadoc etc.) was chosen as a sub-attribute in the documentation section.

In the "functionality"-category the extent of device support in the southbound direction was chosen as one sub-attribute under the name "southbound support". Secondly, the extent and depth in which the device implements the full vision of software-defined networking was chosen as a sub-attribute. This takes a stance on whether each of the SDN controllers has watered down the SDN philosophy by making questionable technological choices.

"API scope and extent" was taken into consideration in third sub-attribute. In this attribute the size, but also quality, feasibility and perceived usability are taken into account into a single, but very important sub-attribute. Lastly, the number of possible APIs is taken into account in another sub-attribute. This practically means checking whether respective SDN controller support for instance REST, SOAP and/or Java APIs or only some of these. The problem setup is presented in the Table 1.

Table 1. *AHP setup in a tabular form.*

Problem	
Choose an SDN controller	
Alternatives	
Cisco APIC-EM, HP VAN SDN Controller, Juniper Contrail, OpenDaylight, Project Floodlight	
Attributes	
Documentation	Installation and administration
	API method descriptions
	Code examples
	Machine-readable documentation
Functionality	Southbound support
	SDN vision fulfilment
	API scope and extent
	Number of APIs

After the problem had been set-up within the Action Hierarchy Process domain, the next step was determining relative weights of each of the selected comparison attributes.

AHP uses a pair-wise comparison technique, which is firstly implemented for the top-tier attributes; documentation and functionality. The top-tier attributes are compared against each other in their relative importance; either both items are equally important, or another of the attributes has to be somewhat more important, definitely more important, much more important or extremely more important, like described in Table 2. This terminology is given by the AHP process, and these relative terms are given numeric representations in the following manner:

Table 2. *Action Hierarchy Process comparison values.*

Relative importance	Value
Equal importance/quality	1
Somewhat more important/better	3
Definitely more important/better	5
Much more important/better	7
Extremely more important/better	9

In this AHP implementation it was decided that functionality is somewhere between equal importance with documentation, and somewhat more important than documentation. Therefore a numerical value of 2 was chosen for this relationship. Rough interpretation of this value would be that functionality is twice as important within the scope of this Master's thesis as documentation. Placing the value 2 in AHP relative weight table, gives a pair-wise comparison described in the Table 3.

Table 3. *Pair-wise comparison of top-level attributes.*

	Documentation	Functionality
Documentation	1.000	0.500
Functionality	2.000	1.000

The next step in the AHP is to calculate the priority vector from the table. The process involves summing the columns, and dividing every entry in the table with its column sum. With two items this procedure stays very straightforward, but it increases in complexity when calculating the numeric values for sub-attributes. Calculated values for the top-tier attributes are described in the Table 4.

Table 4. *Calculating the priority vector for the attributes.*

	Documentation	Functionality
Documentation	0.333	0.333
Functionality	0.666	0.666

In the AHP priority vector calculation, the last step for specific tier calculations is taking the average from each row. As pointed out earlier, for a matrix of only two attributes, this phase is also very straightforward.

Determining the relative importance of sub-attributes is procedurally a copy of the work done for the first tier. The Table 5 illustrates the relative pair-wise importance of each of the sub-attributes. Pair-wise importance indicates, for instance that from this thesis' point of view, API methods descriptions are definitely more important than installation and administration, and also that code examples are between somewhat and definitely more important than machine-readable documentation. Relationships like these are expressed between every pair in both directions in this 4x4 matrix.

Table 5. *Pair-wise comparison of second-level attributes in Documentation category*

	Installation and administration	API method descriptions	Code examples	Machine-readable documentation
Installation and administration	1.000	0.200	0.333	1.000
API method descriptions	5.000	1.000	2.000	5.000
Code examples	3.000	0.500	1.000	5.000
Machine-readable documentation	1.000	0.200	0.200	1.000
Column sum	10.000	1.900	3.533	12.000

The data presented in the Table 5 was used to calculate priority vectors for each of the pairs by summing the columns, and dividing every entry in the table with its column sum. This procedure led to the temporary priority vector values presented in Table 6.

Table 6. *Interphase results for priority vector calculation in Documentation category.*

	Installation and administration	API method descriptions	Code examples	Machine-readable documentation
Installation and administration	0.100	0.105	0.094	0.083
API method descriptions	0.500	0.526	0.566	0.417
Code examples	0.300	0.263	0.283	0.417
Machine-readable documentation	0.100	0.105	0.056	0.083

The next step in AHP would then produce so-called eigenvalues for each of the attributes, and this happens by calculating the average of each of the rows. The eigenvalues or priority vectors for each of the four sub attributes in Documentation category are presented in the Table 7.

Table 7. *Eigenvalues for sub attributes in Documentation –category.*

Installation and administration	0.095
API method descriptions	0.502
Code examples	0.316
Machine-readable documentation	0.086

In this case a rounding error occurs, because the sum of the eigenvalues adds only up to 0.999 and not 1.000. This error is left into the system, because randomly adding an arbitrary 0.001 to one of the sub-attributes would just introduce more error.

The same process was applied on the Functionality -category's sub-attributes, and the Table 8 presents the results from the process. The results include a similar rounding error of 0.999 versus 1.000 that occurred in the other category as well, which is only a coincidence.

Table 8. *Eigenvalues for sub attributes in Functionality –category.*

SB support	0.109
SDN vision fulfilment	0.209
API scope and extent	0.572
Number of APIs	0.109

3.2.2 Comparing the SDN controllers on each attribute

After the problem had been set-up for analytic hierarchy process, the next step in the process was performing a comparison of each SDN controllers based on each of the lowest-level attributes. This comparison was to be done eight times, once per each of the sub-attribute, using the five aforementioned SDN controllers as candidates in the selection process. The process itself is identical to what was performed earlier for each of the sub-attributes earlier when determining their importance. As an example, differences of API scope and extent (the single most important attribute in the whole process) started by making a pair-wise comparison with the numerical values presented in Table 9. In the table it can be seen that HP VAN SDN Controller is deemed between definitely and somewhat better in API scope and extent than Cisco APIC, for instance. On the other hand, Cisco APIC has (with the numerical value of 2) a little less than "somewhat better" (which would require numerical 3) API scope and extent than Project Floodlight.

Table 9. Pair-wise comparison of API scope and extent –attribute.

API scope and extent	Cisco APIC	HP VAN SDN Controller	Juniper Contrail	OpenDaylight	Project Floodlight
Cisco APIC	1.000	0.250	0.500	0.250	2.000
HP VAN SDN Controller	4.000	1.000	3.000	1.000	6.000
Juniper Contrail	2.000	0.333	1.000	0.333	2.000
OpenDaylight	4.000	1.000	3.000	1.000	6.000
Project Floodlight	0.500	0.167	0.500	0.167	1.000
Column sum	11.500	2.750	8.000	2.750	17.000

The AHP then led, with an identical process compared to pair-wise importance calculations, to the results of comparative quality of API scope and extent between all the controllers expressed in single numerical values. These results are presented in the Table 10.

Table 10. Quality vectors for each of the assessed SDN controllers in one sub-attribute.

API scope and extent (.572)	
Cisco APIC	0.090
HP VAN SDN Controller	0.361
Juniper Contrail	0.131
OpenDaylight	0.361
Project Floodlight	0.057

The process described above was then repeated seven times, once for every other sub-attribute. It could be calculated that the “API method descriptions” –sub-attribute under Documentation –category was the second most important attribute in the whole AHP system, which was interesting, because this was despite the fact that Functionality –category was after all ranked to be twice as important as the Documentation –category.

Every SDN controller platform provided an API documentation that was sufficient for using each of the methods. Writing styles varied, as well as the formats they were provided in (such as MediaWiki-based documentation versus a PDF-document). Also because code examples and machine-readable documentation were separated from API method descriptions –attribute, a very conservative approach to differences between platforms was established. The full pair-wise comparison for initial values can be seen in the Table 11.

Table 11. Pair-wise comparison of API method descriptions –attribute.

API method descriptions	**Cisco APIC**	**HP VAN SDN Controller**	**Juniper Contrail**	**OpenDaylight**	**Project Floodlight**
Cisco APIC	1.000	0.500	1.000	1.000	0.500
HP VAN SDN Controller	2.000	1.000	2.000	1.000	2.000
Juniper Contrail	1.000	0.500	1.000	0.500	0.500
OpenDaylight	1.000	1.000	2.000	1.000	2.000
Project Floodlight	2.000	0.500	2.000	0.500	1.000
Column sum	7.000	4.500	8.000	4.000	6.000

The table shows that according to the initial values inserted into the AHP, for instance HP VAN SDN Controller's API method descriptions are a little better than Cisco's, Juniper's and Project Floodlights, while OpenDaylight is assumed to be roughly on the same level. OpenDaylight on the other hand has a little better API method documentation than Juniper Contrail and Project Floodlight, but Cisco's offering is estimated to be on the same level. The core idea of the pair-wise comparison system seems to be a built-in consistency checking between inputted data as well as built-in system for finding out minimal perceived differences between systems via multiple pair-wise comparisons.

The pair-wise comparison of API method descriptions, as well as six of the other sub-attributes, led to comparative quality values presented in the Table 12. The table indicates also the relative importance of both Documentation- and Functionality –categories, as well as all the sub-attributes within them as fractional values.

Table 12. *Quality vectors for each of the assessed SDN controllers in seven sub-attributes.*

Documentation (.333)	
Installation and administration (.095)	
Cisco APIC	0.199
HP VAN SDN Controller	0.191
Juniper Contrail	0.204
OpenDaylight	0.258
Project Floodlight	0.147
API method descriptions (.502)	
Cisco APIC	0.144
HP VAN SDN Controller	0.288
Juniper Contrail	0.125
OpenDaylight	0.250
Project Floodlight	0.193
Code examples (.316)	
Cisco APIC	0.128
HP VAN SDN Controller	0.298
Juniper Contrail	0.118
OpenDaylight	0.290
Project Floodlight	0.166
Machine-readable documentation (.086)	
Cisco APIC	0.171
HP VAN SDN Controller	0.296
Juniper Contrail	0.142
OpenDaylight	0.225
Project Floodlight	0.165
Functionality (.667)	
Southbound support (.109)	
Cisco APIC	0.110
HP VAN SDN Controller	0.248
Juniper Contrail	0.146
OpenDaylight	0.248
Project Floodlight	0.248
SDN vision fulfilment (.209)	
Cisco APIC	0.094
HP VAN SDN Controller	0.243
Juniper Contrail	0.140
OpenDaylight	0.280
Project Floodlight	0.243
Number of APIs (.109)	
Cisco APIC	0.149
HP VAN SDN Controller	0.259
Juniper Contrail	0.170
OpenDaylight	0.225
Project Floodlight	0.196

From data presented in the Table 12, it was possible to calculate aggregate weights for all the SDN Controllers. This was done by calculating sum of the quality values multiplied by their relative importance, which in turn were product of the importance of the category and importance of the sub-attribute. For example Juniper Contrail's final aggregate weight was calculated as:

$$0.333 * 0.095 * 0.204 + 0.333 * 0.502 * 0.125 + 0.333 * 0.316 * 0.118 + 0.333 * 0.086 * 0.142 + 0.667 * 0.572 * 0.131 + 0.667 * 0.109 * 0.146 + 0.667 * 0.209 * 0.140 + 0.667 * 0.109 * 0.170 = 0.136$$

Similar calculation was executed for every candidate in the AHP system. Finally, an error-checking procedure was carried out by calculating the sum of all the final quality values, which sum to 1.00. The final quality values are presented in the Table 13.

Table 13. *Final quality values of every SDN controller according to the AHP analysis.*

SDN controller	Final quality value
Cisco APIC	0.115
HP VAN SDN Controller	0.302
Juniper Contrail	0.136
OpenDaylight	0.298
Project Floodlight	0.153

Based on the results obtained from the Analytic Hierarchy Process –method, a choice of two SDN controllers were made. In the results, HP VAN SDN Controller and OpenDaylight differentiated themselves from the rest of the group based on the criteria that was weighted for the needs of this thesis. Both systems were available for hands on –testing as well, HP's controller through official channels in HP's sales unit and OpenDaylight directly from the website of the project. Because HP VAN SDN Controller won the comparison and it is a commercial product and not an open source project, making it more suitable for eventual deployment in production environments for TeliaSonera, it was chosen as the primary platform. In practice this means that the solution will be developed on HP's platform first, and then converted for the secondary selection, OpenDaylight. This conversion process is intended to answer to the question what happens if a SDN controller is changed from one product to some another. The thesis project aims to encounter and deal with possible compatibility issues and at least will make sure, if conversion work is possible or not.

3.2.3 Controller descriptions

In this chapter a closer look to the internal designs of the chosen controllers is taken. The three–tiered generalized SDN architecture, or the SDN stack, was introduced in earlier chapters. In this chapter the point is to look at how the components within specif-

ic controllers correspond to each of the layers, what kind of messages is sent between these components and what kind of possible APIs or other interfaces they expose to outer world. Also some basic things such as what the controllers are based on, how they are installed - as virtual machines, Linux software, or are they e.g. hardware controllers instead of software ones, are clarified. In case of software controllers, the software stack is described as closely as possible.

3.2.4 HP VAN SDN Controller internals

The HP VAN SDN Controller is a Java-based SDN controller. It can be installed on an Ubuntu 12.04 LTS Linux, which is the only officially supported Linux distribution for the SDN controller, and it is offered as a Debian package.

From software-perspective, on top of the Ubuntu Linux, the virtualized HP controller software is built upon Java 1.7 and OSGi (Virgo stack and Equinox framework) for software module interaction. In addition to these, key modules installed on the Linux distribution are Keystone for authentication management, PostgreSQL for databases and logging and Zookeeper for inter-module synchronization purposes, as well as RabbitMQ for inter-module messaging. Aforementioned software components are not HP-specific but generally available software, on top of which the HP stack resides. The HP's controller base consists of modules dealing with teaming, with device drivers, OpenFlow controller and so forth. These are internal to the controller and therefore cannot precisely be separated or listed from an installed copy of the SDN controller platform, but existence and meaning of these can be deducted from generally available architectural description images.

The logical perspective of the controller architecture can be described as presented in the figure 11. The controller part of the architecture contains a set of southbound components, which deal with the network devices. These components, or modules, include OpenFlow module interacting with OpenFlow-enabled devices, but also REST interfaces for DNS/DHCP and for some limited Layer 2 and Layer 3 functionality.

