MSc. Thesis

A Qualitative Study of SDN Controllers

Dimitra Sakellaropoulou

Supervisor: George Xylomenos

Athens, September 2017

Abstract

Software Defined Networking (SDN) is a recent network architecture paradigm based on the separation between the data plane and the control plane, which handles network traffic by means of software. This thesis focuses on SDN ecosystem's main component, the SDN controller.

The growth of SDN in terms of functionalities and applications has led to the development of different controller solutions, with a wide variety of characteristics. The goal of this thesis is to provide a qualitative analysis of existing implementations in terms of features and capabilities.

In the first chapter, the existing network architecture and its restrictions are briefly described. In the second chapter, an introduction to SDN concept is given. Specifically, the background, architecture, applications, use cases and challenges of SDN are presented. Then, the main area of interest, the SDN controller is analyzed. In the third chapter, its history, types, capabilities and main components are described. In chapter four, the most popular open-source controllers are analyzed and compared in terms of their architecture components and basic features. Finally, the controller's efficiency and its main characteristics as well as relevant studies are presented.

Περίληψη

Η τεχνολογία SDN είναι ένα πρόσφατο παράδειγμα αρχιτεκτονικής δικτύου που βασίζεται στο διαχωρισμό μεταξύ του επιπέδου δεδομένων και του επιπέδου ελέγχου για τη διαχείριση του δικτύου κίνησης μέσω λογισμικού. Η συγκεκριμένη εργασία επικεντρώνεται στο βασικό στοιχείο του δικτύου SDN, τον ελεγκτή.

Η ανάπτυξη του SDN όσον αφορά τις λειτουργικότητες και τις εφαρμογές οδήγησε στην ανάπτυξη διαφορετικών ελεγκτών οι οποίοι χαρακτηρίζονται από διαφορετικές αρχιτεκτονικές και ιδιότητες. Στόχος αυτής της εργασίας είναι η ποιοτική ανάλυση των υπαρχουσών υλοποιήσεων από πλευράς χαρακτηριστικών και δυνατοτήτων.

Στο πρώτο κεφάλαιο περιγράφεται συνοπτικά η υπάρχουσα αρχιτεκτονική δικτύων και οι περιορισμοί της. Στο δεύτερο κεφάλαιο δίνεται μια εισαγωγή στην έννοια του SDN. Πιο συγκεκριμένα, περιγράφονται το υπόβαθρο, η αρχιτεκτονική, οι εφαρμογές, οι περιπτώσεις χρήσης και οι προκλήσεις στον συγκεκριμένο τομέα. Στη συνέχεια αναλύεται το κύριο θέμα της εργασίας, ο ελεγκτής SDN. Στο τρίτο κεφάλαιο περιγράφεται η ιστορία, οι τύποι, οι δυνατότητές του και τα κύρια συστατικά του. Στο τέταρτο κεφάλαιο, περιγράφονται οι πιο δημοφιλείς ελεγκτές ανοικτού κώδικα και συγκρίνονται με βάση την αρχιτεκτονική και τα χαρακτηριστικά τους. Τέλος, το επόμενο κεφάλαιο εστιάζει στην αποτελεσματικότητα του ελεγκτή και παρουσιάζει τα στοιχεία που την καθορίζουν καθώς και σχετικές μελέτες που έχουν γίνει τα τελευταία χρόνια.

Contents

List of Tables

List of Figures

List of Acronyms

Acronym	Description
SDN	Software Defined Networks/Networking
ACL	Access Control List
VLAN	Virtual Local Area Network
QoS	Quality of Service
NFV	Network Functions Virtualization
ForCes	Forwarding and Control Element Separation
ONF	Open Network Foundation
IETF	Internet Engineering Task Force
API	Application Programming Interface
OSI	Open Systems Interconnection
NetConf	Network Configuration Protocol
IXP	Internet Exchange Points
VRF	Virtual Rooting and Forwarding
VM	Virtual Machine
TLS	Transport Layer Security
PCE	Path Computation Element
MPLS-TP	Multiprotocol Label Switching - Transport Profile Protocol
XMPP	Extensible Messaging and Presence Protocol
LLDP	Link Layer Discovery Protocol
ODL	OpenDayLight
ONOS	Open Networking Operating System
DIFANE	Distributed Flow Architecture for Networked Enterprises
RTT	Round Trip Time
NRO	Network Resources Optimization
OPNFV	Open Platform for NFV
IoT	Internet of Things
OSGi	Open Service Gateway Initiative
GPLV2	GNU General Public License version 2
On.Lab	Open Networking Lab
I/O	Input/Output

1. Introduction

The explosion of mobile devices and content, as well as virtualization and the emergence of cloud services have led network industry to reconsider traditional architectures. Data centers have changed significantly. In contrast to client-server applications, where most of the communication takes place between the client and the server, today's applications access different databases and servers, creating an east-west machine-to-machine traffic. Moreover, network traffic patterns are changing as users, access content from any type of device. Data centers have evolved in recent years, constantly attempting to meet increasingly higher and rapidly changing demands. Thus, data center infrastructure becomes more complex as additional traffic across wide area networks is added. Since more mobile devices such as smart devices, tablets, notebooks are used to access the network, it is required not just to serve traffic, but also to ensure quality at all aspects of communication (i.e. speed, security). Furthermore, the adoption of public and private cloud services, has also resulted in an unprecedented growth in these services. Finally, big data handling requires mass parallel processing on thousands of connected servers. The increase in the amount of data requires additional network capacity in data centers. Flexibility, high availability, scalability and security are key quality attributes that should be ensured in all these use cases.

Based on the above, it can be concluded that current network requirements cannot be covered using traditional network architecture. The existing architecture is not designed to meet the requirements of users, companies and providers in terms of traffic, availability and scalability for many reasons. Network communication constitutes different sets of protocols, which are designed to connect the nodes over distances with different speed, topology, and service requirements. This results in one of the main constraints of today's networks. To add or remove a device, multiple switches, routers, firewalls, network authentication portals and other infrastructure needs to be set up. Moreover, Access Control Lists (ACLs), Virtual Local Area Networks (VLANs), Quality of Service (QoS), and other protocol-related parameters need to be updated. The same applies in case of any configuration change or update that needs to be applied directly to each device. In addition, network topology, switch manufacturer and software version must be considered.

For all the above reasons, existing networks are considered relatively static. On the other hand, server virtualization has significantly increased the number of nodes that require network connectivity. Applications are now distributed along virtual machines, which share traffic flows with others. In addition to that, many companies use a network that converges to IP for voice, data, and video traffic. While existing networks can provide differentiated levels of service quality for different applications, the provisioning of such resources is manual. The network is unable to dynamically adapt to changes in application traffic and user requirements.

To overcome the existing architecture's restrictions, the Software Defined Networking or Networks (SDN) concept was introduced. SDN aims at changing the way networks are designed and managed and over the past years it has gained significant traction in industry. This thesis elaborates on the SDN concept and specifically to the main build building block, SDN controllers.

2. Software Defined Networks

This chapter is an overview of SDN. A brief description of the SDN concept and its history are provided. The transition from traditional architecture and SDN's main building blocks are presented. Finally, most popular applications and use cases are described to point out SDN's importance in accordance with today's network architecture. Finally, since SDN is under continuous development and research, concepts that need to be further studied are listed.

2.1 The SDN Concept

One of the main features that SDN focuses on, is the separation of control and data plane. These are the basic components of any network architecture. Control plane refers to the logic of controlling and forwarding behavior. Its main functionalities include: tracking topology changes, installing forwarding rules, computing routes, service provisioning etc. Another basic component is the management plane which is sometimes considered as a subset of the control plane. In general, management plane refers to the functionalities responsible for configuring, monitoring, and providing management services to all layers of the network stack and other parts of the system. Data plane (also known as forwarding or user plane) on the other hand refers to the network part that forward user traffic. Forwarding is based on rules as set by the control plane. Other functionalities related to data plane are filtering, buffering, packet measurement etc.

In the case of SDN, the control plane is centralized, controls a distributed data plane and can be implemented completely in software and installed on hardware. Therefore, a SDN may be characterized as a programmable network. The idea of programmable networks has been doing the rounds for many years. It refers to networks in which the behavior of network devices is handled by software. Decoupling the control plane from the data plane makes the control plane programmable, thereby enabling abstraction of the underlying network devices from the application and service layers, which in turn treats them as a virtual entity. Besides the network abstraction, the SDN architecture provides a set of Application Programing Interfaces (APIs) that simplify the implementation of common network services (for example, routing, multicast, security,

access control, bandwidth management, traffic engineering, QoS, energy efficiency, and various forms of policy management).

This separation provides a more flexible, programmable, vendor-agnostic, cost-efficient and innovative network architecture [13]. SDN is one way to solve some problems of the Internet including security, managing complexity, multi-casting, load balancing, and energy efficiency. The separation of the forwarding hardware from the control logic allows easier deployment of new protocols and applications, straightforward network visualization and management, and consolidation of various middle boxes into software control [10]. Instead of enforcing policies and running protocols on different devices, the network is simplified and reduced to forwarding hardware devices and the network controllers.

There are currently many organizations that are focusing part of their research on SDN standardization. For example, the Open Network Foundation (ONF) focuses on OpenFlow protocol standardization. The IETF's Forwarding and Control Element Separation (ForCES) Working Group has been working on standardizing mechanisms, interfaces, and protocols aiming at the centralization of network control and abstraction of network infrastructure. Some of the Study Groups (SGs) of ITU's Telecommunication Standardization Sector (ITU-T) are currently working in several areas related to SDN such cloud computing, mobile and next generation networks, protocols and test specifications. Finally, the Software- Defined Networking Research Group (SDNRG) at IRTF has also focused on SDN under various perspectives

2.2 History

The concept of SDN has been driven by the desire to provide user-controlled management of forwarding in network nodes. It is worth mentioning that the idea of programmable networks and the separation of control plane and data plane has been around for many years. In this section, an overview of earlier programmable networking efforts is provided.

The Open Signaling (OPENSIG)

This working group began in 1995 with a series of workshops dedicated to "making ATM, Internet and mobile networks more open, extensible, and programmable". The main idea was that a separation between the communication hardware and control

software was necessary and would bring a lot of advantages. The core of its proposal was to provide access to the network hardware via open, programmable network interfaces.

General Switch Management Protocol (GSMP)

This Internet Engineering Task Force (IETF) working group was responsible for the specification of the GSMP protocol that describes how a switch should be controlled. GSMP allows a controller to establish and release connections across the switch, add and delete multicast connections, manage switch ports, request configuration information, request and delete reservation of switch resource and others [10]. The working group was officially concluded and its latest standards proposal, (GSMPv3), was published in June 2002.

Active Networking

A programmable network infrastructure for customized services was proposed by the Defense Advanced Research Projects Agency (DARPA) in the mid-90s. It focused on two main approaches: user-programmable switches and capsules. Capsules referred to program fragments that would be carried in user messages and could be interpreted and executed by routers. Despite considerable activity, Active Networking never gathered critical mass or widespread use and industry deployment, mainly due to practical security and performance concerns.

Tempest

Tempest was a framework for safe, programmable networks, introduced in 1998. The Tempest framework provides a programmable network environment by allowing the introduction and modification of network services at two levels. The Tempest framework also allows refinement of services at a finer level of granularity by means of the connection closure concept [64]. In this case, modification of services can be performed at an application-specific level. These attributes of the Tempest framework allowed service providers to effectively become network operators for some well-defined partition of the physical network. This enabled them to take advantage of the knowledge they possess about how the network resources are to be used, by programming their own specially tailored control architecture.

Path Computation Element (PCE)

The Path Computation Element Protocol (IETF standard) that was introduced in 2004 is a control protocol that works in MPLS networks, and partially removes the responsibility from routers to define network paths. The PCE architecture, defined in RFC 4655 (2006), simplifies path computation by separating network topology determination from path creation.