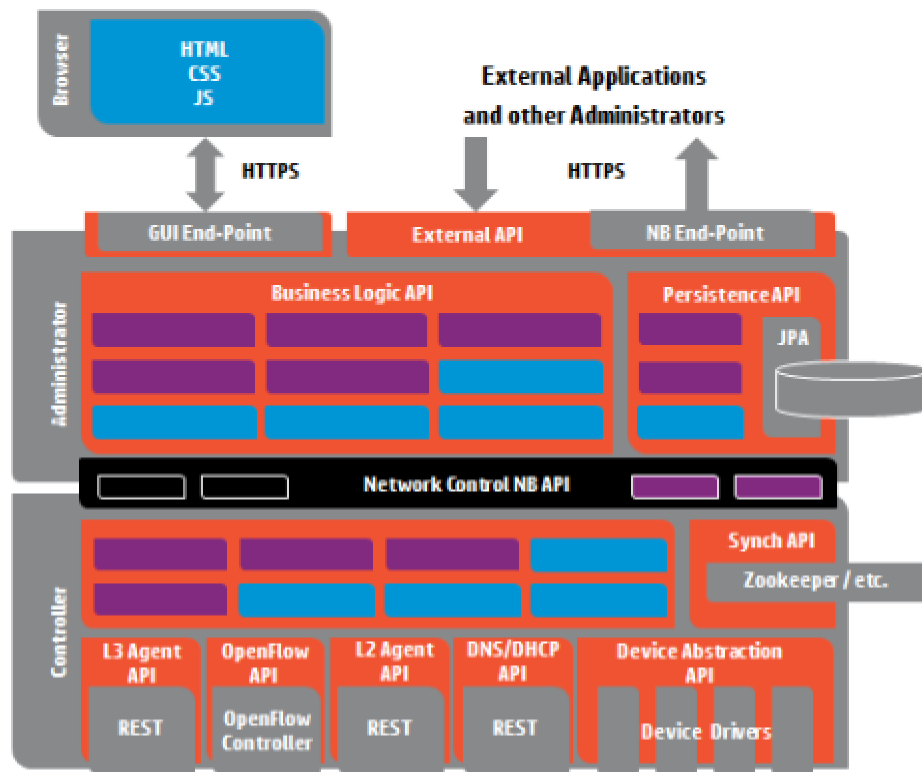

Figure 11. *HP SDN VAN Controller architecture (Hewlett-Packard Development Company, L.P., 2013c).*

The administrator-tier of the diagram contains northbound components which deal with other network management frameworks, tools or applications, such as OpenStack or self-created applications like the one developed later in this thesis. Some of these components, such as the GUI End-Point, are meant to interact directly with the user via HTTPS calls from the user's web browser.

The Controller and Administrator tiers communicate with each other over well-defined mutual interfaces included in the Network Control NB API segment of the architecture, and the architecture is presented with increasing coarseness from top-to-bottom. Basically this means that e.g. when some very high level instructions arrive to the controller from an external application via HTTPS to the External API, the high level instruction is broken down by the upper tier into a specific plan for achieving the instructions in SDN terms, which then gets communicated via the inter-tier API to the lower controller tier. The controller tier then turns the plan into detailed instructions and forwards them to the network usually via the OpenFlow API, but other possible APIs are available as well.

3.2.5 OpenDaylight internals

Much like HP's SDN VAN Controller, OpenDaylight controller runs in a Java Virtual Machine within an operating system installation. OpenDaylight claims to be more flexible with the operating system used for running the controller, and goes as far as to claim

that it can be run also on other operating systems than Linux. For best results OpenDaylight documentation recommends usage of “a recent Linux distribution”, however. Java Virtual Machine is expected to be in the version 1.7, like was the case with HP’s controller as well.

OpenDaylight is installed to a system via freely-downloadable build package that contains both Linux and Windows executables for running the controller. The controller installation requires minimal initial configuration, and could even be instantly runnable after unpacking the downloaded build.

The OpenDaylight controller’s architectural figure was already presented in the chapter 3.1.7. The architectural figure clearly resembles the HP’s logical architecture with few exceptions. OpenDaylight controller offers significantly larger support for southbound protocols which are not supported by almost-pure OpenFlow-design of the HP’s platform. OpenDaylight’s southbound interfaces and protocol plugins include support for protocols such as SNMP, BGP and NETCONF for interaction with the underlying network in addition the standard set of OpenFlow (of which currently two versions are supported, 1.0 and 1.3).

The OpenDaylight’s architecture is presented in tiers much like in HP’s case, but the separation of these tiers differs slightly. A large tier called Controller Platform includes everything else within the controller except for southbound and northbound APIs. OpenDaylight’s northbound APIs are offered in REST architectural style and they can serve a set of SDN applications and orchestration software just like the HPs platform.

3.2.6 Northbound APIs in SDN controllers

Because the lack of any kind of northbound API standardization, the SDN controllers’ northbound APIs are not locked in to being implemented as REST APIs. Other API designs such as SOAP or RPC could have been used, but both controllers here have selected the REST architecture as their primary design for the Northbound APIs. The similarity between controllers is expected to ease the development process and because only single set of web APIs are offered by both of the controllers, decision making on API selection is not required.

3.2.7 HP VAN SDN Controller’s Northbound API

Because platforms chosen, HP VAN SDN Controller and OpenDaylight controller implement REST APIs, and the purpose of these APIs is very similar, HP’s API will be dissected here on much deeper level of detail than OpenDaylight’s. The mentioned interfaces are compared almost on per method –basis when analyzing the interchangeability of SDN controllers with self-developed SDN applications running on top of them.

HP VAN SDN Controller's REST API is very extensive and a well-formed REST API with API methods divided into three separate namespaces. The Core namespace ("`/sdn/v2.0`") includes the core API, dealing with e.g. SDN controller's alarm, logs and backups. The OpenFlow-namespace ("`/sdn/v2.0/of`") contains methods for accessing and manipulating network data flows with OpenFlow functionality. The same REST API can be used on both OpenFlow 1.0 and OpenFlow 1.3 devices. However, certain API calls (such as meters) are only available when speaking to an OpenFlow 1.3 devices. In addition to the standard OpenFlow-functionality the Network namespace ("`/sdn/v2.0/net`") contains a set of proprietary functions such as ability to read and manipulate topologies, routes and make packet inspections not available in the OpenFlow 1.3 standard. The controller offers JSON schemas for each of the namespaces at `/sdn/v2.0/models`. The API consists totally of around 70 methods.

List controllers

Sample request

List all controllers:

```
GET /sdn/v2.0/systems
```

List controller by the given IP:

```
GET /sdn/v2.0/systems?ip="192.168.1.100"
```

There is no request body for this API.

Sample response


```
 1 {
 2   "systems": [
 3     {
 4       "uid": "adc5e492-957c-4f8c-aa0a-97fa2dac5f23",
 5       "version": "01.11.00.0000",
 6       "role": "leader",
 7       "core_data_version": 0,
 8       "core_data_version_timestamp": "1970-01-01T00:00:00.000Z",
 9       "time": "1970-01-01T00:00:00.000Z",
10       "self": true,
11       "status": "active"
12     }
13   ]
14 }
```


Response codes

- Normal: OK (200)
- Error: Unauthorized (401), Not Found (404), Service Unavailable (503)

Figure 12. *An example of HP's API documentation (Hewlett-Packard Development Company, L.P., 2014b).*

The whole API is documented in openly available API documentation, of which an example can be seen in Figure 12. For a web developer or a programmer the documentation is very descriptive and easily understood. In the example documentation a method residing behind the URI `/sdn/v2.0/systems` is described. The method is accessed via HTTP's GET-method, and it is stated to list either all the controllers, or if combined with a parameter including a controller's IP address, list the controller residing behind that address. Listing would include information on the controller as described in the sample response, e.g. controller's software version, role and timestamp.

If requests are sent with accompanying request bodies, an example of these bodies are given in the same manner like the example response body was shown, highlighting if not all, the most commonly used possibilities for request bodies. If more information is needed, JSON Schemas would answer to the rest of the questions. The documentation also states how HTTP status codes are used, so error handling for them could be programmed correctly in the SDN application.

/diag Show/Hide | List Operations | Expand Operations | Raw

Method	Path	Description
GET	/diag/packets/{packet_uid}/nexthops	Returns the next hop neighbors
GET	/diag/packets/{packet_uid}/path	Returns the list of ordered path links
POST	/diag/packets/{packet_uid}/action	Performs an action to the packet
GET	/diag/packets	Gets all packets
POST	/diag/packets	Creates a packet
GET	/diag/observations	Returns the observation posts
POST	/diag/observations	Sets the observation post
DELETE	/diag/observations	Removes the observation post
GET	/diag/packets/{packet_uid}	Gets the packet
DELETE	/diag/packets/{packet_uid}	Removes a packet

/lldp Show/Hide | List Operations | Expand Operations | Raw

/logs Show/Hide | List Operations | Expand Operations | Raw

/forward_path Show/Hide | List Operations | Expand Operations | Raw

/datapaths Show/Hide | List Operations | Expand Operations | Raw

Method	Path	Description
GET	/of/datapaths/{dpid}/ports/{port_id}	Get info on a given port
POST	/of/datapaths/{dpid}/ports/{port_id}/action	Enable or disable port on a given datapath
GET	/of/datapaths/{dpid}/groups/{group_id}	get group
PUT	/of/datapaths/{dpid}/groups/{group_id}	Update a given group
DELETE	/of/datapaths/{dpid}/groups/{group_id}	Delete a given group

Figure 13. *A view into HP VAN SDN Controller's Rsdoc.*

In addition to the PDF-documentation freely available, the HP's VAN SDN Controller installation comes with a system called Rsdoc, which is an HP's own method for describing the REST API in dynamic form, named after two existing mechanisms, Javadoc and JAX-RS. Rsdoc is very useful tool for testing single REST calls and browsing through the API's methods with simple descriptions on each of the methods. Unlike Javadoc, Rsdoc does not easily integrate into currently available IDEs, but support for Rsdoc could be coming, if HP manages to make Rsdoc widely adopted mechanism or even a standard.

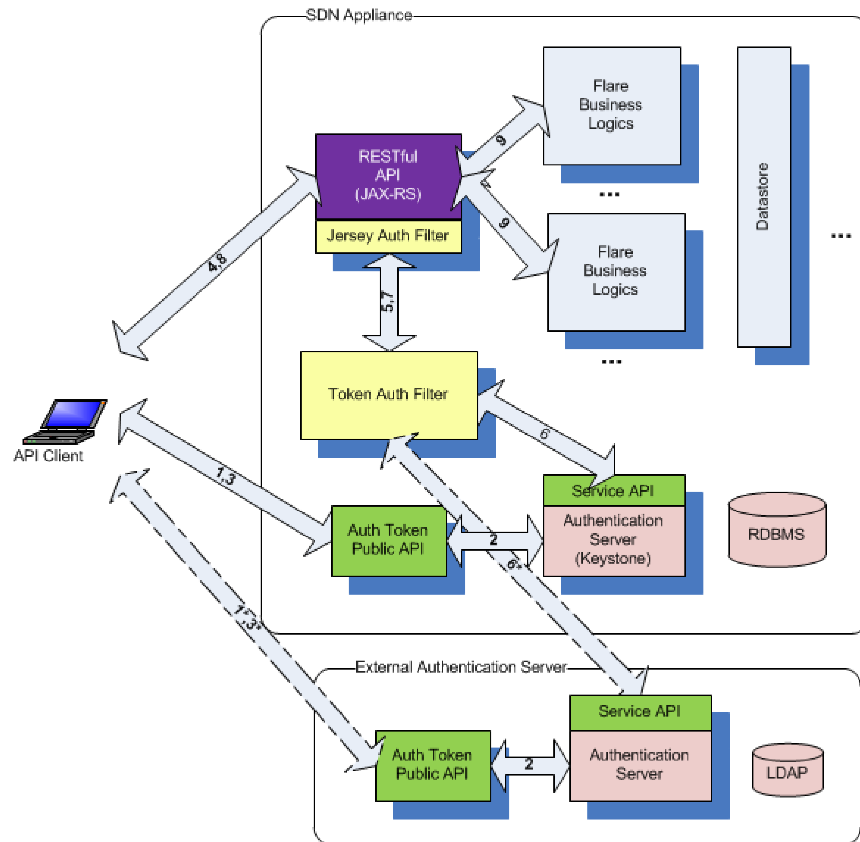

Figure 14. *Diagram of authentication process on REST API (Hewlett-Packard Development Company, L.P., 2013d).*

In the HP SDN VAN Controller's REST API, there are only two REST resources which can be accessed without authentication, found at `/` (for API discovery) and `/sdn/v2.0/auth` (user for authentication). Rest of the methods are usable only after OpenStack Keystone authentication, which creates authentication tokens valid for 24 hours after creation.

The authentication mechanism is described in the Figure 14. The API client, or SDN application first contacts the public API asking for authentication token, accompanied with authentication information. Public API confirms the privileges from the internal Keystone-based authentication server, after which the public API returns the authentication token. After acquiring the token, SDN application will interact with rest of the API methods getting the access to them with the said token. Before any business logics behind API calls are activated, authentication filter attached to all API methods verifies the authentication token at the Keystone server.

SDN applications developed with the REST API of the SDN controller are characterized in a few ways. As stated earlier, they can be written in any programming language of choice (e.g. Java, C, C++, Python, C#, PHP) and they are deployed outside of the SDN controller instance. Even if deployment of the said application is possible to be done on the same Linux platform the controller resides on, the application will still be

external application from SDN controller's perspective. Because these applications are external and do not modify the SDN controller platform in any way, they cannot for instance extend the REST APIs of the controller and due to external placement, they might have some latency issues depending on locations and network functionality. To solve these two problems, allowing the extension the REST API and giving ways to create extremely latency-critical SDN applications, in addition to the REST API HP has also exposed a "native" Java API for their SDN controller platform.

Native Java API means that some of the Java classes are available for any byte-compatible program code, such as Java, Scala or Scala SDL. HP's platform has implemented a OSGi-based modularization of their platform, meaning that from within the standard GUI of the SDN controller, it is possible to deploy self-written modules as OSGi-bundles as additions to the standard controller functionality. Java API offers virtually identical functionality with the REST API, and is divided in Control API, Model API, Communications API and Data Access API.

Modules written in this manner have the capability of extending both the Java API itself as well as adding more REST interfaces. REST API extensions would happen by extending REST API namespaces by adding `/ext/<vendor-id>`, e.g. `/sdn/v2.0/net/ext/teliasonera/business`.

HP points out in their documentation that internal Java API –based applications could serve best in tasks which require low latency and continuous action, such as handling packet-in events. In other words, Java API makes it possible to go to a lower abstraction level and closer to the network, while REST API offers more abstractions for business level, where latency is not as critical factor as it would be in packet-switching, etc.

3.2.8 OpenDaylight Northbound API

As stated earlier, OpenDaylight's REST API effectively achieves the same functionality than HP's API. OpenDaylight's API documentation is only available as MediaWiki-based interactive and community-driven documentation. Instead of a mechanism like Rsdoc, WADL is used to describe the REST API in addition to the MediaWiki-documentation.

As a general notion, OpenDaylight's API is much more modularized than HP's. The API has been divided into twelve modules, including a module for OpenFlow-based manipulation of the network with the Flow Programmer –module, topology discovery via Topology –module etc. As a downside, the WADL describing the API is only available as per module, and not a single WADL instance covering the whole API is available.

As a helpful feature, client-side libraries for C, .NET, Java and Objective C are available directly from the OpenDaylight's API documentation for the data model the REST resources expose. In this thesis these libraries did not offer any help, however, because the implementation language was chosen to be Python.

3.3 Development on HP VAN SDN Controller

3.3.1 SDK features

HP has composed a software development kit for developing third party applications and/or functionality with HP SDN VAN Controller. The SDK is freely available (Hewlett-Packard Development Company, L.P., 2014c) for download from HP's SDN Dev Center -website without any necessary credentials.

The SDK is a compressed file that consists of three PDF-documents titled HP VAN SDN Controller Administrator Guide, SDN Controller Programming Guide and HP VAN SDN Controller REST API. The contents of the SDK are assessed next.

3.3.2 HP VAN SDN Controller Administrator Guide -document

The HP VAN SDN Controller Administrator Guide is a 102-page document intended for network administrator and other personnel who install, set up, configure and administrate the HP's SDN controller. From this thesis' perspective it contains lot of useful information, because significant part of the work is setting up the development environment for the first time – including the SDN controller environment. The discussed document also points out how to deploy OSGi-packaged internal, third-party developed Java-applications on the SDN controller environment, itemizes Java application states, helps to view OSGi artifacts and points out how to manage these applications (tasks such as starting, stopping, replacing, uninstalling applications).

This information is useful and necessary when someone is developing internal Java-applications with the HP SDN controller. However, as it has been stated in this thesis earlier, the goal of this project is not to develop internal Java-application, but an external application running on third party server, relying on HTTP queries and responses through a REST API. The document also walks through all the views of the controller UI, such as logs, configurations and OpenFlow monitor-, topology- and trace-views, and gives pointers in making system backups and restores.

From this thesis' point of view, the most interesting parts of the first document of the SDK are the chapters which discuss REST Authentication and SDN Administrative REST API. The REST Authentication –chapter explains the used X-Auth-Token mechanism and describes how the SDN controller uses OpenStack Keystone as an identity management purposes. The document goes as far as giving CLI-commands for example

authentication with curl as the command line tool. The document also explains how the acquired X-Auth-Tokens are valid for 24 hour period since token creation.

The chapter called “SDN Administrative REST API” points out how the HP’s SDN controller has a REST API which includes a subsection for administrative tasks, which could be handled through the mentioned API from a third party application, such as network operations center personnel’s network management tool. The document walks through the REST API’s administrative capabilities, pointing out that the REST API is capable of stopping/starting/restarting the SDN Controller daemon (sdnc), downloading a ZIP bundle of log files, uploading upgrade ZIP bundles and executing upgrade commands and rebooting the system, among other things.

In the appendix-part of the document examples for all the above-mentioned administrative tasks via REST API are provided as CLI-style curl-commands. Also some longer Perl-scripts are available as examples, which aim to setting up and configuring a controller team for high-availability deployments, and making a backup and restoring this setup.

3.3.3 SDN Controller Programming Guide -document

The SDN Controller Programming Guide attempts to provide a detailed documentation for writing applications to run on the HP VAN SDN Controller platform. Document is a 225-page manual which contains lot of information, but most of it is aimed for internal Java-application development, rendering the document mostly informative, but not very useful in achieving the goals that have been set up for this thesis. There are useful parts for REST API development work too, however.

In its introductory part the Programming Guide discusses the differences of internal applications (native applications written in Java and packaged with OSGi and run on the SDN controller device) and external application (REST-based applications located in third party servers) in length and describes the basic architecture of the SDN controller, web application architecture and even goes as far as describing the importance of MVC-architecture in web development. All information given here is adaptable for REST API development work that will happen during the creation of this thesis.

The document begins its chapter on application development on generalized description and recap of authentication architecture on HP SDN VAN Controller platform, and advances to presenting the REST API, its benefits, weaknesses and features, and also walking the reader through Rsdoc, which is a semi-automated interactive RESTful API documentation (having much in common with Javadoc –documentations which tend to integrate themselves to programming IDEs) that is provided alongside the controller installations.

After these chapters the document dives very quickly to the Java development on code-level, which mostly produces documentation that is interesting but not very useful when implementing web services with another language based on the APIs offered by the controller device.

3.3.4 HP VAN SDN Controller REST API –document

Clearly the most useful piece of the existing SDK is the last document named HP VAN SDN Controller REST API. The document does not contain lengthy introductions or architectural descriptions, but rather acts as a comprehensive textual documentation of the whole REST API available on the HP SDN controller. It directly provides commands, sample requests, sample responses and response codes for interacting with the API.

In the compact introduction –part the document explains how the REST API has been divided into three distinct namespaces: core (`/sdn/v2.0`), openflow (`/sdn/v2.0/of`) and network services (`/sdn/v2.0/net`) and explains ways to access each of the namespaces' JSON schema descriptions for their JSON data formats. The document also points out the ideological differences of these three namespaces.

After this the document advances into listing, and later on describing each of the REST API methods available in the whole API. The whole API description is packaged into a 78-page document. Every method description begins by declaring a resource URI, such as "`datapaths/{dpid}/flows`" and then continues by describing what the API does when the resource behind this URI is manipulated with each of the HTTP methods (GET, PUT, POST, DELETE). With the mentioned example of "`datapaths/{dpid}/flows`", GET lists all flows, POST creates a new flow, PUT updates existing flow and DELETE removes a flow from the controller. From each of these examples the document provides a sample request, including an example of the JSON message body to go with the request, and a sample response in JSON –data format, as well as clarifies the usage of HTTP response codes method by method, such as the usage for 201 for OK, 400 for Bad Request, 401 for Unauthorized, 404 for Not Found and 503 for Service Unavailable.

The document is almost entirely without textual descriptions or guidelines to the issue at hand (web development against the API of HP SDN controller), but for an experienced developer it acts as a great API documentation, easily comparable to public and open source descriptions of large scale web API providers such as Google or Yahoo!. From a personal point of view, the document is one of the best API documents the author of this thesis has ever had to work with, and this is a big achievement considering the documentation is an every now and then updated, static PDF file, compared to a common style of Wiki (or other comparable) –based documents which are freely for anyone to keep up to date.

A general assessment of the SDK is that it is not very extensive, because it doesn't contain any kind of tools or larger chunks of example code, but consists only of documentation. Documents themselves are very descriptive and useful, but often "devkits" or SDKs contain for example libraries, example applications, debugging facilities, IDE integrations, test suites and other useable tools for developing and testing the applications created.

3.3.5 Development environment specification

The working phase from initial hands-on testing to completed development lasted several months and because technology was new, not easily available and only supported by a small portion of active network devices, some changes to the development environment occurred during the work done.

3.3.6 Initial physical development environment

The initial development environment consisted of HP's VAN SDN Controller deployed as a virtual appliance on an HP Compaq DC7900 SFF hardware. On the hardware, Ubuntu LTS 12.04 64-bit Server was installed as a platform for the HP SDN VAN Controller. Before installing the SDN controller, a few dependencies and pre-requisites were installed. These dependencies included OpenJDK 7 JRE, PostgreSQL 9.1 and Keystone 2012.2. After dependencies were installed, HP VAN SDN Controller was installed as a Debian package with dpkg. The version of the SDN controller was 2.0.0.4253. Because this development environment was built into a physical laboratory environment at TeliaSonera's network laboratory in Tampere, due to the best practices administered in the lab, each of the machines and devices involved in the environment were assigned a hostname. The SDN controller's hostname was "juice" and it was located in the domain "mit.sonera.com".

Initially only two OpenFlow (and thus SDN) –capable network devices could be acquired and reserved for the development work happening within the Master of Science thesis' framework. These two switches were identical and their model names were HP 2920-24G. The switches were upgraded to run software version WB.15.14.0002, because it added the support for OpenFlow 1.3 (instead of just OpenFlow 1.0). The switches were placed in the same network domain, and their hostnames were "tig" and "happy".

The switches were configured to be under influence of the SDN controller with the configuration presented in the Program 1. In the configuration shown, the controller IP pointed to the IP address of the HP SDN controller in the specified virtual LAN, and `version` -row defined the used OpenFlow version.

```
openflow
      controller-id  1  ip  194.142.21.118  controller-interface
vlan 11
      instance aggregate
            controller-id 1
            version 1.3
            connection-interruption-mode fail-standalone
            enable
            exit
      enable
      exit
```

Program 1. *OpenFlow configuration on HP switches.*

At this stage the development work was focused on getting the SDN-controlled network running, getting the SDN controller to have visibility for the network topology and traffic, while two end-system laptops were reaching each other with ICMP–protocol's ping-feature. After the basic setup had been done, first initial tests for queries against the SDN controller's REST API were done from the localhost machine. In other words, at this stage a separate web development environment didn't exist, but initial testing, proof-of-concepts and raw prototyping was done locally within the SDN controller's Ubuntu using CLI curl and later with Python 2.7.3.

Until very late in the thesis timeline, the initial development environment remained in the state described. The physical environment was later enhanced with a Juniper EX3200 switch, granting the possibility to test cross-vendor functionality with OpenFlow 1.0 and 1.3, giving important insight on real life deployments of HP VAN SDN Controller, but this was not until the thesis was in its finalizing phase.

3.3.7 Move to virtualized environment

When the development work speeded up, a need for more agile development methods and especially more complex underlying network environment clearly arose as an issue. Two switches, or later (but after the move to the virtualized environment) three switches could not form a network that could demonstrate or represent use cases that are interesting enough for application development in the SDN realm.

Therefore a virtualized development environment was created. This environment offered more control over its specification as well as made it remotely available anywhere, not just while working at the TeliaSonera premises.

VMware Workstation 10.0.2 was installed as the virtualization platform. On this platform two virtual machines were created: the server that offers SDN networking capabilities and the server, which contained the web development environment.

The SDN networking server was built on top of Ubuntu 12.04 LTS 64-bit, because it is the only officially supported Linux distribution by the HP VAN SDN Controller. The installation of SDN controller proceeded like it did in the physical environment.

Next step was to create a virtualized network, as opposed to physical switches used earlier, for the SDN controller to manage. This was achieved through the combination of two software components: Open vSwitch and Mininet. Open vSwitch is Apache 2.0 – licensed multilayer virtual switch, claiming production quality. Mininet on the other hand is a network simulator, which uses Open vSwitch to create a realistic virtual network on a single machine. Used Open vSwitch version was 2.0.0 and Mininet version was 2.1.0.

Mininet was instructed to use Open vSwitch as the virtualized switch within its virtualized network, while the network topology of these virtualized switches (and added virtual hosts) was defined with a short Python script, by creating a topology class, with self-initialization with presented switches and hosts in addition to key-value paired connection (links) between these nodes. The topology was introduced with code described in the Program 2.

```
# Add hosts and switches
h1 = self.addHost( 'h1' )
h2 = self.addHost( 'h2' )
s1 = self.addSwitch( 's1' )
s2 = self.addSwitch( 's2' )
s3 = self.addSwitch( 's3' )
s4 = self.addSwitch( 's4' )
s5 = self.addSwitch( 's5' )

# Add links
self.addLink( h1, s1 )
self.addLink( s1, s2 )
self.addLink( s2, s3 )
self.addLink( s3, h2 )
self.addLink( s1, s4 )
self.addLink( s4, s5 )
self.addLink( s5, s3 )
```

Program 2. *Topology creation in Python.*

This combination allowed the HP VAN SDN Controller to control a virtualized network running on the same machine, without basically even realizing the network it was commanding was a virtualized and not a physical, real-life environment. In the SDN controller, the network topology was seen like in the Figure 15 below.

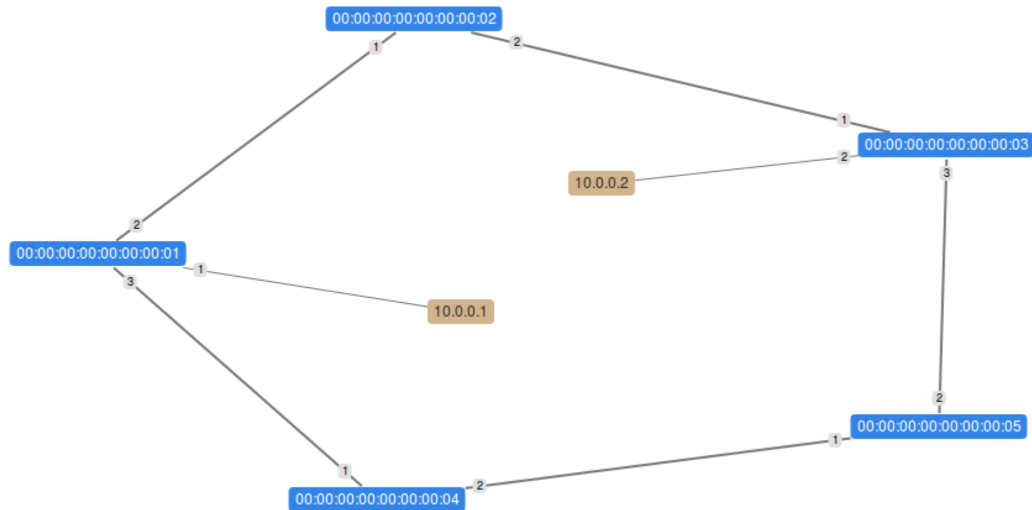

Figure 15. *Created topology viewed from within the SDN controller.*

On the web development server, Ubuntu 14.04 LTS was used because the web development environment was wanted to represent the newest available technologies. On this Ubuntu 14.04, the default installation of Python 3 and Apache 2.4.7 were used, and mod_wsgi version 3.5 was installed as the Apache module to allow it to run Python applications supporting Python WSGI interface. No other Python frameworks were deployed in order to keep the development work as rudimentary as possible, without hiding any of the possible issues behind abstraction brought by frameworks such as Django. Both these virtualized machines were given a private IP addresses from 192.168.1.0/24 subnet with DNS.

During the thesis' research part, some programming work was done with Eclipse and some minor programming work was done with lighter editors such as Notepad++ and gedit. Backups were handled as virtual machine snapshots, with the capability of resuming certain states in the development work, as opposed to just copying the code files themselves. Version control was not used for this thesis because of the light codebase and backups. Bug tracking was done on ordinary office tools such as Notepad and Excel.

4. UTILIZING SDN FUNCTIONALITY VIA HP VAN SDN CONTROLLER NORTHBOUND API

4.1 Description of new network management functionality

The next step in the thesis scope was to create some new network management functionalities based on the SDN technology palette. More precisely the work aimed to creation of a web-based management tool for both normal web browsers and mobile devices, branded with the TeliaSonera logotypes and CSS styles, and having two distinct functionalities: switch rerouting and QoS (quality of service) modification.

The core idea is that a user, e.g. a network operations center (NOC) employee could on a single web page insert two IP addresses, one for destination and one for source, and order the network to reroute traffic between these two endpoints not to go through the default, shortest path, but a longer path, that, in the conceived use case, would host an intrusion detection and prevention system (IDPS) and/or deep packet inspection (DPI) device along the datapath, checking, if the traffic rerouted through it is legit and allowed to continue on its path in the network. The change would have to occur instantaneously, without any detectable packet loss in repeatable tests.