The 4D project

Published in 2004, the 4D Project emphasized the separation among the routing decision logic protocols. The "decision" plane would have a global view of the network which would call upon the services of the "dissemination" and "discovery" planes, for controlling a "data" plane for forwarding traffic [10]. Discovery refers to information about what resources are available to network controller. Dissemination refers to how to detect network topology. Later works like NOX were inspired from these ideas, which proposed an operating system for networks in the context of an OpenFlow enabled network. The latest proposed standard was published in June, 2011

Network Configuration Protocol (NetConf)

In 2006, the IETF Network Configuration Working Group proposed NetConf as a management protocol for modifying the configuration of network devices. The protocol allowed network devices to expose an API through which extensible configuration data could be sent and retrieved. A network with NetConf should not be regarded as fully programmable, as any new functionality would have to be implemented at both the network device and the manager. The NetConf working group is currently active.

Forwarding and Control Element Separation (ForCES)

This working group led a parallel approach to SDN. With ForCES, the internal network device architecture is redefined as the control element is separated from the forwarding element, but the combined entity is still represented as a single network element to the outside world [13]. The ForCES Network Element is separated into forwarding elements and control elements, whereas the ForCES protocol is used to communicate between the two. In contrast to the SDN architecture, this approach still presents the combined entity (forwarding and control elements) as a single network element to the outside world. ForCES concluded in 2015.

Ethane

Ethane is a security management architecture combining simple flow-based switches with a central controller managing admittance and routing of flows. Introduced in 2007, it most closely resembles the SDN architecture. In fact, Ethane laid the foundation stone for SDN [15]. It proposed a centralized controller to manage policy and security in a network. The controller element was to decide the policy for packet handling and an Ethane switch consisting of a flow table and a secure channel to the controller was to implement it.

Open Networking Foundation (ONF)

ONF is a user-driven organization dedicated to the promotion and adoption of SDN, and implementing SDN through open standards where such standards are necessary to move the networking industry forward. ONF is developing open standards such as the OpenFlow Standard and the OpenFlow Configuration and Management Protocol Standard. The OpenFlow Standard is the first vendor-neutral standard communications interface defined between the control and forwarding layers of an SDN architecture. ONF working groups are also collaborating with the world's leading experts on SDN and OpenFlow regarding SDN concepts, frameworks, architecture, and standards.

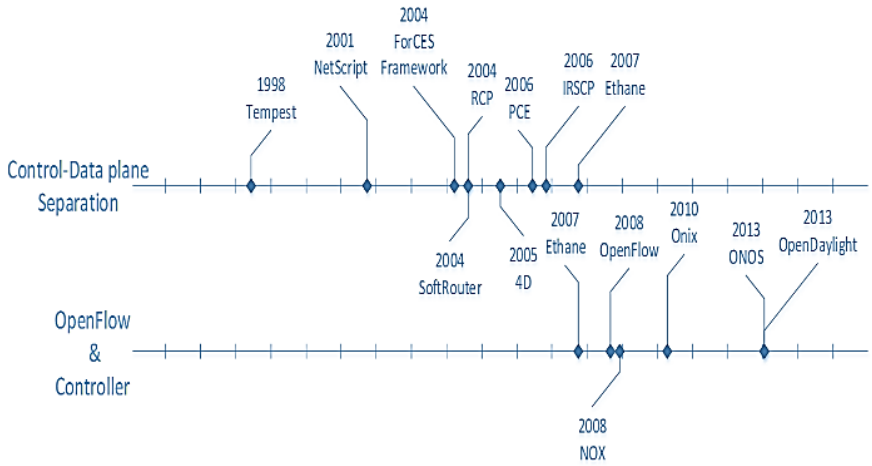

Figure 1: SDN Timeline

2.3 Architecture

As already mentioned, the main aim of the SDN architecture is to achieve the separation of the control and data plane that leads to a more centralized network. Moreover, SDN supports open interfaces between the devices in the control plane and those in the data plane. Programmability by external applications is also supported. This section describes how the above are achieved through SDN's basic architectural concepts. Firstly, the differences between the traditional and the SDN architecture are presented. Then, a description of the SDN architecture both in terms of network planes and building blocks is given.

2.3.1 From traditional to SDN architecture

In traditional networks, the control and data planes are combined. Each node is responsible for both functionalities. The control plane is responsible for node configuration and path programming. Once paths have been determined, they are pushed down to the data plane. Examples of existing network nodes that achieve this are Ethernet switches. An Ethernet switch operates at the data link layer of the Open Systems Interconnection (OSI) model and is built up from both planes. Ports used to serve inbound and outbound traffic represent the data plane. These are controlled and configured by the control logic, containing the forwarding logic for the switch. The important part of the control logic is the forwarding table, which contains a list of MAC addresses coupled to the corresponding port. Based on rules represented in this table, traffic is either forwarded to the proper port or flooded in case no match is found. A high-level list of switch functionalities includes: priority settings, MAC filtering, device monitor and link health check, VLAN settings etc. In most cases, switches and other network elements are combined to form a distributed network architecture, offering, in comparison to a centralized one, improved scalability and redundancy. In contrast, the control networks in decoupled architectures are closer to client-server networks. Forwarding devices have limited decision-making capabilities and implement decisions made by controllers. However, it should be noted that configuration changes and other updates are applicable only by directly updating each device. Inevitably, large-scale networks that require, for example, adaptation to traffic demands (through corresponding bandwidth allocation) need both time and resources to be updated. Furthermore, in the traditional architecture resources and policy controls are updated

each time the requirements of external applications are updated. There is, finally, no exposure of information to these applications regarding network state.

On the other hand, SDN is a model based on the idea of moving from the traditional fully distributed model to a more centralized approach. This is achieved by separating the functionalities related to each plane to different elements. In SDN, switches are decoupled from the control plane and serve only the data plane while controllers are responsible for manipulating them. Control decisions in this case are made considering a global view of the network state. In SDN, the control plane acts as a single, logically centralized network operating system in terms of both scheduling and resolving resource conflicts, as well as abstracting away low-level device details, e.g., electrical vs. optical transmission. However, this does not imply that the controller is physically centralized. For performance, scalability, and reliability reasons, the logically centralized SDN Controller can be distributed, so that several physical controller instances cooperate to control the network and serve the applications. Since the controller is aware of the whole network topology, it can easily adapt to requirements regarding scalability and flexibility. For example, the problem of bandwidth allocation is solved by dynamically programming the controller through the exposed northbound API. This architecture gives applications more information about the state of the entire network from the controller, as opposed to traditional networks where the network is not application aware.

The SDN architecture APIs are often referred to as the northbound and southbound interfaces. Many devices from the data layer can be connected to a single centralized control plane which enables the controller to have a network wide view of topology hence providing flexibility for traffic engineers to develop and deploy applications like routing and security. Control networks for SDNs may take any form, including a star (a single controller), a hierarchy or even a dynamic ring.

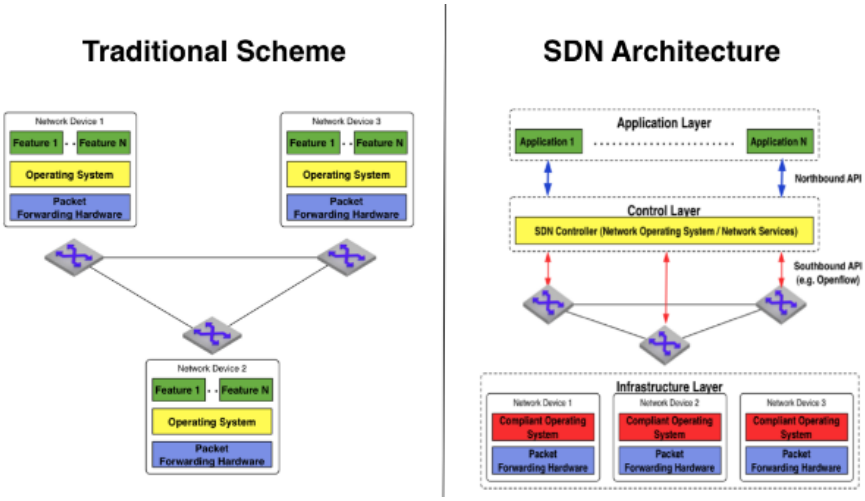

Figure 2: Traditional VS SDN architecture

2.3.2 SDN planes

The first fundamental characteristic of SDN is the separation of the forwarding and control planes. Most research around SDN lists and analyzes the major horizontal groupings (planes or layers) of the SDN architecture. Except for the definition of data and control plane, many references are found that define also application, operational and management planes. For the purposes of this thesis, the three main planes (data, control, application) are considered. A brief description is given below:

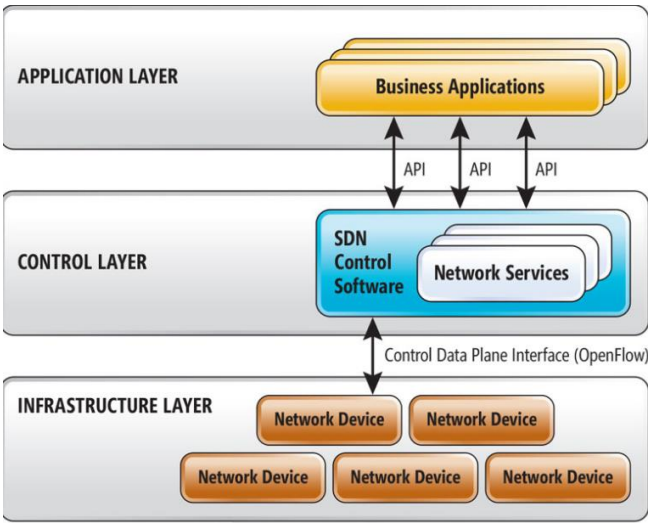

Figure 3: SDN planes

Data Plane

Data Plane (known also as Forwarding Plane or Data Path) is responsible for handling packets in the data path based on the instructions received from the control plane. Actions of the forwarding plane include: forwarding, dropping, replicating and changing packets. The forwarding plane is usually the termination point for control-plane services and applications. For basic forwarding, the device determines the correct output port by performing a lookup in the address table. Special-case packets that require processing by the control or management planes are consumed and passed to the appropriate plane. Finally, a special case of forwarding pertains to multicast, where the incoming packet must be replicated before forwarding the various copies out different output ports [56]. In some cases, in the literature, the operational plane is mentioned as part of the data plane or as a separate plane. It is responsible for managing the operational state of the network device.

Control Plane

Control plane is responsible for making decisions on how packets should be forwarded by one or more network devices and pushing such decisions down to the network devices for execution. The control plane usually focuses mostly on the forwarding plane and on the operational plane of the device. The control plane may be interested in operational plane information. Management functionalities are also part of the control plane (in some cases, these constitute a separate plane). These refer to monitoring, configuring and maintaining network devices, e.g., making decisions regarding the state of a network device. The management plane may be used to configure the forwarding plane. For instance, the management plane may set up all or part of the forwarding rules at once, although such action would be expected to be taken sparingly [54].

Application Plane

Application plane is the plane where applications and services that define network behavior reside. Applications that directly support the operation of the forwarding plane (such as routing processes within the control plane) are not considered part of the application plane.

2.3.3 Building Blocks

The SDN switch, the SDN controller, southbound and northbound interfaces and SDN applications are the fundamental building blocks of the SDN architecture. The following paragraphs provide details for each of the above components.

2.3.3.1 Switch

In general, a network device refers to an entity that receives packets on its ports and performs one or more network functions on them. Examples of functions are: forward a received packet, drop it, alter the packet header etc. A device consists of multiple resources such as ports, memory, and others. Resources are either simple or can be aggregated to form complex resources that can be viewed as a single resource to external network [54]. Examples of network devices include switches and routers. In general, network elements also include devices such as firewalls, load balancers, video transcoders and optical or microwave network elements [54]. Network devices can be implemented in hardware or software and can be either physical or virtual. In SDN, network devices are responsible for forwarding and data processing. Switches in an SDN are often represented as basic forwarding hardware accessible via an open interface, as the control logic and algorithms are offloaded to a controller [13]. Such forwarding devices are commonly referred to, in SDN terminology, as “switches”. The SDN data plane, as described above, constists of network elements, which expose their capabilities to the control plane via interfaces southbound from the controller. Many SDN switches behave much like a standard Ethernet switch and flood traffic out all ports for Ethernet frames destined to broadcast, multicast or unknown MAC addresses. Most SDN switches also flood normal ARP traffic like a typical hardware-based Ethernet switch. However, it is possible to put an SDN switch into an explicit forwarding mode, whereby only flows allowed or configured by the controller are allowed.