The other functionality in question is aimed into a situation where a traffic stream (or a flow) is deemed unnecessary and unimportant, in a corporate environment where business critical data must precede other data such as video streaming, social media or network gaming. In a situation such like this, a NOC employee should be able to enter a destination and source IP addresses into a single web page, and with the press of a single button, the network should start treating the flow between these two endpoints as lower importance data, that will be served (forwarded, routed, etc.) after the more important and time-critical data has passed. This will happen by modifying TCP/IP headers in packets on that flow, more precisely by rewriting the quality of service (QoS) –field with the value of DSCP 12, class 1 (lowest), medium drop. The change would have to occur instantaneously, without any detectable packet loss in repeatable tests.

These use cases could be widened by allowing the NOC employees to be able to define the route into which the data should be rerouted (in this use case it is statically configured to be the aforementioned IDPS/DPI –system) and/or allowing the NOC employee to be able to manually define the QoS values which would then be rewritten for the de-

tected dataflow. These changes would be trivial from the implementation perspective (giving the user ability to change pre-defined values in a HTML form).

The use cases suppose that the trigger for these changes have come from some external source, for an example from a network analyzer software relying on NetFlow- and/or NBAR/NBAR2 -protocols (such as Plixer Scrutinizer). In this use case, the information within a system like that would have been interpreted by an administrative user, who would have recognized the need for network changes planned here. In that sense full automation of network changes is not developed within the scope of this M.Sc. thesis, as the use case assumes human interaction to an event detected elsewhere.

4.2 Technical design

The solution was developed with Python programming language in a Linux environment. The Python code resulted in a single Apache-serviceable (via mod_wsgi Apache module) web page that contained the specified functionality, presented as a HTML page using HTML forms.

The specified solution ended up using only two separate API methods (but a many number of calls per especially the other method), one of which being authentication API method (`/sdn/v2.0/auth`) and the other being the flow creator API method (`/sdn/v2.0/of/datapaths/{dpid}/flows`). The explanation for the low amount of necessary API methods is the fact that the flow creator / flow modifier method acts as the core component in managing OpenFlow –based networks, and what it does varies greatly based on the JSON-payload delivered with the HTTP queries.

4.2.1 Reroute -functionality

When user on the web page attempts to reroute traffic in the network by giving the page source and destination IP addresses as arguments and forcing reroute by pushing the HTML button, the Python script activates and firstly reads the whole submitted form. If the page received some POST data on the page load, and the POST data contains an indicator that it came from the click on the 'Reroute' –button (if the POST data contains a field embedded in the Reroute form called 'reroute' and its value is 'true'), firstly (just after escaping all the POST data for information security purposes) a login on to the SDN controller is attempted.

The login is currently done by forming a JSON-serialized payload for a HTTP POST, containing username, password, and mandatory domain information for the SDN controller. Then this information is sent to the SDN controller's REST API's authentication method (`/sdn/v2.0/auth`) via HTTP POST, using third party Python HTTP module called Requests, instead of Python's standard urllib2, which has been claimed to be broken. All the HTTP traffic and REST API queries are sent as HTTPS encrypted traffic in

the port 8443, however for testing purposes SSL certificate verification is disabled in the HTTP POST phase in Python's Requests–library. This is due to the fact that testing environment doesn't contain trusted certificates, but self-signed certificates which would not pass certificate verification. In real-life environment certificate would have been signed by a trusted Certificate Authority (CA). TeliaSonera does offer this service and is the only widely trusted Certificate Authority in Finland (e.g. the only trusted CA that for instance Ubuntu 14.04 LTS or Firefox (on any platforms) accept by default). The Python login on the SDN controller happens with code snippet presented next:

```
	payload = {"login":{"user":"sdn","password":"skyline","domain":"sdn"}}
	headers = {'content-type': 'application/json'}
	r = requests.post('https://192.168.1.113:8443/sdn/v2.0/auth', da-
ta=json.dumps(payload), headers=headers, verify=False)
```

From a successful authentication, an X-Auth-Token is received as a reply from the API query. This token is parsed with the help from Python's native json module and saved to a Python variable for a later use with the row of code like this:

```
token = r.json()['record']['token']
```

After a successful acquisition of an X-Auth-Token, the preparation for rerouting the network data flow is started. The preparation in practice means creating a JSON-serialized version of OpenFlow 1.0 flow-modification command. The serialization format is HP-specific, but is so straightforward that other actors could have easily have adapted exactly same kind of serialization. There will be a chapter later in this thesis which compares HP controller's API request's message body variations to OpenDaylight API request's message bodies. An example serialization from developed prototype code is shown in Program 3.

```
payload1 = {
	"flow": {
		"priority": 30000,
		"idle_timeout": 60,
		"match": [
			{"ipv4_src": source},
			{"ipv4_dst": destip},
			{"eth_type": "ipv4"}
		],
		"actions": [{"output": 2}]
	}
}
```

Program 3: *Serialization of an OpenFlow command.*

Basically the JSON serialization (placed into a Python variable called "`payload1`") is a direct representation of OpenFlow's standard, where a flow is defined by its matches. In this case a flow is defined in such a manner, that all the data in the network which has a source IP address defined in the variable "`source`", destination IP address defined in the variable "`destip`" and the traffic protocol is IPv4 fall under the category of this certain flow.

The `priority`-field defines the order in which the SDN controller applies the flows, or rules from the flow table. Value of 29999 is the default priority on the default-generated flow rules on the SDN controller, and when in this example the flow is set to the priority of 30000, it means that it supersedes all the other flow rules, except for ones that have been possible created by some other third party management tool which has set their flow priorities higher than the one over default of 30000 used by the network management tool created in this thesis.

SDN controller reads the flow table in such a manner that when a new data stream is detected on a network, a sample (the first packet) of it is sent to the SDN controller by the network switches. The SDN controller then starts comparing its characteristics defined in the match-fields to the set of rules the SDN controller has in a prioritized order. When SDN controller finds a flow from the flow table that matches the new data stream on the network, that flow and only that flow activates, that stream is handled per instructions written in the activated flow, and rest of the flow table is never read for this flow, until possibly after a certain time has passed after the data flow has stopped on the network.

This time is defined in the `idle_timeout`–field in the JSON-serialization, and the value of 60 means that 60 seconds after the data stream on the network has stopped, this flow will be deleted from the network devices, and if after that point in time the same data stream enters the network again, network devices do not immediately know what to do with that stream, and send the first packet for inspection at the SDN controller again, and the SDN controller would then return the same flow rule to the network devices, unless new rules with higher priorities have been written on to the SDN controller during this timeframe.

The `action`–value in the JSON–serialization contains instructions on how the data should be processed by the network devices. In the conceived example in the prototype network topology, the default action that the network would do for a data stream would be to forward it always to the port 1 on the network switches, in order to be able to get the data into its destination shortest possible path available. However, the point of the whole prototype application was to cancel this default behavior by rerouting the traffic to another switch port, eventually getting the data stream to its destination, but via a

longer route, which will have an IDPS system making a deep packet inspection for a suspicious data flow. Therefore the action is to output the data into the port 2.

One thing to note is that a similar flow has to be written into the flow tables of each of the network nodes (virtual switches) on the longer route the data is wished to travel. This is due to the fact that if such task would not be done, the first rerouted destination (the first hop) would look at the data and its destination and decide that this data should be treated as normal (if no abnormal flow actions are defined as flow rules) and send it back to the shortest route, creating a network loop. This is also one of the weaknesses of SDN-managed networks: chances for network loops in badly programmed network configurations increase greatly.

In practice, this issue is handled also by writing the new flows to each of the network devices' flow tables, but also taking care in the order in which the flows are written to the devices. If the first action would be to write the flow to the first hop of the data, rerouting the data to the second network device on the longer route the data is supposed to travel, the data would instantaneously reroute there, but the second device would not yet know what to do with it. It would ask the SDN controller what to do with data stream like that, and because our SDN controller setup doesn't have reactive application running which would know we're trying to achieve rerouting via the longer route, the SDN controller would reply to the device, saying that the action is '`NORMAL`', which would mean passing the data to the shortest possible path. This would mean back to the first hop, which would want to pass the data back to the second device which just sent the data to it. Thus, at least a momentary network loop would exist.

The solution is to start writing the new flows to the flow tables of network devices starting from the last hop of the topology the data would need to travel. In our use case, that would mean the network device with `datapath_id` (which is basically a MAC-address coupled with unique two-digit hex value in the beginning of the MAC-address) 00:00:00:00:00:00:00:03. After that, a similar flow creation request via SDN controller's REST API would be targeted to 00:00:00:00:00:00:00:05, then 00:00:00:00:00:00:00:04 and last to 00:00:00:00:00:00:00:01. The order described here may seem counterintuitive, but it must be remembered these datapath IDs represent unique IDs based in real life on unique MAC addresses. Therefore the datapath IDs in our example are not ordinal numbers, but just random identifiers. The order of the datapaths can be verified from the topology in the Figure 15. The API requests are formed like shown in the next code snippet, and the request is repeated for each of the datapath IDs. Note that the X-Auth-Token is passed in the request headers as a JSON key-value pair.

```
    headers =
    {'content-type': 'application/json', 'X-Auth-Token': token}
    r = requests.post('https://192.168.1.113:8443/sdn/v2.0/of/datapaths/
00:00:00:00:00:00:00:03/flows', data=json.dumps(payload1), headers=headers,
verify=False)
```

4.2.2 Quality-of-Service modification -functionality

The quality of service modification is achieved much like the reroute-functionality was implemented. After user has inputted the source and destination IP-addresses to the web management tool, and clicked on a button on the web form to activate the change, an authentication with the SDN controller occurs and the X-Auth-Token is acquired and saved into a Python variable.

This token is then used in order to query the flow modifier method in the SDN controller's REST API with similar match conditions which were used with the SDN reroute functionality. The idea is to compare the data flow in the network to destination and source IP addresses and also making sure the protocol is IPv4.

The action-part of the API query message body has some differences, though. In the action part, because in the QoS-modification -use case any kind of need for reroute from the optimal shortest path was not defined, SDN controller is told in OpenFlow terms that the traffic should be output in the network to port, which would be the normal behavior for the device if the SDN controller would not exist at all. In practice this translates to a situation where the network devices route the traffic via the shortest path. In OpenFlow terminology this means, that the "`output`" key's value is "`NORMAL`". This part is necessary for the flow to work at all, if it was omitted, the data stream would not be output anywhere from the network device, under the guidance of this flow. Effectively the QoS-modification part of the action wouldn't work either.

The real modification of Quality of Service occurs in the API query's action part, within the key "`set_field`". The key's value is a JSON-structured instruction in itself, which would allow the management tool to instruct the network to change any of the values of TCP/IP header fields of the matched data stream in the network node, where the flow is eventually applied. In this use case the goal was to modify data stream's Quality of Service to be on a very low level, allowing prioritization of other, more important network traffic instead of the detected, non-business-critical data between the defined source and destination IP addresses. The name of the QoS field in the TCP/IP –header is `ip_dscp`, and its numerical value of 12 translates to Class 1 (lowest), medium drop treatment in the network, meaning a level above the least important traffic possible. The API query is presented in the Program 4.

```
payload1 = {
        "flow": {
                "priority": 30000,
                "idle_timeout": 30,
                "hard_timeout": 30,
                "match": [
                        {"ipv4_src": source},
                        {"ipv4_dst": destip},
                        {"eth_type": "ipv4"}
                ],
                "actions": [
                        {"output": "NORMAL"},
                        {"set_field": {"ip_dscp": 12}}

                ]
        }
}
```

Program 4. *API request for QoS rewrite.*

Like with the reroute-functionality's implementation, also the QoS-modifier has to be precisely designed to target the correct network node with the flow, or otherwise the flow would not work in any sensible way. The correct point in the network in this use case is the first SDN and OpenFlow-enabled node, where the traffic enters the network. In the development environment's case, that's the first switch where the end-system behind the IP address defined in the Python variable "source" is located. Effectively this means that if the user inputted source IP address of 10.0.0.1, the closest network device would be the one with `datapath_id` 00:00:00:00:00:00:00:01, and if the user inputted the source IP address of 10.0.0.2, the target for the flow creation would be the switch behind `datapath_id` 00:00:00:00:00:00:00:03. The targeted switch is once again defined in the Python code, when doing the HTTP POST with the Requests-modules `.post`-method.

4.3 Testing and validating the created service

Despite the fact that the development work was moved to a virtualized environment in the middle of the development work, the physical environment remained online and accessible throughout the timeframe of the thesis writing process. This allowed two-way approach to the testing and validation.

The network simulation suite called Mininet, coupled with the virtual switch application Open vSwitch allowed creating varying scenarios, testing the performance boundaries of the SDN controller with huge (from a hundred to thousands network devices) networks and both exotic and common network topologies (ring, mesh, full mesh / fully connected, tree, bus etc.), while making these changes to the network commonly with a

small amount of Python scripting or just changing parameters in CLI-commands (mininet natively supports some topologies such as tree).

Regardless the topology or the amount of devices in the topology, the created SDN-based network management tool worked with no clear changes in its behavior or reliability. Both use cases (rerouting traffic with a click of a button and modifying a Quality of Service of a dataflow between two defined endpoints) were satisfied with the prototype created and with just some feature additions functionality could possibly be brought into real-life testing for NOC workers. Testing was done on three levels; viewing the visual changes in the SDN controller's user interface, tracking traffic on the network with traceroute–application (especially in the case of the SDN reroute) as well as directly accessing and dissecting TCP/IP headers of the created traffic with WireShark.

One issue with virtualized network running as the testing platform was the fact that because of how OpenFlow–based SDN-networks work (sending a first packet of a new, unforeseen data stream from the switch to the SDN controller, SDN controller analyzing this packet and making decisions on how to treat it, and the ones that follow it, and sending the instructions in the OpenFlow-form back to the switch or switches), a short delay in establishing the connection in the networks is introduced. The problem is that virtualized network runs estimated at least ten times faster than any real life physical Local Area Network, based on round-trip times of a ICMP PING tests. This made it more difficult to analyze the performance impact of the implemented SDN environment's way of treating newly detected data flows on the network. An example of the perceived network delay is presented in the next command line interface snippet, a view inside mininet-application, which introduces a new flow into the network when host 1 ("h1") starts pinging host 2 ("h2") for the first time.

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=3.98 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.442 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.091 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.081 ms
64 bytes from 10.0.0.2: icmp_req=5 ttl=64 time=0.079 ms
```

As can be seen in the snippet of the mininet output, the first packet travels about 4 milliseconds, while the rest travel in fractions of a millisecond. This same behavior is present in physical SDN networks too. Similar performance tests were performed in the physical environment as well, but there the problem was that the size of the environment was so small, that the results were not directly comparable or even directly interesting.

Based on the gathered data some estimates on the severity of this issue could be generated, however. A delay of from a few milliseconds to less than 50 milliseconds in the first packet of the network is not severe. This delay can be also minimized with faster network equipment and faster hardware on the SDN controller, which is proven by the fact that in virtualized environment the first-packet delay is minimized to a value which would be considered a good value for normal physical networks existing in offices and homes today (3,98 ms). Also because of how the OpenFlow–directed SDN-network is architecturally designed to work, all the packets after the first one move with the network speed that would be achieved without SDN control. This basically renders the issue insignificant, but still an issue that must be raised when considering the pros and cons of SDN-enabled networks.

The fact that there exists a way of validating and testing newly developed functionality in a virtualized environment as opposed to having to use specific, often costly hardware is a great thing considering the fact that SDN-proponents are hoping to see an ecosystem booming around SDN application development. This is significant boost for smaller companies and even independent developers who are venturing in to the SDN development realm. Basically this is a positive thing for TeliaSonera also, which very often uses subcontractors in software development work due to the fact that its internal software development resources are limited, and work like that is not a core competence of the company.

The result of the testing process was that no detectable faults remained in the system and the system worked in both virtualized and hardware environments within the boundaries the environment themselves set to the possibilities of using the created web application's features.

4.4 Analysis of the development process

The development process was not formally defined in the beginning of the thesis work. This was because in the first stages the work was experimental and it acted almost like a proof of concept for the thesis worker himself, because in the beginning it was not even clear that all the goals of the thesis work could be theoretically achieved.

Because the thesis work was executed by a single person and the goals were unclear at the beginning of the thesis timeline, and because the requirements were predicted to possibly change during the development process, an agile way of working was chosen. Choice was made because it was impossible to specify the end-product in the beginning of the process, and because there had to be room for changes in goal-setting based on newly gathered knowledge in SDN development limitations and technical restrictions. The development process was not formalized as any specific agile method and/or pro-

cess framework, but it embraced iterativeness, adaptive goal-setting and embraced code and working software over documentation, fulfilling many of the agile cornerstones. The agile method that is being applied in the daily work of the thesis writer at TeliaSonera is Scrum, but adapting that to a one-man process made no sense, as the thesis was not worked for as part of the ongoing daily routines in the workplace.

The process worked as predicted, and no significant process-related problems were encountered during the development work.

4.5 Analysis of the codebase created

The main significance of the created codebase, and the software it forms, is two-fold. Firstly, it can be used as a real-life demonstration inside the company for executives in order to make them easier understand the significance and features of Software-defined Networks. In this area the created web application has already proven successful and will continue being used for similar demonstrations in the future.

Secondly, while it is clear that the created tool is only capable of working as a demonstration and a prototype, it could easily be used in order to make new programmers acquainted with HP VAN SDN Controller's REST API, development against it and through the demonstration some of the core ideas on OpenFlow architecture (and limitations placed by it). It is easy to see that the created tool would be opened and dissected once an officially defined SDN development project started in the TeliaSonera context. The project itself would most likely start a new codebase and create its end-products from scratch, but the tool's code and also the knowledge gathered by making it and demonstrating it in function would greatly aid the project and especially speed up the starting phase. An SDN-development project is currently being defined within the organization and such a project is expected to start during the fall of 2014.

The thesis project did not end up with and end product that could be categorized as a Python library of any manner. Some code could possibly be copied and pasted to some other project, but one essential limitation is that most of the systems used by TeliaSonera in network management are developed with Java, making Python code directly useless in this manner. The work results could, however be packaged with some documentation to form some sort of internal SDK for Python development in HP VAN SDN Controller platform. Such a task was not finalized within the thesis' scope.

4.6 Moving the solution to another SDN Controller platform

The next question this thesis aims to answer is that how much work would it require to move an already finished SDN application from one SDN controller platform to another controller platform. In this thesis, an SDN application in the form of web-based man-

agement tool for NOC employees was created. A fully functional prototype was tested in virtualized and hardware environments and it was used for demonstration purposes within the organization. This application was created utilizing HP VAN SDN Controller's REST API. The HP's platform was chosen based on Analytic Hierarchy Process (AHP) and the selection process was explained and fully walkthrough in chapter 3. As the results of the AHP described in aforementioned chapter point out, the second best choice for SDN platform based on the needs of this thesis would be OpenDaylight. The ODL controller was chosen as the target platform for the question what happens, if the SDN controller is changed.

It is clear that the lack of northbound standardization makes it impossible to straightforwardly just change the controller and keep everything else intact. Thanks to the OpenFlow standardization (of which the versions 1.0 and 1.3 have become the most used incarnations) the southbound direction can be handled relatively easily. In practice this means that if an existing SDN controller would be changed to replace an SDN controller with the same IP address, the existing hardware network devices (e.g. switches) would not disrupt their service because of the change, and keep continuing as they would always, with the default configuration of the SDN controller that was just used to replace the earlier one. This can be pointed out with the switches' OpenFlow configuration presented in the chapter 3.3.6. The SDN controller is only identified by its IP address, and only other meaningful parameter passed in the configuration is the used OpenFlow–protocols version (which in the example was 1.3). The SDN controller is not being identified by any kind of vendor specific means, but just defined by the version of the standard used for communication, and the network location the controller should be found at.

The northbound direction, or the SDN applications that bring the functionality to the SDN networks, is the more problematic part. SDN controller could be replaced and network would continue operating, but because the created SDN applications on one platform wouldn't be directly transferrable to other platforms thanks to the lack of a standard, the special functionalities brought to the network management would cease working.

In this chapter the thesis will analyze the amount of work that would go into refactoring an HP-based SDN application into OpenDaylight–platform. The chapter will make direct REST API comparisons on method per method –basis, makes sure both APIs make the same things possible, discusses data interchange formats for request bodies and later attempts to present two more robust solutions for this problem.

4.6.1 Direct comparison of HP and OpenDaylight REST API methods

Because HP SDN VAN Controller –platform's API was thoroughly described in the chapter 3 and its used examples described in the chapter 4, it will not be described very

thoroughly here. This chapter's goal is to find corresponding REST API methods and define the necessary payloads for these methods in order to achieve similar functionality that was developed on the HP platform.

4.6.2 Authentication and serialization formats

The first clear difference between HP's REST API architecture and OpenDaylight's architecture is encountered as early as when designing the way the client application should authenticate itself to the REST API. HP's platform used separate REST API method for authenticating the user with username, password and domain information once before using other methods at all. This information resulted in an X-Auth-Token as a response, which would stay valid for 24 hours. This token would then be always passed in HTTP headers as authentication information to allow usage of other methods. Therefore, the programmed application would need to keep track of these X-Auth-Tokens either on server-side or in client-side cookies or in session.

OpenDaylight's solution for authentication is different. OpenDaylight uses HTTP Digest and HTTP Basic authentication. Basically this means that on OpenDaylight's REST API there is no corresponding method for HP VAN SDN Controller's `/sdn/v2.0/auth` –method and no X-Auth-Token would be passed in the headers of every API request after that, until 24-hour timeframe had passed and the token would be invalidated.

The difference between HTTP Digest and HTTP Basic is that with HTTP Digest the SDN controller's admin could give API access to only some parts of the API based on the username. For the purposes of this thesis, an API user with full access to the API was used and HTTP Basic was used as the authentication method. OpenDaylight's API supports also HTTPS, as does HP VAN SDN Controller's, which is crucial when transferring authentication data via HTTP request headers.

While the authentication on HP's platform was described earlier in the chapter 4, authentication with HTTP Basic –method of OpenDaylight controller, using the same Requests –module of Python that was used with the implementation on HP platform, would happen with the way described in the next code entry. In real life query, the "`user`" and "`pass`" would be replaced with real username and the corresponding password.

```
from requests.auth import HTTPBasicAuth
requests.get('https://
192.168.1.113:8443/controller/nb/v2/flowprogrammer/default/',
auth=HTTPBasicAuth('user', 'pass'), verify=False)
```

A separate authentication API method is never used with the ODL platform. The authentication occurs as part of every REST API query based on HTTP features.

Data serialization differs also between the platforms. For example, if the REST API of the OpenDaylight SDN controller is called with an HTTP GET on the method `/controller/nb/v2/flowprogrammer/{containerName}`, a list of flows configured on a given container could be returned. The value "`default`" could be used as the `containerName` value, and then the API would return a list of all the flows in all nodes of the network known by the SDN controller like in the Program 5.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<list>
   <flowConfig>
      <installInHw>true</installInHw>
      <name>flow1</name>
      <node>
         <id>00:00:00:00:00:00:00:01</id>
         <type>OF</type>
      </node>
      <ingressPort>1</ingressPort>
      <priority>500</priority>
      <etherType>0x800</etherType>
      <nwSrc>10.0.0.1</nwSrc>
      <actions>OUTPUT=2</actions>
   </flowConfig>
</list>
```

Program 5. *XML-serialized network topology information.*

The returning information could be serialized in either UTF-8 encoded XML 1.0 like in the Program 5 or in JSON like in the Program 6. The shown data in the two examples signifies a simple network consisting of only one switch.

```
{"flowConfig": [
{
"installInHw": "true",
"name": "flow1",
"node": {
"type": "OF",
"id": "00:00:00:00:00:00:00:01"
},
"ingressPort": "1",
"priority": "500",
"etherType": "0x800",
"nwSrc":"10.0.0.1",
"actions": [
"OUTPUT=2"
]}]}
```

Program 6. *JSON-serialized, identical information.*

The shown examples of possible response bodies reveal many differences in both platforms' REST APIs, and also the way the OpenFlow–protocol is described in a serialized form with OpenDaylight. Because the method used here as an example (the listing of all the flows in the network) was not used at all in the SDN application implemented on HP platform, the differences will not be dissected any further than bringing up the fact that OpenDaylight API supports both XML and JSON in both request bodies and response bodies as opposed to HP platform's singular support for JSON in either direction. The used format is controlled by HTTP request headers, where the `Content-Type`- and `Accept`-fields are used as the way of communicating which serialization formats are being used. An example of such communication is highlighted in the next code entry, using Python query with Requests module.

```
headers = {'content-type': 'application/json', 'accept': 'applica-
tion/json'}
requests.get('https://
192.168.1.113:8443/controller/nb/v2/flowprogrammer/default/',
auth=HTTPBasicAuth('user', 'pass'), verify=False, headers=headers)
```

The example shown in code above tells the API to expect JSON as the serialization format in API requests and asks API to return JSON as the serialization format of API responses. This is the desired behavior in this thesis' OpenDaylight implementations as well, because the HP platform used JSON for serialization, and we are looking for the least possible amount of work in converting the existing SDN application.

4.6.3 Flow creation and modification

After the authentication had been taken care of, and serialization issues had been cleared, it was time to proceed into creating similar functionality with the OpenDaylight platform that was created with HP's SDN platform earlier.

The reroute-functionality's core idea is to create flows into the network devices, branded with higher priority than the default flows. These new flows would then make OpenFlow–matches to source IP address, destination IP address and IPv4 protocol. After a successful match the flows would then instruct the device(s) to output the data to the switch port 2 instead of the default port 1, which would normally lead to the shortest path.

The described functionality can be achieved with the OpenDaylight-platform using the Flow Programmer REST API –module. The API method "`/controller/nb/v2/flowprogrammer/default/node/OF/{datapath_id}/staticFlow/{flow_name}`" is used to create and modify flow configurations. Unlike HP's way of using HTTP POST to create a flow, OpenDaylight's REST API uses HTTP PUT to achieve both flow creation, and flow modification. Basically, if the flow already exists, the introduced flow will replace the current flow and an update occurs. If the flow does not exist, a new flow is created.

The method's URI reveals the next difference between HP and OpenDaylight REST APIs. OpenDaylight assigns names to flows, which are unique for those flows within a single network node (e.g. within a switch). HP's flow architecture does not have a similar naming concept at all. There flows are identified by their OpenFlow matches – a unique match can only have one possible set of actions defined within the flow table. If we consider a web application developed on OpenDaylight–platform intended for creating and possibly updating flows, the application has to keep track of the flow names. The other solution would be the application being programmed to read all the flows from the network with the API method presented in the chapter 4.6.2, and modifying its behavior based on this information. Keeping track of the flow naming (and creating a naming scheme in the first place) adds a light layer of complexity to the application, while at the same time allowing more elegant ways of flow modification than on the HP platform. On the HP controller, when overwriting a flow, the flow has to be defined with identical set of OpenFlow matches, rather than just specifying the name of the flow the application is wishing to update. The "`datapath_id`" variable in the ODL-method's URI is used like in HP environments: it refers to the network node or network switch the code is trying to manipulate.

The next difference between HP and OpenDaylight was already revealed by the Program 6. It is the fact that while HP's platform is supporting only OpenFlow as the underlying SDN-technology, OpenDaylight was built from the scratch to support many southbound protocols and standards. So when using the OpenDaylight's REST API, every query must define in the request body, what type of a flow the API application is attempting to install. This is done with the key "`type`" and its corresponding value, e.g. "`OF`", where "`OF`" would refer to OpenFlow. If we were to create an API request that would install an identical flow with the one created on HP platform for SDN reroute purposes, the request body in JSON would be composed as in the Program 7.

```
{"installInHw":"true",
"name":"flow1",
"node":{
"id":"00:00:00:00:00:00:00:03",
"type":"OF"
},
"priority":"501",
"etherType":"0x800",
"nwSrc":source,
"nwDst":destip,
"actions":[
"OUTPUT=2"
]}
```

Program 7. *Request body for flow creation in the reroute use case.*

This request body, when sent to the `staticFlow`-method and embedded with the HTTP authentication would create an OpenFlow-flow to a network node at ID 00:00:00:00:00:00:00:03 with the flow name “`flow1`”. This flow’s priority would be 501, which is one higher than the default value of 500 in OpenDaylight environment. Similar prioritization was done with the HP’s case by increasing the default value of 29999 by one, giving us the priority of 30000. The newly created flow matches inbound traffic to network source (“`nwSrc`”) and network destination (“`nwDst`”) IP addresses given in Python variables “`source`” and “`destip`”, just like the HP implementation. The flow also matches the “`etherType`” to the value “`0x800`”, which is OpenDaylight’s way of referring to IPv4. HP platform used clear text for “`etherType`” key with the value “`IPv4`”. The action done for the traffic matching to these conditions is outputting it to the port 2, like in HP’s case.

In the implementation of quality of service -change with OpenDaylight–platform, the request body is much like the one used for rerouting, except for necessary modifications for the datapath ID signifying the target device (which would have to be 00:00:00:00:00:00:00:01, like in the HP case). The most significant difference happens with the `actions`-part, which is shown next.

```
"actions":[
"OUTPUT='NORMAL'",
"SetNwTOS=12"
]
```

OpenDaylight uses different terminology for making the change. Instead of QoS, OpenDaylight refers to ToS (type of service), but the documentation at OpenDaylight wiki reveals that the NwTOS referred to in the JSON notation really stands for Ethernet DSCP field. This makes the “`SetNwTOS`”-key work just like HP VAN SDN Controller’s “`ip_dscp`”-field does.

The last differences between the APIs can be discovered on how both platforms use HTTP status codes after calls to the interface are made. The standard HTTP response codes such as 201 for successful action, 400 for bad request, 401 for unauthorized, 404 for not found and 503 for service unavailable are common with both platforms. However, the OpenDaylight platform implements few other HTTP responses, which can help with error handling and debugging. These additions in ODL do not have counterparts in the HP environment.