Taking these into consideration, it seems that custom-made SDN switches are not required to implement SDN. Nowadays, in SDN architectures different approaches regarding switches are used, depending on needs, vendor etc. For example, hardware switches that are specifically designed to deliver enhanced SDN performance may be used. In other cases, SDN switches that offer high performance network fabrics in

conjunction with SDN to speed data center operations are proposed. Finally, virtual switches are available and can be used for developing services over SDN or to test an application.

2.3.3.2 Controller

The SDN Controller is a logical entity that receives instructions or requirements from the SDN application layer and relays them to the networking components. The controller is also responsible for extracting information about the network from the hardware devices and communicating it back to the SDN applications. Using the southbound API, the controller can add, update, and delete flow entries, both reactively and proactively. The SDN controller represents the SDN control plane and has complete control of the data plane.

The SDN architecture does not specify the internal design or implementation of an SDN controller. It could be a single monolithic process or a set of identical processes arranged to share load or protect one another from failures. It could also be a set of functional components in a collaborative arrangement. Controller components can be executed on computer platforms, including computing resources local to a physical network element. They may also execute on distributed resources such as virtual machines in data centers. It suffices to say that the SDN controller is understood to have global scope and that its components are understood to share with the controller their information and state. Multiple manager or controller components may have joint write access to network resources. In this case, they must either be configured to control disjoint sets of resources or actions, or to be synchronized with each other so that they never issue inconsistent or conflicting commands. More information regarding SDN controllers may be found in the following chapters, since their architecture, capabilities and existing implementations (both commercial and open-source) are the main areas of interest for this thesis.

2.3.3.3 Southbound Interface

Southbound APIs facilitate efficient control over the network and enable the SDN controller to dynamically make changes according to real-time demands and needs. OpenFlow, which was developed by the ONF, is the first and probably most

well-known southbound interface. It is an industry standard that defines the way the SDN controller should interact with the forwarding plane to adjust the network, so as to adapt to changing business requirements [52]. While OpenFlow is the most well-known of the SDN protocols for southbound APIs, it is not the only one available or in development. NetConf uses the Extensible Markup Language (XML) to communicate with the switches and routers to install and make configuration changes. Lisp, also promoted by ONF, is available to support flow mapping.

2.3.3.4 OpenFlow

The OpenFlow protocol can be viewed as one possible implementation of controller-switch (southbound) interactions, as it defines the communication between the switching hardware and a network controller. This provides an abstraction for business applications to use facility provided by the control layer without going into the details of their implementation [15]. It should be noted however, that although one of the basic ideas of SDN is to avoid vendor locking, dependence on one protocol (OpenFlow in this case) does not serve this purpose.

The OpenFlow switch is the basic forwarding element, which is accessible via the OpenFlow protocol and interface. Although at first glance this setup would appear to simplify the switching hardware, flow-based SDN architectures such as OpenFlow may require additional forwarding table entries, buffer space, and statistical counters etc. [13]. OpenFlow switches may be either hybrid (OpenFlow enabled) or pure OpenFlow. An OpenFlow switch consists of a flow table, which performs packet lookup and forwarding. Each flow table in the switch holds a set of flow entries that consists of header fields or match fields and counters (used to collect statistics for a specific flow, such as number of received packets, number of bytes, and duration of the flow [13]). When a packet arrives at the switch the header fields are extracted and matched with the flow entries installed in the switch. If a match is found, the corresponding action in the relevant flow is performed. A default action for packets is executed in case of a table miss.

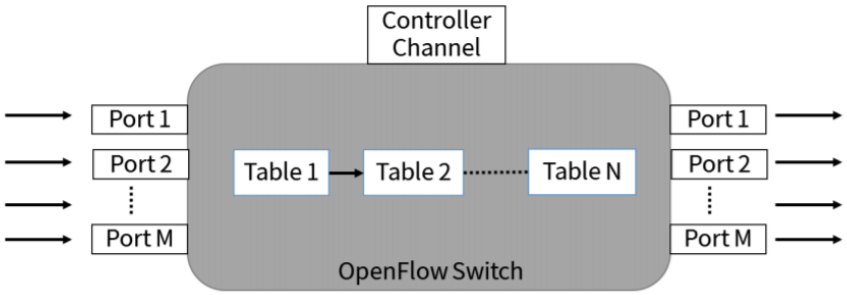

Figure 4: OpenFlow Switch Architecture

A controller uses a secure channel to communicate with the switch. OpenFlow packets are sent over this channel. For security, OpenFlow 1.3.0 provides optional support for encrypted Transport Layer Security (TLS) communication and a certificate exchange between the switches and the controller. The types of message supported by OpenFlow are the following:

Controller-to-switch

Messages sent from the controller that, in some cases, require a response from the switch. This class of messages enables the controller to manage the logical state of the switch, including its configuration and details of flow and group table entries.

Symmetric

Messages from either the controller or the switch. They include “hello” messages that are typically sent back and forth between the controller and switch when the connection is first established [52]. The also include echo request and reply messages that can be used by either the switch or controller to measure the latency or bandwidth of a controller-switch connection or just verify that the device is operating.

Asynchronous

This class includes various asynchronous status messages that are sent to the controller.

2.3.3.5 Northbound Interface

Northbound communication refers to the communication between the business application layer and the control layer. It is considered the least researched and standardized area of SDN. A northbound API is used to implement and develop vendor

independent applications for network management and monitoring, load balancing etc. Another advantage of northbound APIs is that they are easy modifiable using high level languages like Python, Java, C++ etc. At present, no accepted standard protocol exists. Existing APIs have been implemented on ad-hoc basis for specific applications. External management systems or network services may wish to extract information about the underlying network or control an aspect of network behavior or policy. Additionally, controllers may find it necessary to communicate with each other for a variety of reasons. For example, an internal control application may need to reserve resources across multiple domains of control.

The northbound interface is defined entirely in software, while controller-switch interactions must be enabled by the hardware implementation. While there are several controllers in existence, their application interfaces are still in the early stages and independent from each other and incompatible. Until a clear northbound interface standard emerges, SDN applications will continue to be developed in an ad-hoc fashion and the concept of flexible and portable network apps may have to wait for some time [10].

2.3.3.6 Network Applications

Network applications are programs that communicate with the SDN controller via APIs. These applications can build an abstracted view of the network by collecting information from the controller for decision-making purposes. These applications could include networking management, analytics, or business applications. For example, an analytics application might be built to recognize network activity for security purposes. In an SDN-enabled network, service providers can create several applications, aiming at cutting costs, improving customer experience and others. It is expected that not all the SDN applications will be completely new. A lot of them replicate or improve applications that are currently running on routers and switches (control and data plane). For example, routing applications will enable routing decisions based on application level insights and characteristics. Furthermore, using SDN applications, content routing can be designed to perform service availability checks before provisioning flows to the network switches. It is concluded that although SDN is focused mostly on control and data plane, network applications are also responsible for bringing improvements to both operators and users.

2.4 SDN Applications

SDN promises a lot of opportunities in several domains related to computer networking. The separation of control and data plane and other SDN principles set the baseline for using SDN in different cases and enjoy the several benefits it offers. In the following paragraphs scenarios where SDN solutions have been proposed or implemented are presented.

Data Centers

Data centers have evolved in recent years, constantly attempting to meet increasingly higher and rapidly changing demands. Traditional data centers employ routers to connect the core with the internet and switches to connect with servers and other switches. Careful traffic management and policy enforcement is critical when operating at such large scales, especially when any service disruption or additional delay may lead to massive productivity or profit loss [10]. Data center operators continuously migrate virtual machines per changing traffic patterns and demands. Today's data centers have many design requirements, such as easy migration of virtual machines, efficient communication among servers and minimal configuration of switches and hosts [15]. Due to the challenges of engineering networks of this scale and complexity to dynamically adapt to application requirements, it is often the case that data centers are provisioned for peak demand. Thus, they run well below capacity most of the time but are ready to rapidly service higher workloads. For the above reasons, the SDN architecture is highly recommended. Moreover, according to Heller et al. [58] an increasingly important consideration is energy consumption, which has a non-trivial cost in large-scale data centers. Heller et al. indicates that much research has focused on improved servers and cooling through better hardware or software management, but the data center's network infrastructure (which accounts for 10-20% of the total energy cost) still consumed 3 billion kWh in 2006. They proposed ElasticTree, a network-wide power manager that utilizes SDN to find the minimum-power network subset which satisfies current traffic conditions and turns off switches that are not needed. Thus, they show energy savings between 25-62% under varying traffic conditions

Backbone Networks

An SDN architecture may be used in case of large-scale backbone networks to achieve programmability and high availability. The most characteristic example is that of Google (presented later in this chapter). In general, by following a centralized control approach, results bear fruits like better network utilization due to global view, planned deterministic resource allocation by reducing over provisioning and others [15]. Moreover, this approach also makes testing the network easier, since a centralized control can use real production network input to research new ideas and test new implementations.

Internet Exchange Points (IXP)

Today's IXPs employ BGP as their inter-domain routing protocol which suffers a few limitations as it can route traffic only based on destination IP prefix. Deploying SDN at an IXP promises opportunities like freeing it from the constraints of internet protocols, advanced load balancing and others.

Enterprise Networks

Enterprises often run large networks, while also having strict security and performance requirements. Furthermore, different enterprise environments can have very different requirements, characteristics, and user populations. Adequate management is critically important in enterprise environments, and SDN can be used to programmatically enforce and adjust network policies as well as help monitor network activity and tune network performance. Additionally, SDN can be used to simplify the network by ridding it from middle-boxes and integrating their functionality within the network controller

Wireless Access Networks

Several efforts have focused on connectivity in the context of infrastructure-based wireless access networks, such as cellular and Wi-Fi. The vast majority of this end-user demand for application services is originating on mobile devices connected to the network via Wi-Fi. As a result, it's important to consider the role SDN will play in wireless LANs today, and determine how this role will evolve over the next few years. For example, the OpenRoads project [59] envisions a world in which users could move across different wireless infrastructures, which may be managed by various providers.

They proposed the deployment of an SDN based wireless architecture that is backwards-compatible, yet open and sharable between different service providers. This project provided inspiration for subsequent work that attempts to address specific requirements and challenges in deploying a software-defined cellular network.

Optical Networks

Handling data traffic as flows, allows software-defined networks, and OpenFlow networks to support and integrate multiple network technologies. Thus, it is possible to also provide technology-agnostic unified control for optical transport networks and fir facilitating interaction between both packet and circuit switched networks. According to the Optical Transport Working Group (OTWG) created in 2013 by the ONF, the benefits from applying SDN and the OpenFlow standard in particular to optical transport networks include: improving optical transport network control and management flexibility, enabling deployment of third-party management and control systems, and deploying new services by leveraging virtualization and SDN [15].

Home and Small Business

Several projects have examined how SDN could be used in smaller networks, such as those found in the home or small businesses. As these environments have become increasingly complex, the need for more careful network management and tighter security has correspondingly increased. Unfortunately, it is not practical to have a dedicated network administrator in every home and office. Feamster proposes that such networks should operate in a “plug in and forget” fashion, namely by outsourcing management to third-party experts, and that this could be accomplished successfully through the remote control of programmable switches and the application of distributed network monitoring and inference algorithms used to detect possible security problems [9].

2.5 SDN and Network Functions Virtualization

Network functions virtualization (NFV) is a network architecture concept that uses the technologies of IT virtualization to virtualize network node functions into building blocks that may connect to create communication services. For example, NFV may refer to moving services like load balancing and firewalling away from dedicated

hardware into a virtualized environment. NFV solutions have mostly been deployed within data centers for cloud platforms, used in both enterprises and service providers.

NFV and SDN are two closely related technologies that are in some cases complementary. Most of today's NFV platforms contain SDN controllers. On the one hand, although NFV goals can be achieved using non-SDN mechanisms, if SDN principles are used, this can enhance performance, simplify compatibility with existing deployments, and facilitate operation and maintenance procedures. On the other hand, NFV can support SDN by providing the infrastructure upon which the SDN software can be run. Furthermore, NFV aligns closely with the SDN objectives to use commodity servers and switches. It is expected that these solutions may end up merging with orchestration systems such as cloud management platforms or network service orchestration platforms.

Currently, many networking vendors are investing in both NFV and SDN technology. The drivers and benefits of both NFV and SDN technology are similar, which is not surprising since NFV often depends on the use of an SDN controller to achieve its results. This can simplify networking management, speed up the delivery of new services, and potentially reduce costs [5].

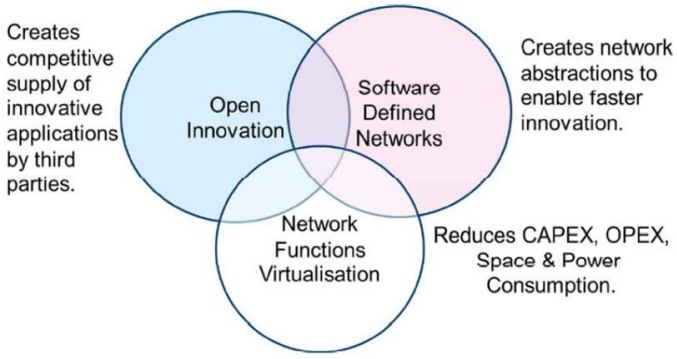

Figure 5: SDN and NFV

2.6 SDN Use cases

There are currently organizations that have partially or totally re-designed their network based on SDN principles. The most known and impressive is the Google backbone network described below.

Google B4 case

An SDN implementation is used for Google's data center interconnection across the planet [18]. These data centers are characterized by massive bandwidth requirements, elastic traffic demand and need of full control over the edge servers and network. These requirements have led to a globally deployed WAN (2011) using an SDN architecture to optimize utilization and balance capacity against application. Several publications describe not only the design and the implementation of the network but also the experience and the lessons learned. A summary is given below.

Google has created two different WANs, one for user-facing peers and one to provide connectivity among data centers (B4). In terms of application the second includes user data copies to remote data centers for availability, remote storage, large-scale data synchronization. These classes are ordered in increasing volume, decreasing latency sensitivity, and decreasing overall priority. For example, user-data represents the lowest volume on B4 [18] and is the most latency sensitive, so it is of the highest priority. Taking into consideration the above and the unique characteristics of Google networks a WAN network was designed and implemented using SDN principles. The design decision for B4 included: use of routers built from merchant switch silicon, 100% link utilization, centralized traffic engineering and hardware and software separation. Regarding OpenFlow, it was chosen since it can leverage a variety of switch elements. B4 has been in deployment for six years offering several advantages but also raising issues regarding system performance, availability and scalability.

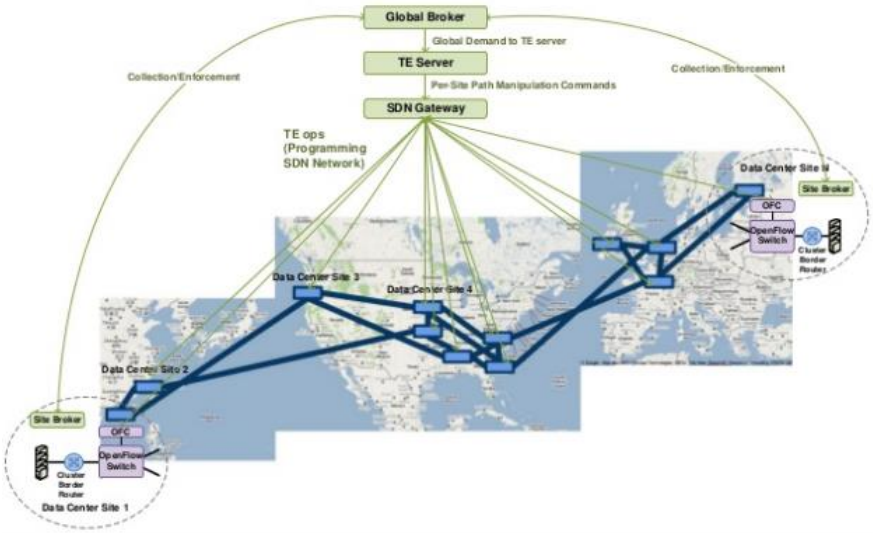

Figure 6: Google SDN Network

2.7 Challenges

Although the SDN concept is not new, only during the last decade has there been a systematic effort not only to standardize SDN but also to apply it to various network topologies. It is therefore expected that there are still areas that need to be further researched and challenges to be overcome. In terms of SDN characteristics, concerns have been expressed for network security, scalability and programmability. This mostly concerns controllers which are responsible for ensuring these characteristics. In terms of areas of research, except for efforts focused on enhancing the above, areas such as northbound interface, controller and switch design are also taken into consideration.

Regarding the northbound interface, currently not many protocols have been defined for controller-service interactions. The goal seems to be to create a more standardized interface that will use a common protocol (as in the case of the southbound interface, through which applications can access the hardware and communicate with other applications. Existing literature (Procera [9], Pyretic [60]) refers to northbound interfaces and proposes the creation of a simple and reusable abstraction that will be used to program controllers.

Furthermore, networking devices must go through a lot of improvement in terms of protocols, business models and applications. Future work will certainly focus on making it easier to compose different components for control and to facilitate easier debugging and testing of the applications.

Security is a major challenge for SDN. There has been limited industry and research community discussion to date on the security issues associated with SDN. A greater focus on security is therefore required if SDN is going to be acceptable in broader deployment. The ONF has formed a corresponding group to highlight and study security related issues. The most significant are centralized control, which exposes a high-value asset to attackers, programmability, integration of legacy protocols, cross domain connection and others. In general vulnerabilities exist across the SDN platform. For example, at the controller-application level, questions have been raised around authentication and authorization mechanisms to enable multiple organizations to access network resources while providing the appropriate protection of these.

Scalability is also another challenge for SDN. In contrast to today's hierarchical networks that are very scalable, the concern with a single controller is whether it can scale to large autonomous systems and handle large volume of traffic. The issue can

loosely be split into controller scalability and network node scalability. As far as the controller is concerned, the following challenges are identified. The first is the latency introduced by exchanging network information between multiple nodes and a single controller. The second is how SDN controllers communicate with other controllers using the east and westbound APIs. The third challenge is the size and operation of the controller back-end database.

To sum up, SDN provides flexibility, centralized control, and open interfaces between nodes, thus enabling a more efficient network. However, to achieve this goal and turn to SDN instead of traditional architecture, a number of outstanding challenges must be resolved.

3. The SDN Controller

Controllers constitute the core of the SDN architecture. They are the central point of the network, responsible for enabling and orchestrating the communication between applications, which represent business logic and network devices (switches, routers etc.). This communication is feasible through northbound and southbound APIs that allow the simplification and automation of the above process. Ideally, controllers should contribute to the intelligence, flexibility, scalability and cost-effectiveness of the overall SDN infrastructure. Consequently, the controller's strategic role has led to its continuous development both in the research and commercial fields. During the last years, progress has been made in many areas. Several controller models have been created by vendors aiming at improving existing network infrastructures of large companies. At the same time, open source solutions are also crucial part of the transition to more intelligent and automated networks. A lot of research has also focused on improving controller characteristics such as availability, scalability and others. Finally, the overall SDN architecture in terms of controller numbers and placement has also been analyzed.

This chapter is an overview of SDN controllers. A brief history is firstly provided. Then, their main capabilities, components and types are described. Finally, a summary of different controller architectures, as these have been described in the literature, is given.

3.1 Brief History

As expected, the controller timeline begins together with the introduction of SDN itself. The first SDN Controller was NOX (2009), which was initially developed by Nicira Neworks. At the same time the OpenFlow protocol's first version was released. Therefore, the first controllers introduced were designed based on the OpenFlow protocol. Since NOX was an open source project, it was donated to the SDN community and soon became the basis for many other solutions. Multiple versions of NOX were then released, i.e. a faster version of NOX or POX (provides Python support). Today, NOX appears to be inactive, while POX is used only by the research community. The next step worth to be mentioned in the SDN controller timeline is the

development of the ONIX platform. ONIX was a distributed control platform for large-scale production networks co-developed by Nicira, NTT and Google. It was also the base for VMware's controller, which is nowadays one of the leading implementations in the market. In 2010, the Beacon [34] controller was introduced. It was an open-source solution, created by Stanford University. Beacon was a Java-based OpenFlow controller and became popular since it was easy to deploy and run. Beacon influenced the design choices of almost all the controllers that came after it. Beacon's descendant was Floodlight, which was developed by Big Switch Networks. Floodlight fixed many of the issues associated with Beacon and became the most feature rich controller available. Through the respective OpenStack plug-in, Floodlight could be used to control large pools of compute, storage and networking resources. Most controllers nowadays provide OpenStack support. Other open source controllers include Trema, Ryu, FlowER, LOOM and OpenMUL. More details regarding the controllers mentioned above may be found in chapter 4 of this thesis. Many controllers have also been produced for commercial use. Vendors such as Cisco, HP, IBM, VMware and Juniper have jumped into the SDN Controller market with their own offerings. On April 8, 2013, the open-source foundation, OpenDaylight, which is part of the Linux Foundation, was announced. OpenDaylight is a Java-based controller, based on the Beacon design. It supports OpenFlow and other southbound APIs and includes critical features, such as high-availability and clustering. As a challenge to OpenDaylight Controllers, On.Lab created the Open Networking Operating System (ONOS) Controller. Companies supporting it include AT&T, Microsoft, HP, Ericsson, NTT, Ciena and Extreme Networks.

3.2 Capabilities

As already mentioned, SDN's main aims are: network management, programmability, data and control plane separation. These are achieved by a centralized model that is driven by controller's components and capabilities. The following paragraphs will describe in detail the capabilities of an SDN controller.

As SDN evolved, the controller's role and capabilities have been also developed regardless of whether the solutions were based on an open source or specific vendor platform. Basic characteristics have been enhanced, and new ones have been added to

offer to organizations more compact and effective solutions. Some of the capabilities of a SDN controller include:

Efficiency

Efficiency is a term used in the scope of this thesis to describe performance, scalability and security. It is desirable that a controller covers these three attributes in an optimal way. In the literature, performance and scalability are used to describe the response time and the number of flows that a controller can handle. This is an important characteristic independently of the use case. Security may refer to several functionalities that a controller should execute to be compliant with the continuously growing number of respective requirements. As the number of controller implementations and versions is growing, the need of comparative studies regarding controller efficiency has arisen. A representative number of researches along with their conclusions will be presented in chapter 5 of this thesis.

Southbound Support

Southbound support has already been defined as the way a controller manipulates network devices to achieve optimized traffic flow. As already mentioned, there are many southbound protocols that may be used, OpenFlow being the most popular. Some of the basic functionality that any OpenFlow controller should be able to support includes field matching, network discovery with Link Layer Discovery Protocol (LLDP) etc. In case of southbound support, implementers need to take into consideration not only protocol features but also possible extensions, newer versions and others. For example, in the case of OpenFlow, functionality such as IPv6 support is not part of OpenFlow v1.0 but is part of the OpenFlow v1.3 standard [4].

Northbound Support

Northbound APIs are used to implement network abstraction and programmability and may be used by customer-facing orchestration systems and third-party applications. It is crucial to ensure that a controller is suitably deployed to orchestrate communications, both at Layer 2- 3 and Layer 4-7. For example, the controller should be able to support OpenStack orchestration systems. OpenStack refers to an open source software platform for cloud computing, mostly deployed as infrastructure-as-a-service (IaaS). Moreover, the controller should also support vendor-specific protocols. Typical SDN

applications include traditional network services such as firewalls and load balancers as well as orchestration systems such as OpenStack. These applications could also include traffic engineering or applications that gather data used to perform tasks such as managing the network.