OpenDaylight uses HTTP status code 406 for not acceptable requests, which in here refer to a situation where the controller could not operate on the default flow container, because other flow containers are active. HTTP status code of 409 represents conflict, which occurs when a static flow entry attempts to write itself to the network with a conflicting flow name, or a conflicting flow configuration. OpenDaylight also uses HTTP status 500 for Internal Server Errors when static flow creation fails. The failure reason is

included in HTTP Error response. Unlike HP platform, OpenDaylight also differentiates the successful flow creation and modification actions with the status codes of 200 for modification and 201 for creation. This would not be needed for HP platform anyway, because that platform takes advantage of both PUT and POST HTTP methods for update and creation correspondingly, so the application always knows which task it is handling, unlike OpenDaylight, which performs both with HTTP PUT.

4.6.4 Other differences

While HP solution contains Rsdoc as a computer-readable API reference, and a JSON Schema as the data specification, OpenDaylight on the other hand offers a WADL describing the Flow Programmer REST API. WADL is the REST-equivalent of SOAPs WSDL-format, which allows software to create the API calls by creating a class containing the code for all the calls, based on an XML-serialized description of the whole API. WADL enables the same functionality for REST APIs, but has not been widely adopted and has not been officially standardized.

4.6.5 More robust solution through standardization or encapsulation

Showstopper-type issues were not encountered while transferring the functionality developed on HP SDN platform to the OpenDaylight controller. It is clear that the transfer is not possible without programming work. The amount of work required could be described as moderate to major, and effectively is quite close to the actual API programming work done, when the application was initially developed for the original controller.

Instead of modifying the existing software every time an SDN controller is replaced in a network, there are two more robust ways of solving the issue, which could act as the answer for complete SDN controller interchangeability. Firstly, if the Open Network Foundation's Northbound Interfaces Working Group could finish its work and release a northbound API standard the situation might change quickly for better via controller developers unifying their products' capabilities, functionalities and API calls. To speed up the process it should be evaluated whether it would be reasonable to make the northbound API standard mandatory in all officially OpenFlow-implementing SDN controllers. Standardization of northbound interfaces would probably be the best situation of all available solutions, but it currently seems far-fetched. It is also realistic to realize that just the releasing a standard is still a step away from wide-scale adoption of the standard. Without powerful means and methods for making controller vendors to adopt the standard, the northbound API standard could very well remain just a document that has no significance in real life deployments.

Industry veterans have also expressed their doubts about ONF Northbound Interfaces Working Group. A common consensus is that the working group is too late already, now that all the large vendors have invested large sums of money in vendor specific implementations. If guidelines or memos on the subject would have existed long before the development work on all the controller choices was started, meaning more than a year ago, the situation might have been better. Right now no one has expressed the will for standardizing the northbound feature set or functionality.

The Open Networking Foundation was also contacted as part of this thesis on the subject of northbound API standardization. The purpose of contact was to offer help in use case creation for the standard's needs. The contact was made via official channels as a managerial level representative of TeliaSonera, one of the world's largest telecommunication companies, but no reply was received from ONF, suggesting either lack of interest of lack of activity within the Northbound Interfaces Working Group.

Thus, in reality, formal standardization may be a far-fetched idea. Nevertheless, a de facto standard might still spring up from the fact that many of the larger networking vendors, such as Cisco and others, have based their own SDN controller products on the codebase of the OpenDaylight open source project. In practice this could lead to a situation, where many of the competitors' products could share common features, one of which could be the northbound API specification.

Instead of formal standardization or a number of de facto standards, the more achievable road for SDN controller interchangeability would be a creation of a middleware application, which would hide the controller-specific APIs behind its own API functions and methods. The idea for the middleware would be to support all significant SDN controllers on the market. This figure is still in very manageable numbers, because not even all the controllers introduced in this thesis would have to be included. Implementing support for almost ten controller products would cover the major market share. Also, this middleware would need to be kept up to date with all the time advancing SDN controllers. Therefore the best possible situation could be if open source community noticed the need for a middleware such as this. Active developer community could keep up with the changing landscape of SDN controller market. A single contributor such as TeliaSonera does not have the resources or a business case for work like this. For TeliaSonera, the most sensible way would still be just modifying the SDN application always when a need arises through changes in SDN controller interfaces or changes in used SDN hardware.

5. DISCUSSION

5.1 The controller choice

In this thesis the choice of the SDN controller was made with the Analytic Hierarchy Process. The process itself is a proven method and thesis pointed out to references of it being used for a similar task, as well as to a general recommendation of AHP for software choice making.

The results obtained from a full run of AHP pointed out that the two most noteworthy SDN controllers for this thesis' purposes were the HP product and the open source controller OpenDaylight. The results showed that the final quality values with the customized criteria for HP and OpenDaylight products were around twice as high as the three other competitors, who were fairly near each other quantitatively. Experimentation with the AHP system pointed out that these groups were so clearly distinctive in the results that small to moderate adjustments in the system could not shift the general situation, where two of the controllers were in a different class than the rest of the five. Greater adjustments obviously made changes to the end results, but such adjustments did not have anything to do with the perceived reality of the devices.

The order of the two platforms of choice was very easily upset, however. This should not be surprising though, as the difference between first and second platform quality values is only 0.004, which could be translated into 0.4%. Even the slightest change in the system, which gave indication of a statement that OpenDaylight would be superior to the HP platform, changed the order of these two entities in the end-results. Even a great number of random changes to the system not directly involving HP or OpenDaylight products managed to make the order change. This is due to the pair-wise nature of the AHP, where every entity within the comparison system has a comparative relation with every other entity. When such relationships are present in a number of attributes, it is possible that shifting the system's outlook on a third product to be of higher or lower quality, the system's stand on one or both of the products placed in the first and second spot could sway a little. Sways could differ, if for instance HP platform was implied to be only a little better than e.g. Floodlight on a given attribute, while at the same time OpenDaylight would have been indicated to be somewhat better than e.g. Floodlight. If Floodlight's value in the system changed, the change would translate into different changes in HP's and OpenDaylight's final quality values.

As a conclusion, AHP results should not be treated as strict ordinal results but rather be abstracted to the form that points out that with the selected criteria, HP and OpenDaylight were clearly the two most suitable SDN platforms, and they were essentially equal in quality. At the same time, the three other candidates were equal in quality between each other, but with a huge gap to the best two.

Analytic Hierarchy Process also includes a way to determine consistency ratios (CR) for the initial values. Experimental adjustments quickly started to cause problems with CR, essentially forcing the person conducting the process to recheck the values and trying to present them in more correlative, cohesive manner. The consistency ratio calculations are not a mandatory part of the Analytic Hierarchy Process, but those were calculated while making the experiments. The calculations also proved that the initial values were consistent, because for example the most significant sub-attribute's ("API scope and extent") consistency ratio was 0.9% and the second most significant sub-attribute's ("API method descriptions") CR was 3.9%. All the consistency ratio values were under 10% for the used initial values in the whole system. According to the general interpretation, if the ratios would have exceeded 10%, the set of judgments may have been too inconsistent to be reliable. (Paraskevopoulos, 2014) No such issues were encountered until experimental customization of the AHP.

Correlation Ratio was not a problem, however, when the system was manipulated logically, by building a new scenario where products' features were altered consistently when compared each of the products. The original results obtained from AHP were clearly distinctive between two groups of SDN controllers with difference of 100% in the final quality value. Thus, results proved to be clear.

5.2 Criticism and risks

SDN has already proven that it can be deployed in certain environments and situations, most notably within data centers. Data centers even were massively considered when SDN was initially designed. Software-defined Networking has made huge promises and has shown potential in other environments as well, and it has had some successes in its early era, such as the creation of the open, over watching authority, Open Network Foundation, standardization of the so-called southbound interfaces and for instance the production level deployment in Google Networks. Still, there are some uncertainties in the path of the SDN in becoming the next revolution of networking.

Some of these issues and uncertainties are more relevant than others. One often cited issue with SDN is related to the claim that the concept introduces a single point of failure to the SDN-enabled networks. Traditional networks, depending on the designed network topology, can recover from disconnection of a single network node. Routing protocols would notice the change in the network and make changes to routes within

tens of seconds or at most minutes. Therefore, traditional networks, unless built with non-redundant topology, do not contain so-called single-point-of-failures. Larger networks, such as the Internet, could even recover from a loss of nation- or continent-wide segments of the network.

However the claim that SDN would introduce a single point of failure to a network is false in two ways. Firstly, vendors have designed and implemented so-called high availability features to their SDN platforms. High availability, briefly mentioned in this thesis as well, is achieved through parallel SDN controllers which are running the same software. In HP networks, all the network devices can be introduced to multiple SDN controllers, as hinted by the HP switches' SDN configuration. The configuration referred to the controller-id with the value of "1", meaning that a device could know more than one controller. These parallel controllers could reside in completely different physical and network locations, as long as the network devices are able to reach each of the controllers via IP connectivity. If one controller fails, the next one would instantly take control of the situation. The parallelization in e.g. Juniper and Cisco environments is achieved not via introducing multiple controllers to all the network devices, but parallelizing the SDN controllers through BGP federation, making them look from outside perspective like one controller.

On the other hand, a complete failure of the whole set of assigned SDN-controllers for example because of a defective software update, would not completely hamstring the whole network. Because network devices underneath the SDN controller are still switches, routers, firewalls (which are essentially routers as well), they always have to have the basic understanding of network operation and their neighboring devices. This is ensured by the fact that in OpenFlow's two-part structure, where first part consists of matches and the second part of actions, one of the available actions is always outputting the data to a port called "`NORMAL`". The concept of normal forwarding has to have some meaning to devices, meaning that devices have to know what to do with a packet when ordered to forward it in the way that they would, if they weren't a part of an SDN network in the first place. This is an important thing to understand, because while the whole point of SDN is to separate decision-making and so-called intelligence from the network devices to a centralized controller, the network devices will still always have to be able to forward packets to their normal route without any kind of interaction from the SDN controller. In a case of total failure of the whole set of SDN controllers, most of the functionality of the network would be retained. Specially developed and configured behavior of the network via SDN applications would be lost, however. This would cause network issues within datacenters and Intranets, but it would not suppress the basic network connectivity. Network outages due device malfunction are possible with current generation technology, as well.

Another often brought up issue with SDN is related to the performance and scalability. As it was described in the chapter 4, the way SDN is designed to function, a small delay to the beginning of all new network flows is introduced. This delay occurs because the first packet of every unrecognized flow is sent to the SDN controller for inspection, before any packets of that flow are forwarded any further. In testing done during the development phase of this thesis, it was confirmed that only the first packet of any flow is affected by this delay, and the delay can be expressed in a few milliseconds in physical network environments. While this is certainly a step back, the significance of this delay can be considered negligible. SDN's aim was never specifically increasing the performance of networks, but adding fundamental features such as programmability to the network stack and making logical changes to the network architecture. SDN always aimed to act as an enabler for third party development in the networks, not as a performance booster. In some extremely performance-critical environments it might be reasonable to consider dismissing SDN because of its performance impact, but there would have to be only a few cases like these in the world.

The second performance / scalability issue is related to the capabilities of a single SDN controller instance. Because all the inspected SDN controllers are virtualized appliances, meaning that they are deployed as software on computer hardware, a lot depends on the hardware used. Some performance issues can be thus avoided or be dealt with by increasing ever-developing hardware resources for the appliance. Software limitations and scalability does step in at some point, however. The exact limitations depend on the specific SDN controller. In the tests carried out in the virtualized development environment, the HP's solution was observed to be able to control a network of around 1000 network nodes with no perceived performance impact. This was true even when the virtualized network was run on the same physical hardware with the SDN controller (in a different virtual machine, however). OpenDaylight on the other hand started showing symptoms of slowing down when instantly attached to a network consisting of more than a hundred of network nodes. The slowing down was perceived to be very linear. Generally it can be said that the HP platform showed to be more capable than OpenDaylight platform, performance-wise.

However, while both of these issues as partly solvable by increasing the hardware resources, the significance of the issues can be brought to question in the first place. While SDN aims to bring centralized controller to the network, it was never intended to be built with only one controller controlling the whole existing network. The core idea of the SDN architecture was dividing the network into segments and introducing one SDN controller for each of these segments. Only thing this noticed performance difference would affect would be the size of these designed network segments, which could be larger on HP platform and would be smaller on OpenDaylight. This doesn't differ from e.g. WLAN Controller realm at all; each of the controllers has their limitations in

the scope and size of the wireless network they're able to control. If faced with this limitation, the solution would be the deployment of a second WLAN Controller.

Probably the most significant downside to the SDN is divided industry stance on the whole technology. Some vendors have adopted an aggressive SDN strategy and are supporting standards such as OpenFlow, while others seem to be more reluctant with SDN as a whole and are developing their own proprietary southbound standards, like Cisco is doing with OpFlex as opposed to the open standard of OpenFlow. This is not criticism towards SDN per se, but probably the most significant risk it is facing, however.

5.3 Analysis of the created application

As an end-product from the work described in the chapter 4, a usable and a working web application was created, which has the capability of making two kinds of changes to the network: changing a certain network flow's quality of service on the network, as well as changing the path that the data is forwarded within the network topology.

While the web application is clearly developed as a prototype, as can be deducted from e.g. the fact that the amount of work aimed in development of CSS styles was very low, and from the lack of ability of doing larger amount of custom parameterization in the API queries sent to the SDN controller (for instance changing the protocol from IPv4 to IPv6), the main goal of the application was achieved. The goal of the development work was making a working proof of concept SDN application on a self-built SDN environment. Even building the SDN-enabled network and getting it smoothly running had its own uncertainties before the project, granted it was not the point of interest in the thesis.

The development work did not spawn any clearly defined code that could be recommended for a re-use in future projects, but this was not a defined goal of the thesis in the first place. From the point of view of the organization for which the application was developed, the main gain, in addition to proving the functionality of the SDN concept, was gaining the understanding of SDN application development via experimental work done with mainly on one vendor's platform.

The development work itself consisted of limited amount of Python programming, combined with HTML and CSS markup and setting up the programming environment, such as making the choice of web server and Python module used. While the tools chosen (Apache 2.4.7, Python 3 and mod_wsgi 3.5) could even be recommended for development work in the area, the future work made by TeliaSonera will most likely move to Java development. This fact renders all of the program code unusable directly. This was a conscious choice made in the beginning of the thesis project however, allowing agility

in the experimental development work, while understanding that the real, production level SDN applications developed later on would start their work from scratch.