Programmability

Programmability is one of the most valuable characteristics of SDN and specifically controllers. Traditionally, network configuration is achieved by applying respective rules on a device-by-device basis. Inevitably, this static approach is time consuming, error prone [4] and in some cases inconsistent. It might also lead to downgraded network performance. Programmatic interfaces are the key components of every controller. One of the most common examples of programmability is traffic redirection that may be requested for several purposes (i.e. traffic allocation in terms of time, place, security). As a first step, the northbound API makes the control information that has been centralized in the controller available to network applications. These are then capable of changing the network to perform tasks such as forwarding packets over the least expensive path or changing the QoS settings based on the available bandwidth or other factors. Another example of programmability in an SDN controller is the ability to apply filters to packets to determine whether to drop or pass them. These dynamic filters may be based on packet header matching and may be simple or more sophisticated, consisting of complex combinations of multiple packet headers. The filters should be able to be deployed dynamically and it is the role of the SDN controller to push the associated flow table entries down to the switches.

Monitoring

Network monitoring is another controller capability. Through protocols (i.e. OpenFlow) and relevant tools, the controller can identify problems in the network and facilitate troubleshooting process. Advantages of the controller include detailed flow monitoring (and not random sampling), monitoring of specific classes of traffic etc. The controller should support standard monitoring protocols and techniques, so the information can be integrated with other management and orchestration systems. It should, for example, be possible to monitor the health of the controller and the virtual networks that the controller supports using SNMP.

Network Virtualization

Network Virtualization is the ability to create logical, virtual networks that are decoupled from the underlying hardware. Network virtualization refers to both OSI Layers 2-3 services (i.e. routing) and Layers 4-7 services such as load-balancing. Common examples of network virtualization that have been in production for decades are virtual LAN (VLAN) and Virtual Rooting and Forwarding (VRF). Due to rapid changes in terms of network volume, performance requirements and others, the above methods are considered limited both in scope and in value. SDN controllers facilitate the implementation of network virtualization in an end-to-end manner and thus enable organizations to dynamically create virtual networks and meet demanding requirements.

Flexibility

Flexibility is another requirement a controller needs to achieve. On the one hand, a variety of applications needs to be supported. On the other hand, controller applications should use a common framework and programming model to ensure that that the exposed APIs are consistent and easily consumed. This is important for several purposes such as troubleshooting, system integration and others.

Topology

SDN controllers should be also evaluated against their adaptability as far as network topology is concerned. Network topology is the arrangement of the elements (in our case controllers, switches etc.) of a computer network. In case of large networks, the options regarding the topology of extra controllers should be analyzed. Moreover, communication between controllers (east-west bound communication) is also part of the topology. Currently, there are some discussions in the industry around standardizing how controllers will talk to one another. A common technique uses BGP for exchanging information between controllers. Finally, issues regarding the level of centralization architecture are considered. Later in this chapter examples of topologies will be presented.

3.3 Components

The SDN architecture (described in paragraph 2.3) does not specify the internal design or implementation of the controller. The controller may be a single software system or multiple systems arranged to execute functionalities such as load balancing, device management etc. Moreover, it may run on local resources or distributed resources such as on Virtual Machines (VMs) in data centers. In general, the controller is considered as a black-box and is defined by the services it provides. Figure 7 constitutes a simplified architecture framework of a SDN controller. Core functional modules and interfaces are depicted. According to [11] there are three well-defined layers in most of the existing controller platforms: application, orchestration and services, the core controller functions, and the elements for southbound communications. However, what should be noted is the fact that the boundaries of an SDN controller are not well defined. For example, some consider service management as an integral part of an SDN controller, while others consider it a separate function or an application that runs independently of the controller. The same is true for other functions too. An indicative description per layer/component is given in the following paragraphs. Depending on circumstances, additional functions may be required.

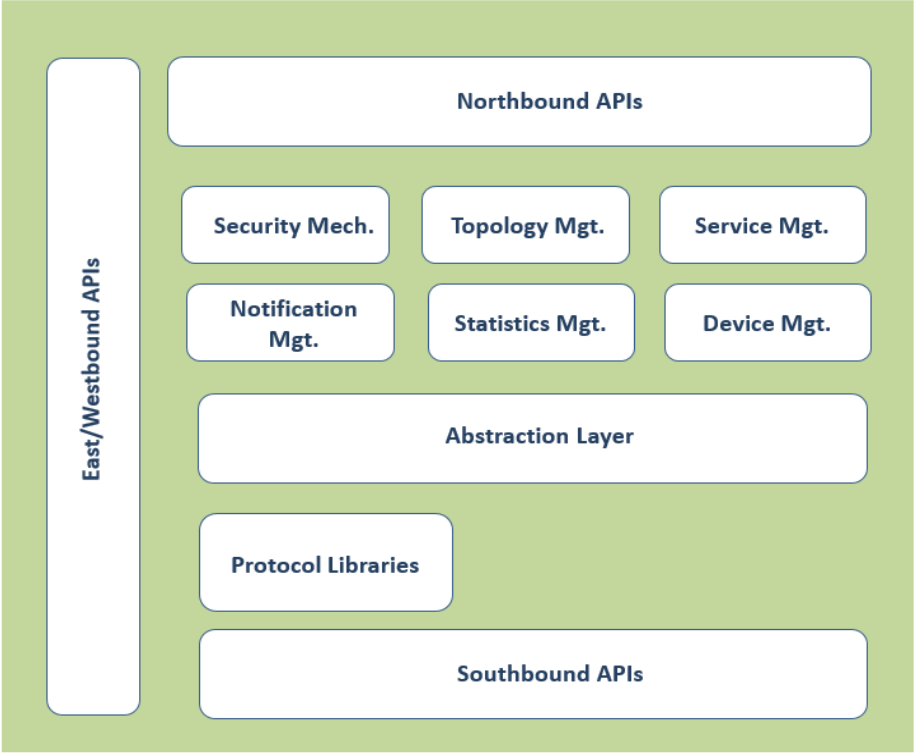

Figure 7: Controller's Architecture Framework

3.3.1 Southbound Layer

In the SDN controller architecture, the southbound layer represents what has been mentioned so far as the southbound interface or API. It is the communication channel between controller and network devices that is realized through an API. In most cases, a TLS connection is established between the devices. TLS is used to ensure a secure and authenticated communication. The southbound layer consists of device drivers [11]. This allows the controller to use different APIs and multiple protocols to manage a range of physical and virtual devices. The choice of the southbound protocol depends on the use case. Except for the Openflow protocol that already been described, the following may be also supported.

NetConf Protocol

It defines a mechanism through which a network device can be managed and configured. It uses an RPC-based mechanism to facilitate communication between the

client and the server. The client can be a script or application, typically running as part of a network manager.

Path Computation Element (PCE) Protocol

It is a full-scale provisioning protocol. It simplifies path computation by separating network topology determination from path creation. In case of other protocols, an SDN implementation may require replacement of existing network elements while in case of PCE protocol only upgrade of head-end routers is required. PCE is also capable of incorporating optical network parameters in path computation [11] and creating paths across routing domains.

Multiprotocol Label Switching - Transport Profile (MPLS-TP) Protocol

It is a version of the MPLS protocol that is used in packet switched data networks. It provides a reliable packet-based technology and its added features include maintenance functions, legacy data traffic management and others.

Extensible Messaging and Presence Protocol (XMPP)

It is a protocol based on XML. It enables the real-time exchange of structured data between two network entities. As in cases of other protocols, it is regarded as a mature protocol that allows interoperability with legacy networks and systems.

Border Gateway Protocol (BGP)

It is an existing core-Internet routing protocol that in the case of SDN it may be used for topology discovery.

3.3.2 Abstraction Layer

The abstraction layer is responsible for translating abstract models that are used by controlles functions such as Service Management and Resource Management to a device-specific data model [14]. The abstraction layer may be based on one or more models and is a uniform point of reference. It should be noted here that the term Abstraction Layer may refer to any layer used to differentiate two physical or logical entities in SDN architecture. The IRTF (RFC 7426 standard) introduced SDN

fundamental concepts in the form of "abstraction layers". The first is that of the Network Services Abstraction Layer (NSAL) which refers to the layer between the higher-level applications and network services and the SDN Controllers. It also introduced other abstraction layers including the Control Abstraction Layer (CAL), the Management Abstraction Layer (MAL) and the Device and resource Abstraction Layer (DAL). Each of these layers or APIs provide the higher layers with a common way to communicate their requirements to the layer or layers below them.

3.3.3 Network Services Layer

The network services layer consists modules executing the main functionalities of a SDN controller. Some examples may be found below:

Service Management Module

The Service Management function is responsible for all aspects of service instantiation and management in the network [14]. Service management defines the characteristics of a service such as protocol, structure (i.e. point-to-point, point-to-multi-point, multipoint-to-multipoint etc.), end-points, other attributes (QoS, bandwidth, latency, prioritization, security, availability etc.). Services provided by the Service Management module are used through northbound interfaces by external applications.

Topology Management Module

The Topology Management function includes the maintenance of existing network topology. Every change in topology is identified by the topology management module. This is achieved through LLDP messages that are exchanged with network devices. For example, if the received LLDP message matches a known switch then a new link is established in the network.

Statistics Module

A controller provides several basic functions, such as statistics collection. These may be configured based on needs to include services, resources or both. This information may be also exposed to external applications.

Forwarding and Routing Management Module

Forwarding and Routing Rules functionality is required to define the set of forwarding rules that should be communicated to network devices. Furthermore, the routes between the source and destination addresses need to be provided. This functionality requires input from other services such as Topology and Service Management.

Device Management Module

This module is responsible for managing the devices in the network. It is notified upon any update regarding the devices such as addition or deletion. It is also notified when the device IP addresses have been added, updated or removed.

Resource Management Module

This function enables northbound systems to create, modify or delete resources. It provides functionalities such as optimal path selection, tunnel load balancing, bulk optimization, and traffic re-route etc.

Openflow Module

An OpenFlow module exists in most controllers to provide functions related to Openflow such as messages, actions, table entry, flow rules matching and statistics. As already described, a controller may support other protocols to manage its switches and underlying networks as required by the application.

3.3.4 Northbound Layer

The interaction between the controller and the applications is realized through a communication interface. As already mentioned, there is currently no formal protocol used to manage this communication. Instead, software APIs enable the programmability of the controllers by exposing network services to the application layer.

So far, different controller use cases have been described. Moreover, reference has been made to the variety of capabilities and features that the controllers offer. Currently, there are both commercial and open-source controllers in the market. The appropriate controller depends on various factors. As already described, there are nowadays several SDN applications and therefore several controllers that have been designed to serve

their needs. There are controllers designed for WAN deployments, enterprise campuses, data centers and others. It is not necessary that a controller could fit in all types of deployments. For example, controllers that are designed for WAN deployments might not work as well in the data center environment [12]. The number of deployments, areas and time these deployments have been running should be also taken into consideration. Except for type of deployment, business needs that need to be covered are also important. Specifically, a controller should be able to integrate through northbound API with applications that cover specific needs. Compatibility of a controller with the overall network architecture is also important. Initially, SDN applications of data centers required controllers being part of an integrated solution. In this case, controllers focused on the management of data center resources such as compute, storage, virtual machine images and network state. Then, needs focused on network abstraction and therefore new types of controllers, specialized in network management emerged. The driver for this second wave of controllers is the potential expansion of SDN applications out of the data center and into other areas of the network where the management of virtual resources like processing and storage does not have to be so tightly coupled in a solution. Integration in terms of applications and network devices is complicated. For example, depending on the use case the controller will integrate with northbound platforms responsible for cloud management or orchestration management (i.e. OpenStack, CloudStack). Controllers should also be capable of integrating with different types of switches (i.e. hybrid, SDN switches) and be compatible with various protocols. As controller deployment matures and new versions are released, stability of APIs will be also estimated. Comparisons of commercial and open source solutions have concluded that the latter might provide better transparency, but they do not always provide long-term stable APIs. To sum up, there are many attributes that should be taken into consideration to choose the appropriate controller, maturity, reliability and smoothness of integration being some of them. In the following paragraphs, types of controllers will be listed. Differentiation depends on various criteria. Fundamental categories (i.e. commercial and open-source) and more specific ones are described.

3.3.5 Commercial and Open Source

The existence of both open-source and commercial solutions has been mentioned multiple times so far. This is because in the case of the SDN architecture, open source solutions are developing at a remarkable pace. Nowadays, open source software development has become the point of evolution since it allows the cooperation of people around the world aiming at solving problems and creating more sophisticated solutions. Universities have created several open source projects related to SDN and OpenFlow. Open source solutions are not restricted to the research area but have also expanded to the market. Many companies such as Adara, Bigswitch and NEC have released open source versions of their products. The open source versus proprietary dilemma makes sense in cases someone considers extending the controller or has proprietary modifications specific to his business. Open source solutions reduce the probability of vendor lock-in. This however will gradually disappear since there are many efforts on preventing lock-in and standardizing architecture components (i.e. southbound APIs). Eventually, the choice between commercial and open source controllers concerns their functionality and the way these may be used in an existing architecture.

The following controller platform tables show some current controller implementations. Open source controllers have been divided into active and not active. Not active refers to controllers that are no longer maintained or updated.

Open Source Controllers 2017	
Active	**Not Active**
Floodlight	Beacon
LOOM	FlowER
OpenCotail	NOX
OpenDaylight	NodeFlow
OpenMUL	
POX	
Ryu	
Trema	
ONOS	
Maestro	

Table 1: Open Source Controllers

Commercial Controllers 2017
Cisco - ACI
VMware - vCloud/vSphere
Nokia - Nouage
Avaya
Huawei
Big Switch Networks
Juniper - Contrail
Ericsson
Brocade

Table 2: Commercial Controllers

According to [5], the most popular open source deployment today is OpenDayLight (ODL). This is a project from the Linux Foundation, which got its first release, named Hydrogen, in 2014. ODL has more than 100 deployments, including companies such as Orange, China Mobile, AT&T, T-Mobile, Telefonica and others. Contributing individuals recently exceeded 500. Another big movement in the controller market is the Open Networking Operating System (ONOS), which was open sourced in December 2014 and focused on serving the needs of service providers. It is not so widely adopted as ODL but has been gaining momentum in cases of WANs

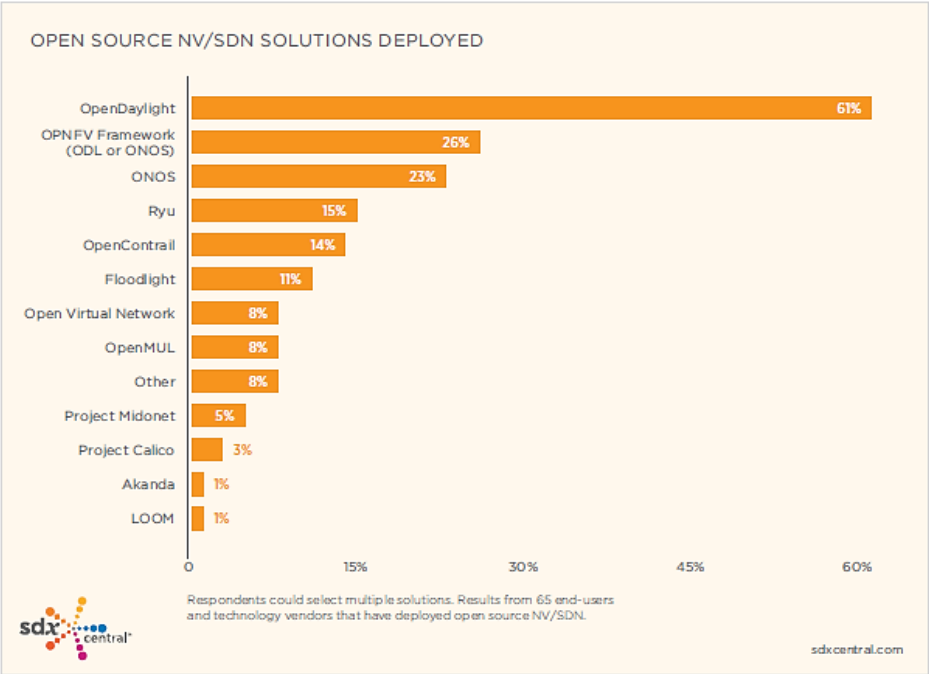

Figure 8: Open Source Deployments 2016 [12]

According to the SDX survey [12] shown in the figure above, ODL was chosen by 61% of the respondents who have current open source deployments. ODL is also the technology cited by the most respondents when asked what open source solutions they would consider (65%). The next most cited technology was the OPNFV framework (ODL or ONOS) at 31%.

3.3.6 Centralization mode

Until this point, centralization has been identified as one of the main characteristics of SDN and controller in particular. However, in practice there are cases where the control plane may be distributed. In the case of centralized controller/s, one or more controllers manage all the network elements in the system, and retain a global view of the entire network [15]. Most SDN controllers today run in this way. It is a simpler solution that ensures that the controller has a global and consistent view of the topology. Integration with legacy systems is also easier in this case. However, there is a high dependency on cluster availability and all services are handled centrally. In case of distributed controller/s, a local controller runs on each compute node and manages the network elements directly. Latency and scalability are better in this case. On the other

hand, a global view of the network is not achieved and synchronization of the overall topology becomes complex as the amount of compute nodes increases.

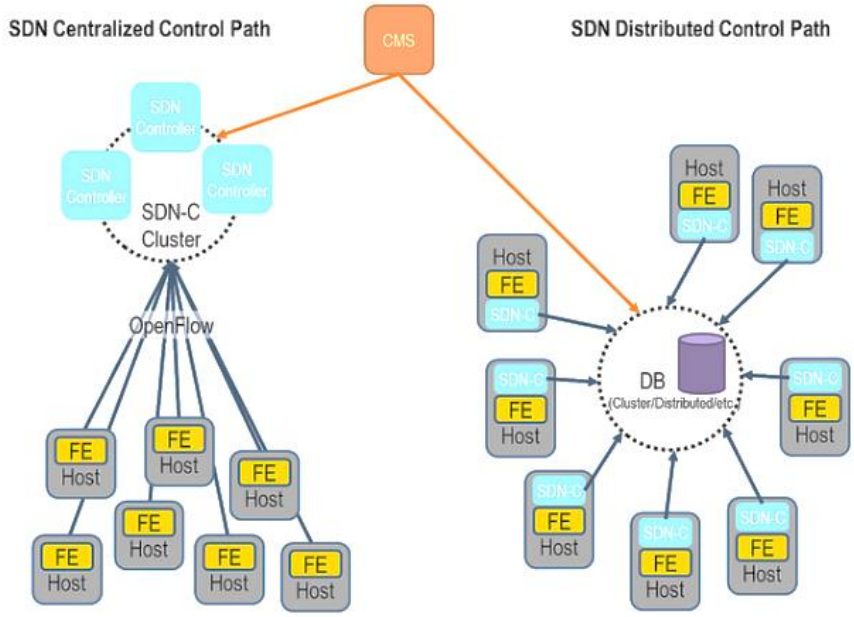

Figure 9: Centralized vs Distributed Architecture

There are cases of controllers (i.e. Onix, HyperFlow) that have been designed to maintain a logically centralized but physically distributed control plane. In this way, look-up overhead is decreased by enabling communication with local controllers, while still allowing applications to be written with a simplified central view of the network [10]. Hybrid approaches, such as Kandoo (to be described later in this chapter) also exist.

3.3.7 Reactive and Proactive policy

A controller may be configured in either reactive or proactive mode. In the reactive approach, network elements must consult a controller each time a decision must be made. For example, in case of a new packet, the switch sends to the controller the first packet of a flow, and the controller is responsible to appropriately configure flow table to handle the rest of the flow. This causes a small performance delay. This delay, in most cases, is considered negligible. However, it may be a concern if the controller is geographically remote [10]. This approach however presents the most efficient use of existing flow table memory. In the proactive approach, the controller programs network

elements proactively based on the existing network view. This approach has zero additional flow setup time because the forwarding rule is already defined. However, network management in this case is more complex. An example of proactive control was the Distributed Flow Architecture for Networked Enterprises (DIFANE). DIFANE [51] was proposed by Princeton University and described an architecture where the controller partitions rules over a hierarchy of switches, such that the controller rarely needs to be consulted about new flows and traffic is kept within the data-plane. Experiments showed that DIFANE reduced first-packet delay from a 10ms average round-trip time (RTT) with a centralized NOX controller to a 0.4ms average RTT for new single-packet flows. These measurements however concern outdated implementations (the DIFANE paper was published in 2010).

3.3.8 Supported Protocols

As expected, another categorization of controllers refers to those supporting the OpenFlow protocol and those that do not. The first OpenFlow controller was NOX and since then most open source controllers support the protocol.

A high-level description of its functionality is the following. An OpenFlow controller is responsible for performing actions related to both control and management plane. It centralizes network intelligence, while the network maintains a distributed forwarding plane through OpenFlow switches and routers. According to [24], the controller runs on a network-attached server and there are different control configurations depending on: location, flow routing and behavior. Location refers to centralized or distributed mode. In the first case, the OpenFlow controller manages all devices while in the second, each controller corresponds to a set of switches. There are also different types of flow routings. Every flow could be individually set up by the controller. In this case a flow table contains one entry per flow. Another strategy for routing could be the aggregated one, in which one flow entry covers large groups of flows. Finally, there could be wildcard flow entries. In this case the flow table contains one entry per category of flows. The most common OpenFlow controller operation mode is the reactive one. The controller listens to switches passively and configures routes on-demand. It receives messages from the switches and maintains a global MAC table. The controller can define the port that the flow must be forwarded to or take other actions, such as dropping the packet. When a switch receives a packet that does not

correspond to an entry in its flow table, it forwards the message to the controller, asking for the action to take for the unknown packet. Upon receiving the message, the controller looks for the destination location and sets the path by sending OpenFlow messages to affected switches. The controller must set the entire path by sending configuration messages to all switches from the source to the destination. In case of a time-out, the entry is excluded from the table. In the proactive mode, paths are set up in advance. More details regarding reactive and proactive modes may be found in the respective paragraphs.

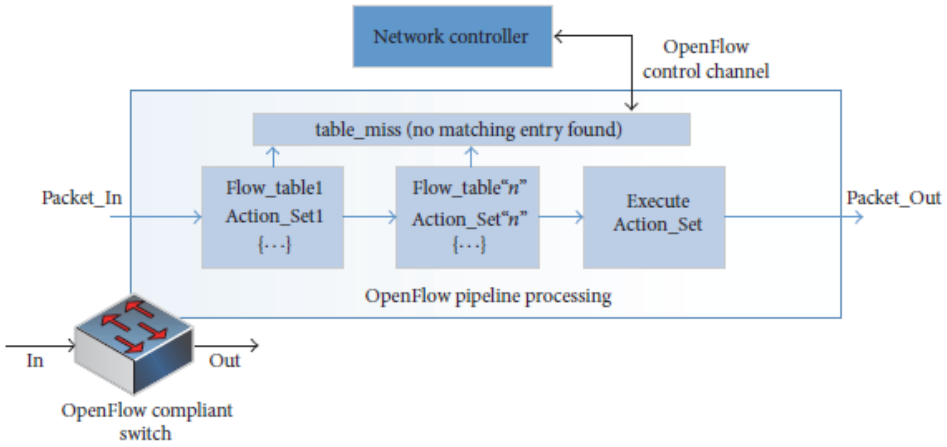

Figure 10: OpenFlow functionality

Besides the use of OpenFlow, there are SDN controllers that leverage IP/MPLS network functionality to create MPLS VPNs. There are also NETCONF-based controllers and Radius/Diameter-based controllers such as PCRF and/or TDF, in mobile environments. The following table includes some of the most popular OpenFlow controllers.

OpenFlow Controllers
NOX
POX
Beacon
Trema
Floodlight
Maestro
Ryu
OpenDaylight

Table 3: OpenFlow Controllers

3.3.9 Special Purpose Controllers

In addition to the variety of open source and commercial controllers that may differ depending on their configuration, the protocols they support and other attributes, special purpose controllers have also been implemented. Some examples found in the literature are Flowvisor, RouteFlow, Simple Network Access Control (SNAC), OpFlops, and Resonance. These serve specific tasks such as transparent proxy between switches and multiple controllers, virtualized IP routing over OpenFlow network switches. Some details may be found below:

FlowVisor

It is an experimental controller that enables network virtualization by dividing a physical network into multiple logical networks. It acts as a transparent proxy between OpenFlow switches and multiple OpenFlow controllers. FlowVisor ensures that each controller touches only the switches and resources assigned to it.