The developed application had its clear limitations. The application was specifically designed for the environment it was run in, and for the specifically designed use cases it managed to fulfill. The created application can be criticized from these aspects. Firstly, even though one of the main benefits of SDN is to allow automation in the network, the created application does not really provide that yet. While the ability of creating self-service tools for both network administrators as well as customers themselves is a valuable thing, with SDN it would be possible to make changes happen in the network not based on a user input, but rather activating the changes via machine-to-machine interaction. This could happen e.g. when a threat is detected in the network by a third-party system, most likely not related to SDN at all. This third party system would then trigger a HTTP call to a specifically developed SDN application, which could provide its own REST API for external use. Integration like this would require flexibility from the envisioned third party application, or even self-made development of that mentioned application. This extended use case was excluded from the goals of the work done for this thesis, however.

Second issue with the developed application has been brought up earlier also. More flexible fields for using the SDN controller's REST API through the form presented for the end-user on the web page would allow more flexible scenarios in which the application could be used in. This was not required from a proof of concept point of view, however. Extending the form and allowing customization of the REST query is just a repetition of the work already done.

Third weakness observed in the developed prototype application is how it only works with a certain network topology. SDN implementations, especially the ones relying on OpenFlow, have a way of finding out the topology of the network. Basically the SDN controller is always aware of the topology of the network at any given time, and keeps track of all the topology changes as they happen. The HP's platform offers methods in its REST API which could tell the application the topology and that topology could be even visually viewed and in most the advanced form, even graphically manipulated in the created SDN application. This prototype does not, however, read the topology from the REST API and does not have any other knowledge of the fact, but the assumption that the five devices in the network, are in a single loop topology.

This means that many of the possibilities offered by the SDN ideology have not been utilized in the prototype application. The purpose of the application was not, however, pointing out the extreme boundaries of SDN as a technology, but rather proving that the concept is technologically feasible and the development work is within the grasp of the TeliaSonera development projects. In this, the prototype application succeeded.

5.3.1 Benefits of SDN application –based network management

The current way of making changes in the network is mainly manual work. Every functional change to the network usually requires keen knowledge of the network in question, careful design of the change chain and making the relevant changes in all the relevant network devices one by one, in an order that does not cause network to malfunction at any point of the workflow. This way of managing networks requires specialists who make the changes, and in real-life organizations work like this tends to pile up as a queue of tickets that need addressing.

This leads to a situation, where even smallest changes to the network could take hours, days or even weeks to complete, due to the fact that the people who are making the changes can be over utilized. There are numerous of use cases, such as multi-tenant cloud environments, where technological advancement in other areas necessitates some faster form of network change management. In other areas being able to do changes faster could simply offer competitive edge over competitors, or vice versa; if competitors adapted the SDN way of managing changes in the network and incumbent operators did not, the competition will pass the incumbents and roles will change.

When changes to the network can be automatized to some degree, and computerized in the rest of the cases, the changes to the network will happen faster. Therefore network management in the style of the created prototype SDN application offers speed: by decreasing the workload and computerizing the work related to functional network changes, more changes can be done in shorter time. When filling up one from on a web page and clicking one button propagates the changes the user wanted to make to the whole network in a secure and effective way via OpenFlow, the difference with the process that involves careful design and manual work, is significant.

The possibility of using the REST API of the SDN controller also adds other possible benefits. SDN-capable networks could change more dynamically, based on some other system detecting something on the network, and triggering an SDN application that could dynamically make changes to the network without any input from a human user. These changes could happen in any conceivable pace, and network could easily become a living, constantly changing entity that does not have some basic state, where it stays, until changes occur.

Computerizing the change work also makes errors more unlikely. Manual process is prone to human errors, while a programmed software will, once introduced to the correct way of making the change, always execute the workflow in same manner, without deviations. Obviously it is still possible to introduce errors, or bugs, to the SDN applications, but once corrected, errors should not spontaneously appear.

5.3.2 Significance of the developed web application

By looking at the currently available applications for instance in the HP's SDN App Store, which include complex applications from specialized companies (such as ECODE Network's Evolve-application, which is a suite of tools to facilitate dynamic network design, provisioning, simulation and automation), it's clear that the developed prototype application does not really have any real novelty value. It's clear that the prototype would not, even if finished and packaged in a useable package, have any reason to enter the SDN App Store, for instance. The other SDN platforms support their own kind of delivery and browsing mechanisms for their application ecosystem. For instance Extreme Networks is using just GitHub as the place to freely share SDN applications, but the general situation is the same on all platforms: more complex applications are already available.

The real value of the prototype is therefore accumulated and gained knowledge on SDN application development by creating the prototype. It could be said that the prototype work opened a new door for TeliaSonera, which could now, if it was willing, develop their own SDN applications, with or without partners in the programming work.

5.4 Controller interchangeability

SDN controllers are in theory interchangeable as long as they support the OpenFlow–specification towards the network devices. The harder issue is solving the compatibility towards third party created SDN applications, which offer the real functionality to the SDN-enabled networks. This compatibility does not currently exist, because lack of standard or any widely available and supported encapsulation frameworks.

In the chapter 4.6 a process for manual conversion of SDN application developed on one platform to work with another SDN controller was described. The process led to a completely identical functionality with both HP's VAN SDN Controller and OpenDaylight controller. However, the conversion involved lots of manual work to the API calls and also some more fundamental changes to the application making the API queries. Therefore vendor lock is an actual problem in the SDN world, which could turn out to be very consequential. Changing SDN platform to another one is possible, but requires time, money and work.

Analysis of the northbound APIs points that the APIs could be considered even surprisingly similar, however. The actual markup of the REST API methods is different, but it is clear that the both APIs are just REST-representations, mark-upped in JSON/XML, of the OpenFlow standard. These APIs do not change the abstraction level, but rather offer every OpenFlow feature as a usable REST resource, while parameterization is handled by accompanied JSON bodies in requests and responses.

The general vendor lock-in -situation is made less severe by the fact that while the SDN controller itself could be seen as relatively fixed choice, the networks which are under it can consist of devices of any vendors, as long as they support the standard southbound interfaces. Therefore a vendor lock-in will not cause a chain reaction, which would force networks operators to buy more of the chosen vendors' devices. A situation like that exists elsewhere, for instance in the WLAN controller world: a WLAN controller only supports WLAN access points manufactured by the same vendor.

5.5 SDN ecosystems

The core idea of the SDN architecture is to allow third party programs to have access to the APIs of the SDN controllers – whatever their technological implementation. This makes it possible to control or observe the network from a third party application. This leads a situation where vendor-specific SDN applications are created, and these applications could then be sold. Therefore small ecosystems will be born around every SDN platform. Vendors have different approaches for the ecosystem–idea, at least initially, though.

HP is aiming to spring up an SDN App Store, much like the ones available for mobile phone operating systems currently (where HP is not a notable operator). The App Store, in HPs vision, is a place where new functionality could be bought and added to the network with ease. The App Store is integrated directly to their SDN controller's UI and while it doesn't have too much content, the number of applications has been increasing steadily. Currently there are applications available in the App Store created by software development companies such as Ecode Networks, F5, Bluecat, GuardiCore, KEMP and RealStatus.

Cisco and Juniper have taken more closed approach, selling applications via their representatives, but not directly via any accessible App Store. They have not taken any steps on forbidding reselling of SDN applications however, meaning that third party App Stores could emerge, or more likely SDN applications for Cisco and Juniper Platforms will be available from some existing application retailers in business-to-business field.

On the other hand, some other SDN vendors have taken more open approach. Extreme Networks uses GitHub as the place for sharing third party SDN applications for their platform, as has done the open source project OpenDaylight. The weakness of this approach is the lack of centralized quality assurance and difficulties in finding the applications necessary for each of the use cases. OpenDaylight and Extreme Networks have not yet started monetizing SDN applications either, which is an issue for parties who are looking at developing SDN applications for money.

TeliaSonera's core business would however not be trying to act as software developer for third party needs, and trying to resell the self-generated SDN applications for competitors. This would cause all the competitive edge caused by supposedly well-created and well-received SDN applications to disappear, when TeliaSonera's local and global competition could instantly acquire the same functionality to their SDN enabled networks – supposing they made the same technological choices for SDN platforms as TeliaSonera had done. Most likely from TeliaSonera's perspective the creation of very specialized applications for own use only is more profitable both economically and from corporate image-perspective, than trying monetizing revenue from well-created SDN applications.

There is no reason, however, why ecosystems around SDN applications couldn't boom to be significant business. This is mostly dependent on SDN itself – if it revolutionizes networking, SDN applications will turn out to being good business. If it's just an incremental and slowly adopted step in reaching next generation networking, which would be something beyond SDN, the business case around SDN application development is much bleaker.

5.6 SDN roadmap

The work done for this thesis painted a clear picture of SDN in few angles. It is clear that while SDN already is technologically working and available as purchasable products, and the rest of the vendors are accelerating their entry to the market, SDN is still raw in many ways. Work developing and specifying the core technology will still continue.

From TeliaSonera's perspective taking part in this work could turn out to be crucial. TeliaSonera has a global presence large enough, to warrant participation in forums and discussions where future roadmaps of SDN and related technologies are drawn between overseeing entities like ONF, vendors and globally prominent network operators. While TeliaSonera is for instance number two in IP transit worldwide and owner of one of only thirteen Tier 1 networks in the world, it is still not participating in international standardization and development work like its international competitors.

In addition to northbound API standardization or at least guidelines, the integration of SDN-controlled networks to the operations support systems (OSS) and business support systems (BSS) telecom companies are currently running is still largely not done. Integration to these systems can be achieved through SDN applications relying on the northbound API of the SDN stack, but in this case the OSS/BSS integrations will be scattered among all the SDN applications running on a certain controller. More centralized and technically more lower-level integration would be a huge competitive benefit for the SDN platform that decided to implement it first.

Operations support systems and business support systems integration could, as a middle ground, be implemented via a single SDN application, which would then provide formally or de facto standardized APIs for its functionality in form or another for all the other applications running on the SDN platform. This would be feasible to implement even today, but the problem would be getting third-party developed applications, for instance Ecode Networks' or F5's SDN applications on the HP SDN VAN Controller to support this newly-created OSS/BSS –integration. Therefore this road could only be taken as a joint and coordinated effort between all the major developers in the SDN area. Partial solutions are possible for self-made applications, however.

6. CONCLUSIONS

In this thesis, a proof of concept was built and validated on two different SDN environments, using variable underlying network combinations and topologies, built both of stacked hardware as well as virtualized networks. The prototype that was used to prove the concept feasible and technologically mature enough was internally demonstrated to TeliaSonera representatives in a live demo.

The work and analysis pointed out that Software-defined networks are providing benefits and possibilities never seen before in the networking realm by adding programmability to the networks. Programmability makes it possible to manage changes in the network via simplified self-service tools, which could change the abstraction level of network changes to a level that would be understandable even to customer's IT department or management, but it could as well be utilized in creating more effective and faster tools for experts working currently in network operations center (NOC). While SDN is already utilized in certain environments in the wild, especially within datacenters via tools such as OpenStack and CloudStack, the ultimate goal of SDN and programmability of the networks is make networks able to react to any kind of triggers from third party systems with speed and pace that manual work would never achieve. Machine-to-machine communication between e.g. network analyzers and intrusion prevention and detection systems could automatize network changes to occur without human interaction and eventually make networks constantly changing organisms, without any clear basic state in which they reside.

SDN solutions in the market from multiple vendors are ready to be moved from internal laboratory testing to limited development and deployment work with pilot customers. Most progressed SDN platforms currently are HP's VAN SDN Controller and the open source project OpenDaylight, as well as all the other solutions technically basing themselves in the OpenDaylight codebase, such as the Cisco Systems' and Extreme Networks' solutions. The next step, working with real-life environments and formally developing and deploying SDN solutions with some of these platforms would give more answers on SDN's real-life reliability and scalability and will validate the possibilities of SDN as a technology when faced with limitations often found in legacy environments, such as old network devices and device software and hard-to-reach network segments. Working with customers directly would also give TeliaSonera important insight on if projected use cases for SDN are what customers are looking for, or if the use cases need to be redefined to better match the needs of the customers.

It is currently impossible to determine if SDN will revolutionize networking in the way its proponents are expecting, but embracing it will keep the network operator in the front lines of technological advancement in its field, and could turn out to providing the crucial competitive edge over local competition within very short timeframe of one to two years. No clear or ambiguous technical roadblocks have been identified in hands on development with prevalent SDN platforms, but the boundaries where SDN currently is able to reach, are still unclear. SDN should definitely be embraced and work on it should be continued without interruptions.

The main conclusion made by this thesis is the fact that SDN is working technology, which creates many possibilities for network operators, as well as other actors such as IT companies. SDN is definitely still in its introductory phase on the product life cycle curve. This means that larger economic gains from increasing sales it are still ahead during the growth and maturity phases. As a recommendation, SDN should be worked with in order to achieve competitive benefits in the networking business, as the technology is ready for customer level deployments, after the required knowledge and understanding of it within the organization is achieved.

REFERENCES

Atif, A. & Scott, M. (2007). An Introduction to JavaScript Object Notation (JSON) in JavaScript and .NET. Online. Available (accessed on 26.9.2014): http://msdn.microsoft.com/en-us/library/bb299886.aspx

Big Switch Networks, Inc. (2014). Big Switch Networks: Big Network Controller. Online. Available (accessed on 26.9.2014): http://www.bigswitch.com/products/SDN-Controller

Black, C. & Goransson, P. (2014). Software Defined Networks, 1st ed. Elsevier Science. Chapter 7.