RouteFlow

It is an open source project to provide virtualized IP routing over hardware. It is composed of an OpenFlow Controller application, an independent server, and a virtual network environment that reproduces the connectivity of a physical infrastructure and runs IP routing engine.

SNAC

It is a Controller targeting production enterprise networks. It is based on NOX and uses a web-based policy manager.

3.4 Controller Placement Problem

So far, emphasis has been given to the detailed description of the controller as unit. However, in most cases SDN networks consist of several controllers that need to be appropriately placed to achieve an optimal and simple centralized network. At this point, issues such as scalability, reliability, performance and availability of the network

arise. The overall architecture of controllers influences every aspect of a decoupled control plane, from load-balancing to fault tolerance and performance metrics. In some types of networks, availability, response and convergence time may be affected. What we should also take into consideration is the fact that requirements regarding the above factors (fault tolerance, delays etc.) vary depending on the type of network. For example, in a WAN network it is required to minimize propagation delays, while in a data center or in the enterprise, maximization of fault tolerance is required.

Research has focused on analyzing which is the optimal place and number of controllers given a specific type of network to achieve the desired level of performance and failures. In the literature, this is called the "Controller Placement Problem". In [8], the propagation latency of an Internet2 production deployment and over 100 WAN topologies were studied. As expected, control placement depended on desired reaction bounds, metric choices and the network topology itself. To determine the tradeoffs between reliability and latency, it is suggested that the best controller placement is using one controller that yields the optimal reliability metric, while optimizing the average latency. On the other hand, optimizing reliability increases the average and worst-case latency.

Since SDN is a logically centralized network, reliability of the control plane is of critical importance. Therefore, mechanisms to avoid a single point of failure should exist. According to [7], the most fundamental recovery mechanism is the ''primary-backup replication'' approach. In this model, backup controllers assume network control in case the primary fail. There are, however several issues such as primary and backup controller coordination during fail over, that need to be further studied. High availability is also required in most types of networks. This can be achieved through improved southbound APIs and optimal controller placement. Findings until today indicate that allowing network elements to connect to multiple controllers maximizes availability. It has also been shown that the number of required controllers is more dependent on the topology than on network size. In some cases, it is proposed that SDN controllers delegate control functions, which are used to report state and attribute value changes, threshold crossing alerts, hardware failures etc. In that way, operational efficiency is increased.

As far as performance is concerned, several approaches have also been proposed. For example, one approach was the IRIS IO engine [11], which adopts a multi-queue design that results in a higher flow set-up rate. Flexibility and modularity are proposed

to be handled through hierarchical models of controllers that may offer different abstractions and scopes thus increasing modularity. Finally, interoperability between controllers should be established. Initiatives regarding interoperability include portable programming languages such as Pyretic and development of east/westbound interfaces among controllers (i.e. SDNi, ForCES and others).

Designs of different deployments for multiple controllers have been proposed. In [7], a classification is proposed: logically distributed controller deployments, physically distributed controller deployments, hierarchical controller deployments and hybrid controller deployments. Some examples may be found in the following paragraphs.

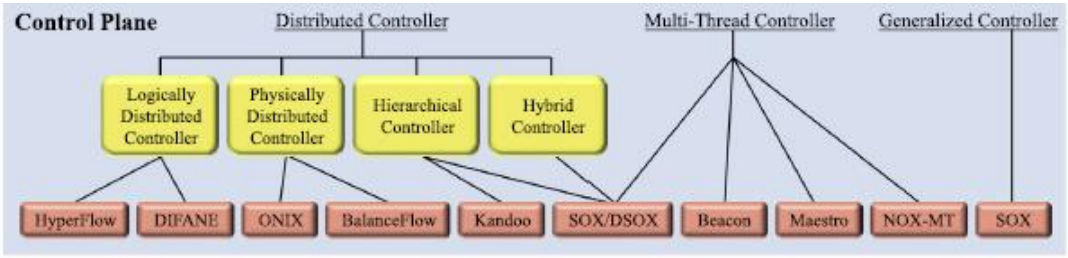

Figure 11: Controller's architectures

HyperFlow

Hyperflow is a logically distributed controller deployment. It is composed of OpenFlow switches and NOX controllers, each of which runs the HyperFlow controller application. HyperFlow localizes decision making to individual controllers for minimizing the control plane response time to data plane requests, and provides scalability while keeping the network control logically centralized. Through the synchronization schemes, all the controllers share the same network view and locally serve requests without actively contacting any remote node, thus minimizing the flow set-up times.

DIFANE

DIFANE has already been mentioned above since it is an example of a controller working in proactive mode. DIFANE is a distributed flow management architecture that consists of a controller that generates the rules, authority switches that are programmed by the controller and simple switches. Authority switches can be a subset of existing switches in the network, or dedicated switches. Upon receiving traffic that does not match the rules, the switch redirects the packet to the appropriate authority

switch. The authority switch handles the packet in the data plane and sends feedback to the switch to cache the relevant rules locally. In this way, all data plane functions required in DIFANE are based on specific rules. Simple switches cache rule so that most of the data traffic hits in this cache. Authority switches also store rules that are installed and updated by the controller. Finally, partition rules are installed by the controller in each switch.

Onix

Onix is a physically distributed controller deployment, which runs on a cluster of one or more physical servers. Onix is responsible for giving the control logic programmatic access to the network. Using the Onix API, which offers a view of the physical network, control applications can read and write state to any element in the network, hence keeping state consistent between the in-network elements and the control application that runs on multiple Onix servers. The copy of the network state tracked by Onix is stored in a data structure named the Network Information Base (NIB), which is a graph of all network entities within a network topology.

Kandoo

The hierarchical controller deployment of Kandoo consists of a two-level hierarchy for controllers: local controllers that execute local applications and a centralized root controller that runs non-local control applications (i.e., applications that require access to the network-wide state). Each switch is controlled by only one Kandoo controller, and each Kandoo controller can control multiple switches. If the root controller needs to install flow-entries on switches of a local controller, it delegates the requests to the respective local controller.

4. Open Source Controllers

As already mentioned, there is a variety of controllers and platforms to consider when selecting an SDN strategy. The following chapter provides further details regarding popular SDN controllers. Due to limitation regarding available information for commercial controllers, only the profiles of open source frameworks and controllers, as these have been listed in paragraph 3.4.1 of this thesis, are presented. A summary table regarding some basic features of the controllers is also presented. What should be highlighted is that comparative analysis and selection of a controller is considered nowadays really challenging since it is a multi-faceted problem. The most suitable controller may be selected only if a specific set of requirements is taken into consideration.

4.1 OpenDaylight

"Communications providers and enterprises alike are eager to build or adapt their networks to be more flexible and responsive to their organizations' and customers' needs. At the same time, they are driving network automation to improve operational efficiency. OpenDaylight, the largest open source SDN controller, is helping lead this transition. ODL is a modular open platform for customizing and automating networks of any size and scale. The ODL project arose out of the SDN movement, with a clear focus on network programmability. It was designed from the outset as a foundation for commercial solutions that address a variety of use cases in existing network environments" [28]. This is the mission of the ODL project, which was introduced in early 2013. It was originally led by IBM and Cisco but then it was hosted under the Linux Foundation Project. Since then, ODL has received the industry's wide support and resources and is nowadays the most popular open source framework in SDN industry.

The ODL platform is designed to cover a lot of use cases such as [28] Network Resources Optimization (NRO), Automated Service Delivery, Cloud, NFV and others. Using ODL services such as dynamic network optimization, on-demand services (i.e. bandwidth, dynamic VPN services etc.), agile service delivery on cloud infrastructure

are provided. ODL may be also used to achieve centralized administration of the network. Until today, the ODL community has introduced several releases, aiming at enhancing the framework through additional features, existing component redesign etc. The first software code release for the ODL controller was Hydrogen. It constituted of three different editions based on user type: Base Edition, the Virtualization Edition, and the Service Provider Edition. The second code release was Helium. It included a new user interface, and a more simplified and customizable installation process, due to the use of the Apache Karaf. This code release also has deeper integration with OpenStack, including improvements in the Open vSwitch Database Integration project, as well as other features like Security Groups, Distributed Virtual Router, and Load Balancing. The most significant change in Helium refers to the service abstraction layer. Hydrogen used an API-driven service abstraction layer, which had limitations. Specifically, the controller needed to know about every type of device in the network and have an inventory of drivers to support them. Helium introduced a model-driven service abstraction layer, which means that the controller didn't have to account for all the types of equipment installed in the network, allowing it to manage a wide range of hardware and southbound protocols.

The ODL platform built on this advancement in its third release, Lithium, which was introduced in June of 2015. This release focused on broadening the programmability of the network, enabling organizations to create their own service architectures to deliver dynamic network service in a cloud environment and craft intent-based policies. The fourth release, Beryllium, was released in February of 2016. Significant improvements in performance, scalability and functionality were introduced. In the Boron release focus was given on two leading types of deployments, with enhancements to cloud and NFV support as well as large-scale network engineering. Boron also provided new tools and documentation to development, as well as greater integration with larger industry frameworks from (Open Platform for NFV) OPNFV and OpenStack to CORD and Atrium Enterprise. Carbon is the sixth release of ODL. With the Carbon release, the ODL community emphasized on enhancements to support Internet of Things (IoT), metropolitan Ethernet and cable operator needs, integrated NFV management [28].

The ODL architecture is based on micro-services that are used to control applications, protocols and plugins, as well as to provide connections between external consumers and providers. To do so, it employs a model driven approach based on a YANG data model to describe the network, the functions to be performed on it and the

resulting state or status achieved. YANG is a data modeling language that was initially designed by the IETF NETCONF Data Modeling Language Working Group. [36] It can be used to model both configuration data as well as state data of network elements. Furthermore, YANG can be used to define the format of event notifications emitted by network elements and it allows data modelers to define the signature of remote procedure calls that can be invoked on network elements [36]. The language, being protocol independent, can then be converted into any encoding format that the network configuration protocol supports. In case of ODL, the YANG data model is used to facilitate the creation and combination of micro-services to solve more complex problems. The XML nature of YANG data model presents an opportunity for self-describing data, which controller components and applications using the controller's northbound APIs can consume in a raw format, along with the data schema [52]. Utilizing a schema language simplifies development of controller components and applications.

In ODL, through Model Driven Service Abstraction Layer (MD-SAL), any application or function can be bundled into a service that is then then loaded into the controller [28]. Services can be configured and bound together in any number of ways to match network needs. This model of service abstraction presents an opportunity to combine both northbound and southbound APIs and the data structures used in various services and components of an SDN controller. These modules are linked dynamically into the MS-SAL. MS-SAL figures out how to fulfill the requested service irrespective of the underlying protocol used between the controller and the network devices. This provides protection to the applications as the OpenFlow and other protocols evolve over time. It provides basic services like Device Discovery which are used by modules like Topology Manager to build the topology and device capabilities. Based on the service request the SAL maps to the appropriate plugin and thus uses the most appropriate southbound protocol to interact with a given network device.

On the northbound side, the interaction between the controller and the applications is done via Web Services for the request-response type of interaction. The controller exposes open Northbound APIs which are used by applications. Open Service Gateway Initiative (OSGi) framework and REST API are supported. The OSGi framework is a Java framework for developing and deploying modular software programs and libraries. It is used for applications that will run in the same address space as the controller while the REST API is used for the rest.

OpenFlow enabled switches connect to two or more instances of the controller through TCP. The platform follows a controller model that enables the use of OpenFlow, as well as alternative southbound protocols. ODL includes support for the broadest set of protocols in any SDN platform. For example, the platform supports OpenFlow and OpenFlow extensions such as Table Type Patterns (TTP), as well as traditional protocols including NETCONF, BGP/PCEP and CAPWAP. Additionally, ODL interfaces with OpenStack and Open vSwitch through the OVSDB Integration Project.