Bort, J. (2013). Business Insider: EXCLUSIVE: Here's What Happened When Cisco Lost A $1 Billion Deal With Amazon. Online. Available (accessed on 26.9.2014): http://www.businessinsider.com/source-cisco-1b-amazon-deal-led-to-insieme-sdn-2013-10

Bray, T. et al. (2008). Extensible Markup Language (XML). Online. Available (accessed on 26.9.2014): http://www.w3.org/TR/2008/REC-xml-20081126/#sec-origin-goals

Butler, B. (2013). ChannelWorld.in: VMware Blends In Nicira SDN Technology, reveals public cloud plans. Online. Available (accessed on 26.9.2014): http://www.channelworld.in/news/vmware-blends-nicira-sdn-technology-reveals-public-cloud-plans-376232013

Cisco Systems, Inc. (2013). The Cisco Application Policy Infrastructure White Paper. Online. Available (accessed on 26.9.2014): http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/unified-fabric/white-paper-c11-730021.pdf

Cisco Systems, Inc. (2014). Cisco Extends Application-centric Infrastructure (ACI) to Access and Wide Area Networks, Increasing Network Automation and IT Agility. Online. Available (accessed on 26.9.2014): http://newsroom.cisco.com/release/1332017/Cisco-Extends-Application-centric-Infrastructure-ACI-to-Access-and-Wide-Area-Networks-Increasing-Network-Automation-and-IT-Agility

Clark, J. (2014). The Register: Google exposes its Andromeda software-defined networking. Online. Available (accessed on 26.9.2014): http://www.theregister.co.uk/2014/04/03/google_andromeda_cloud/

Duffy, J. (2010). NetworkWorld: Cisco's top 10 rivals - Cisco battling Juniper, IBM, HP and more across the enterprise network market. Online. Available (accessed on 26.9.2014): http://www.networkworld.com/article/2191771/data-center/cisco-s-top-10-rivals.html

Duffy, J. (2014). NetworkWorld: Cisco reveals OpenFlow SDN killer. Online. Available (accessed on 26.9.2014): http://www.networkworld.com/news/2014/040214-cisco-openflow-280282.html

Ferro, G. (2012). Ethereal Mind: Northbound API, Southbound API, East/North – LAN Navigation in an OpenFlow World and an SDN Compass. Online. Available (accessed on 26.9.2014): http://etherealmind.com/northbound-api-southbound-api-eastnorth-lan-navigation-in-an-openflow-world-and-an-sdn-compass/

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Online. Available (accessed on 26.9.2014): http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Google Inc. (2014a). Google Finance Beta, Juniper Networks, Inc.. Online. Available (accessed on 26.9.2014): https://www.google.com/finance?q=NYSE:JNPR&fstype=ii&ei=HvxBU4DxBMSvqQGMxAE

Google Inc. (2014b). Google Finance Beta: Hewlett-Packard Company (NYSE:HPQ). Online. Available (accessed on 26.9.2014): https://www.google.com/finance?q=NYSE:HPQ&fstype=ii&ei=qjb6UIjWGpHYkQX0iQE

Gourlay, D. (2014). Arista: Making SDN a Reality. Online. Available (accessed on 26.9.2014): http://www.bradreese.com/blog/4-1-2013.pdf

Hesseldahl, A. (2014). Re/code: IBM Exploring Sale of Software-Defined Networking Business. Online. Available (accessed on 26.9.2014): http://recode.net/2014/01/28/ibm-exploring-sale-of-software-defined-networking-business/

Hewlett-Packard Development Company, L.P. (2013a). HP Virtual Application Networks SDN Controller. Online. Available (accessed on 26.9.2014): http://h20195.www2.hp.com/V2/GetPDF.aspx/4AA4-8807ENW.pdf

Hewlett-Packard Development Company, L.P. (2013b). HP OpenFlow and SDN Technical Overview Technical Solution Guide. Online. Available (accessed on 26.9.2014): http://h17007.www1.hp.com/docs/networking/solutions/sdn/devcenter/02_-_HP_OpenFlow_and_SDN_Technical_Overview_TSG_v1_2013-10-01.pdf

Hewlett-Packard Development Company, L.P. (2013c). HP SDN Controller Architecture. Online. Available (accessed on 26.9.2014): http://h17007.www1.hp.com/docs/networking/solutions/sdn/devcenter/HP_SDN_ControllerArchitectureTSGv1_3013-10-01.pdf

Hewlett-Packard Development Company, L.P. (2013d). HP SDN REST API and Security. Online. Available (accessed on 26.9.2014): http://h17007.www1.hp.com/docs/networking/solutions/sdn/devcenter/09_-_HP_SDN_REST_API_and_Security_TCG_v1_3013-10-01.pdf

Hewlett-Packard Development Company, L.P. (2014a). OpenFlow: Enabling technology for software-defined networking. Online. Available (accessed on 26.9.2014): http://h17007.www1.hp.com/fi/en/solutions/technology/openflow/index.aspx

Hewlett-Packard Development Company, L.P. (2014b). HP VAN SDN Controller 2.2. REST API. Online. Available (accessed on 26.9.2014): http://h20564.www2.hp.com/portal/site/hpsc/public/kb/docDisplay/?docId=c04003972

Hewlett-Packard Development Company, L.P. (2014c). SDN Dev Center: SDN Developer Kit (SDK). Online. Available (accessed on 26.9.2014): http://h17007.www1.hp.com/us/en/networking/solutions/technology/sdn/devcenter/index.aspx#tab=TAB2

IBM Corporation (2014a). IBM: IBM Programmable Network Controller. Online. Available (accessed on 26.9.2014): http://www-03.ibm.com/systems/networking/software/pnc/specs.html

IBM Corporation (2014b). IBM Software Defined Network for Virtual Environments - Thought Leadership White Paper. Online. Available (accessed on 26.9.2014): http://public.dhe.ibm.com/common/ssi/ecm/en/qcw03028usen/QCW03028USEN.PDF

Juniper Networks, Inc. (2013). REST API Details. Online. Available (accessed on 26.9.2014): http://juniper.github.io/contrail-vnc/api-doc/html/rest_details.html

Juniper Networks, Inc. (2014a). Contrail Overview. Online. Available (accessed on 26.9.2014): http://www.juniper.net/us/en/products-services/sdn/contrail/

Juniper Networks, Inc. (2014b). OpenContrail Network Virtualization Architecture - Deep Dive. Online. Available (accessed on 26.9.2014): http://opencontrail.org/network-virtualization-architecture-deep-dive/

Kerner, S. M. (2013). Enterprise Networking Planet: OpenFlow Inventor Martin Casado on SDN, VMware, and Software Defined Networking Hype [VIDEO]. Online. Available (accessed on 26.9.2014):

http://www.enterprisenetworkingplanet.com/netsp/openflow-inventor-martin-casado-sdn-vmware-software-defined-networking-video.html

Kerravala, Z. (2014). Cisco APIC Enterprise Module Simplifies Network Operations. Online. Available (accessed on 26.9.2014): http://www.cisco.com/c/dam/en/us/products/collateral/cloud-systems-management/application-policy-infrastructure-controller-apic/white-paper-c11-730846.pdf

Khondoker, R., Zaalouk, A., Marx, R. & Bayarou, K. (2014). Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers. Online. Available (accessed on 26.9.2014): http://sit.sit.fraunhofer.de/mne/publications/download/SDNControllers.pdf

Khosravi, H. & Anderson, T. (2003). Requirements for Separation of IP Control and Forwarding. Online. Available (accessed on 26.9.2014): https://datatracker.ietf.org/doc/rfc3654/

Little, R. G. (2014). SearchSDN: What the OpenDaylight controller will do for IBM SDN. Online. Available (accessed on 26.9.2014): http://searchsdn.techtarget.com/news/2240214050/What-the-OpenDaylight-controller-will-do-for-IBM-SDN

Matsumoto, C. (2013). SDN Central: Is Cisco's SDN Architecture Really That Special?. Online. Available (accessed on 26.9.2014): http://www.sdncentral.com/news/is-cisco-sdn-architecture-really-that-special/2013/11/

McCaffrey, J. (2005). MSDN Magazine: Test Run: The Analytic Hierarchy Process. Online. Available (accessed on 26.9.2014): http://msdn.microsoft.com/en-us/magazine/cc163785.aspx

Morgan, T. P. (2012). The Register: HP previews OpenFlow virty network controller. Online. Available (accessed on 26.9.2014): http://www.theregister.co.uk/2012/10/03/hp_sdn_openflow_controller/

Morgan, T. P. (2013). Juniper open sources Contrail SDN software stack. Online. Available (accessed on 26.9.2014): http://www.theregister.co.uk/2013/09/16/juniper_contrail_sdn_controller_ships/

Morgan, T. P. (2014a). EnterpriseTech: Google Lifts Veil On "Andromeda" Virtual Networking. Online. Available (accessed on 26.9.2014): http://www.enterprisetech.com/2014/04/02/google-lifts-veil-andromeda-virtual-networking/

Morgan, T. P. (2014b). EnterpriseTech Networks Edition: Cisco Counters OpenFlow SDN With OpFlex, Updates Nexus Switches. Online. Available (accessed on 26.9.2014): http://www.enterprisetech.com/2014/04/04/cisco-counters-openflow-sdn-opflex-updates-nexus-switches/

Murray, A. C. (2012). InformationWeek: NetworkComputing: Plexxi Cuts a New Path to SDN. Online. Available (accessed on 26.9.2014): http://www.networkcomputing.com/networking/plexxi-cuts-a-new-path-to-sdn/d/d-id/1234003?

NOXRepo.org (2014). NOX: About NOX. Online. Available (accessed on 26.9.2014): http://www.noxrepo.org/nox/about-nox/

Open Networking Foundation (2009). OpenFlow Switch Specification. Online. Available (accessed on 26.9.2014): https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf

Open Networking Foundation (2011a). OpenFlow Switch Specification Version 1.1.0 Implemented (Wire Protocol 0x02). Online. Available (accessed on 26.9.2014): https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf

Open Networking Foundation (2011b). OpenFlow Switch Specification Version 1.2 (Wire Protocol 0x03). Online. Available (accessed on 26.9.2014): https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf

Open Networking Foundation (2012a). ONF White Paper: Software-Defined Networking: The New Norm for Networks. Online. Available (accessed on 26.9.2014): https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

Open Networking Foundation (2012b). OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04). Online. Available (accessed on 26.9.2014): https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

Open Networking Foundation (2014a). ONF Specifications. Online. Available (accessed on 26.9.2014): https://www.opennetworking.org/sdn-resources/onf-specifications

Open Networking Foundation (2014b). Software-Defined Networking (SDN) Definition. Online. Available (accessed on 26.9.2014): https://www.opennetworking.org/sdn-resources/sdn-definition

Open Networking Foundation (2014c). SDN Product Directory. Online. Available (accessed on 26.9.2014): https://www.opennetworking.org/sdn-resources/onf-products-listing

Palmer, M. (2013). SDNCentral Exclusive: SDN Market Size Expected to Reach $35B by 2018. Online. Available (accessed on 26.9.2014): https://www.sdncentral.com/market/sdn-market-sizing/2013/04/

Paraskevopoulos, K. (2014). International Hellenic University: Lesson 1: The Analytic Hierarchy Process (AHP). Online. Available (accessed on 26.9.2014): http://rad.ihu.edu.gr/fileadmin/labsfiles/decision_support_systems/lessons/ahp/AHP_Lesson_1.pdf

Project Floodlight (2012). ProjectFloodlight.org Wiki: The Controller. Online. Available (accessed on 26.9.2014): http://docs.projectfloodlight.org/display/floodlightcontroller/The+Controller

Project Floodlight (2013). FAQ Floodlight OpenFlow Controller. Online. Available (accessed on 26.9.2014): http://docs.projectfloodlight.org/display/floodlightcontroller/FAQ+Floodlight+OpenFlow+Controller

Project Floodlight (2014a). Project Floodlight: Floodlight. Online. Available (accessed on 26.9.2014): http://www.projectfloodlight.org/floodlight/

Project Floodlight (2014b). Project Floodlight: Organizations. Online. Available (accessed on 26.9.2014): http://www.projectfloodlight.org/organizations/

Ramel, D. (2014). Virtualization Review: Cisco Offers OpFlex as OpenFlow Alternative. Online. Available (accessed on 26.9.2014): http://virtualizationreview.com/articles/2014/04/07/cisco-opflex.aspx

Salisbury, B. (2012). NetworkStatic: SDN Use Cases for Service Providers. Online. Available (accessed on 26.9.2014): http://networkstatic.net/sdn-use-cases-for-service-providers/

Sherry, J. (2014). Grand View Research: Global Software Defined Networking (SDN) Market By End Users (Enterprises, Cloud Service Providers, Telecommunications Service Providers), By Solutions (Switching, Controllers) Expected To Reach USD 4,909.8 Million By 2020. Online. Available (accessed on 26.9.2014):

http://www.grandviewresearch.com/press-release/global-software-defined-networking-sdn-market

Singla, A. & Rijsman, B. (2014). OpenContrail Architecture Documentation. Online. Available (accessed on 26.9.2014): http://opencontrail.org/opencontrail-architecture-documentation/

Standord University (2006). 6.) A Clean Slate Approach to Enterprise Network Security: Ethane. Online. Available (accessed on 26.9.2014): http://cleanslate.stanford.edu/research_project_ethane.php

Stanford University (2012). Clean Slate Program. Online. Available (accessed on 26.9.2014): http://cleanslate.stanford.edu/

Sundararajan, R. (2012). TRILL & Datacenter technologies - their importance, and alternatives to datacenter network convergence. Online. Available (accessed on 26.9.2014): http://www.slideshare.net/Aricent/trill-and-datacenter-alternatives

TechTarget (2014). Do SDN northbound APIs need standards?. Online. Available (accessed on 26.9.2014): http://searchnetworking.techtarget.com/feature/Do-SDN-northbound-APIs-need-standards

TeliaSonera Finland Oyj (2014). Älykäs Verkko lisää merkittävästi verkon tuottamaa palveluarvoa. Online. Available (accessed on 26.9.2014): http://www.sonera.fi/yrityksille/yritysratkaisut/avaimet+menestykseen/alykas+verkko

The OpenDaylight Project, Inc. (2014a). OpenDaylight: Technical Overview. Online. Available (accessed on 26.9.2014): http://www.opendaylight.org/project/technical-overview

The OpenDaylight Project, Inc. (2014b). OpenDaylight Wiki: OpenDaylight Controller:Architectural Principles. Online. Available (accessed on 26.9.2014): https://wiki.opendaylight.org/view/OpenDaylight_Controller:Architectural_Principles#Open_Extensible_Northbound_API

U.S. Securities and Exchange Commission (2013). U.S. Securities and Exchange Commission: Filing Detail, Form 10-K, Cisco Systems, Inc. (Filer) CIK: 0000858877. Online. Available (accessed on 26.9.2014): http://www.sec.gov/Archives/edgar/data/858877/000085887713000049/0000858877-13-000049-index.htm

Wagner, M. (2014). LightReading: Juniper Gives OpenDaylight Some Loving. Online. Available (accessed on 26.9.2014): http://www.lightreading.com/juniper-gives-opendaylight-some-loving/d/d-id/708705

Wang, K.-C. (2012). Floodlight Controller: Supported Topologies. Online. Available (accessed on 26.9.2014): http://www.openflowhub.org/display/floodlightcontroller/Supported+Topologies

Viellechner, O. & Wulf, T. (2010). Incumbent Inertia Upon Disruptive Change in the Airline Industry: Causal Factors for Routine Rigidity and Top Management Moderators. Online. Available (accessed on 26.9.2014): http://www.hhl.de/fileadmin/texte/publikationen/arbeitspapiere/hhlap0100.pdf

Vissichio, S., Vanbever, L. & Bonaventure, O. (2014). Opportunities and Research Challenges of Hybrid Software Defined Networks. Online. Available (accessed on 26.9.2014): http://www.ietf.org/proceedings/89/slides/slides-89-sdnrg-0.pdf

Yegulalp, S. (2013). InformationWeek's Network Computing: Five SDN Benefits Enterprises Should Consider. Online. Available (accessed on 26.9.2014): http://www.networkcomputing.com/networking/five-sdn-benefits-enterprises-should-consider/a/d-id/1234292?

Zilinskas, R., Fujii, E., Dalnoki-Veress, F. & al., e. (2012). The Analytical Hierarchy Process: A New Tool for Complex Decision-Making in Public Health Preparedness: Workshop title. Online. Available (accessed on 26.9.2014): http://medepi.files.wordpress.com/2012/02/phprep-summit-2012-ahp2.pdf

Zmijewski, E. (2014). Renesys: A Baker’s Dozen (2013) Edition. Online. Available (accessed on 26.9.2014): http://www.renesys.com/2014/01/bakers-dozen-2013-edition/