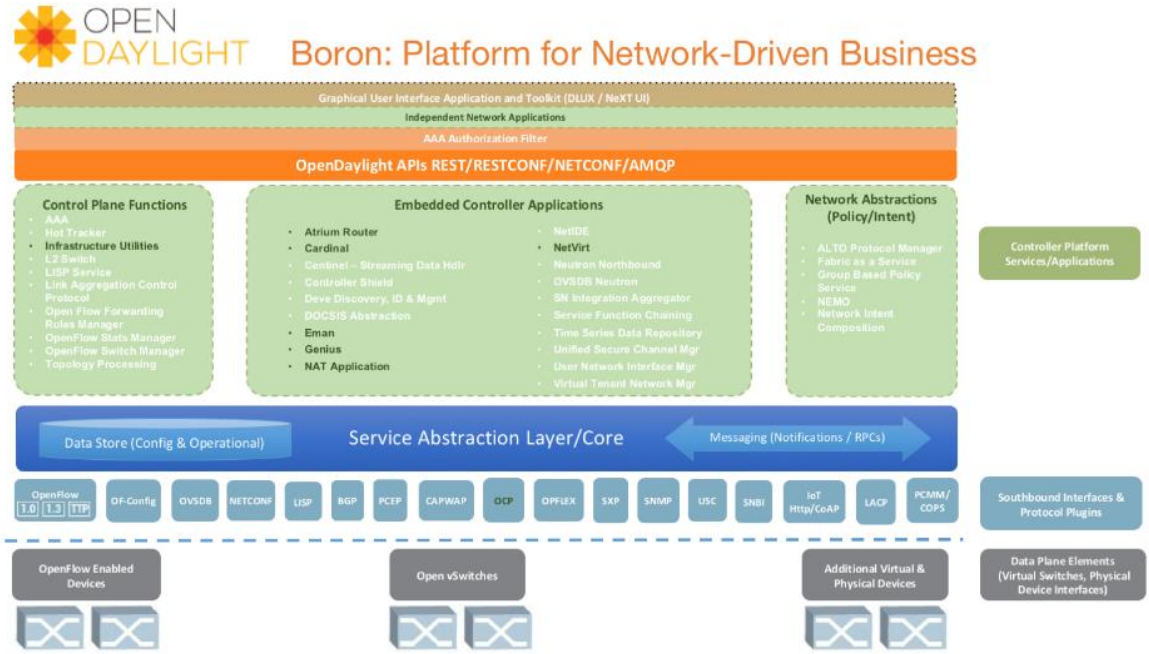

Figure 12: OpenDaylight (Boron version) Architecture diagram [28]

4.2 ONOS

ONOS' mission is to "produce the Open Source Network Operating System that will enable service providers to build real Software Defined Networks" [60]. Its first release, Avocet, was open-sourced in December 2014. Ten releases have followed until today. Blackbird (February 2015), focused on performance and scalability optimizations. Hummingbird (September 2016) delivered important enhancements in areas of core control functions and automation and configuration of legacy networks. In its latest release, Kingfisher (June 2017), ONOS was enhanced among others in terms of Northbound and Southbound layer and

YANG data model tools. The ONOS ecosystem comprises the Open Networking Lab (On. Lab), organizations who are funding and contributing to the ONOS initiative including service providers, leading vendors and other community members.

ONOS has been designed aiming to fulfill the following goals: Code modularity, separation of concern, configurability, and protocol agnosticism [60]. Specifically, design principles define that it should be possible to introduce new functionalities as self-contained units. To achieve this, clear boundaries between subsystems should exist. Moreover, configurability is achieved by allowing loading of several features, on startup or runtime. Finally, it is expected that platform is not bound to specific protocols but should easily allow the introduction of modules that will allow communication based on different network protocols. More details are given in the following paragraphs.

The ONOS kernel and core services, are written in Java as bundles that are loaded into the Karaf OSGi container. Since ONOS runs in the JVM, it can run on several OS platforms. The ONOS platform is designed to support various application categories such as control, configuration and management applications. ONOS is architected with tiers of functionality. It is comprised of a set of sub-projects, each with their own source tree that can be built independently. Each subsystem implements a service and executed in application, core and southbound protocol layers. Its primary subsystems are: Device Subsystem (manages the inventory of infrastructure devices), Link Subsystem (manages the inventory of infrastructure links), Host Subsystem (manages the inventory of end-station hosts and their locations on the network), Topology Subsystem (manages snapshots of network graph views), Path Service (finds paths between infrastructure devices or between end-station hosts using the most recent topology graph snapshot, Flow Rule Subsystem (manages inventory of the match/action flow rules installed on infrastructure devices and provides flow metrics, and Packet Subsystem (allows applications to listen for data packets received from network devices and to emit data packets out onto the network via one or more network devices) [60].

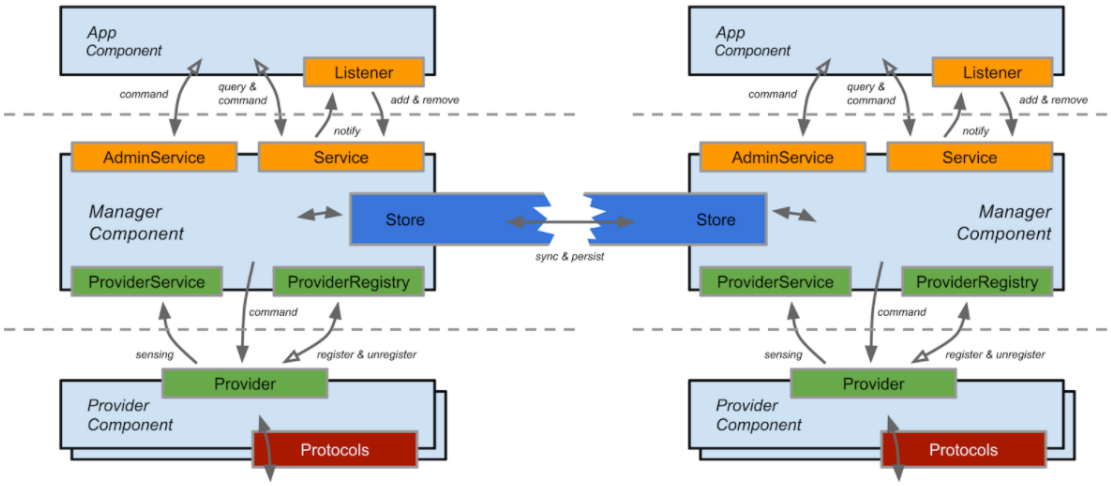

Figure 13: ONOS Subsystems [60]

Figure 13 represents the structure and inter-connection of subsystems. The ONOS core layer exposes the two interfaces, admin service and service, which are used by applications to invoke the different service components in the core. Each application registers to a core service, which in turn provides the application with a unique application identifier. This identifier is used by ONOS to keep track of tasks and objectives, such as intents and flow rules, associated with an application. In a similar way, the protocol-aware providers are responsible for interacting with the network environment using various control and configuration protocols. Providers also collect data from other subsystems to convert them into service-specific data. A provider is associated with a provider identifier. As in the case of the application identifier at the northbound interface, ONOS also uses this id, which is assigned to every provider in the southbound interface. The provider identifier serves the purpose of unique identification and proper mapping with the devices. Finally, from the subsystem perspective, multiple providers may be associated with a single subsystem. A device subsystem supports multiple providers. An example of a subsystem is the Intent Framework. It is a subsystem within the ONOS core. Intent allows applications to specify their network control preferences in form of policy rather than mechanism [60]. Intent is a model object that describes an application's request to the ONOS core to alter the network's behavior. These are described in terms of network resource, constraints, criteria and instructions. The framework mainly includes intent compilers that translate intents into installable intents that are more specific to the network environment, and

coordinators that determine how the network must be programmed, including the order of the installation at a device level. The framework includes translation and compilation, support for managing changes in network conditions and optimization across intents and other functionalities.

In ONOS, distributed-architecture support is also a key design principle. It can be deployed as a collection of controller-servers that coordinate with each other to achieve resiliency, fault-tolerance, and better load management. As in the case of traditional distributed architectures, there are various problems that need to be confronted to achieve the above. One of them is cluster coordination, which in the case of ONOS is achieved by including a distribution mechanism in the different subsystems which generate events. To ensure availability, ONOS uses methods such as vector clocks, distributed queuing and queue-sharing groups. Vector clocks is an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations [60]. To ensure consistency between events, link and host managements use optimistic replication technique and an anti-entropy mechanism, which is based on gossip-protocol and periodic probing of nodes. This mechanism is also used to confront cases of failure (i.e. node unavailability). Moreover, ONOS provides the network graph, and the view of the entire network, as the northbound abstraction. This global network information is presented as logically centralized, even though it is physically distributed across multiple servers. The global network view is built out of the network topology and state discovered by each ONOS instance, such as switch, port, link and host information.

ONOS targets support of multiple protocols (Openflow, NetConf, etc.), at the southbound interface to communicate with diverse devices, and expose APIs at the northbound interface to accommodate the needs of service provider use cases and application developers. If ONOS needs to support a new protocol, it should be possible to build a new network-facing module against the southbound API as a plugin that may be loaded into the system. ONOS, like other controllers (i.e. ODL), uses the concept of providers, which hide protocol complexity from other components of the controller platform. These providers offer all the necessary information of network elements to the core layer.

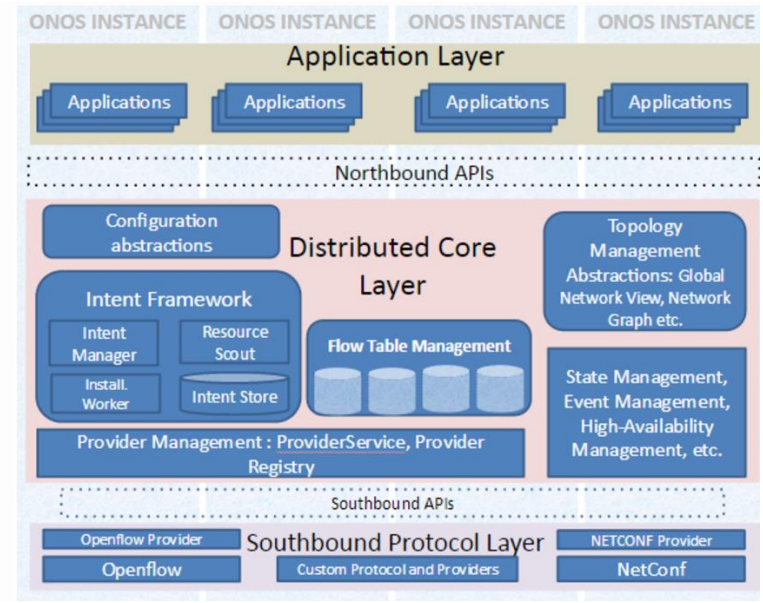

Figure 14: ONOS Architecture Diagram

4.3 Floodlight

Floodlight is a Java multi-threaded OpenFlow controller, initially based on the Beacon implementation. Its last version was released on March 2016. It is intended to be a platform for a wide variety of network applications. It is Apache licensed and is one of the significant contributions from Big Switch Networks to the open source community. Floodlight's architecture is based on the Big Network Controller (BNC), the company's commercial offering. It has been designed as a highly concurrent system, to achieve the throughput required by enterprise class networks and data centers. The controller consists of an autonomous collection of modules that perform Floodlight's main functions as an OpenFlow controller, as well as applications that are deployed on top of REST applications or run with the controller.

The Floodlight core architecture is modular, with components including topology management, device management, path computation, infrastructure for web access, counter store and a generalized storage abstraction for state storage. These components are treated as loadable services with interfaces that export state. Floodlight has been redesigned without the OSGi framework. The controller itself presents a set of extensible REST APIs as well as an event notification system. The API allows applications to get and set this state of the controller, as well as to subscribe to events emitted from the controller using Java. Applications using Floodlight as an underlying layer are deployed as independent Java units and use the REST API. Also, applications

can be deployed and configured at the level of the Floodlight controller. This approach has significant benefits in terms of application execution speeds, as well as offering more bandwidth communication with Floodlight, since the application-controller pairing is more immediate with the Java API.

The Floodlight OpenFlow controller can interoperate with any element agent that supports OpenFlow, but Big Switch also provides an open source agent that has been incorporated into commercial products. In addition, Big Switch has provided Loxi, an open source OpenFlow library generator, with multi-language support to address the problems of multi-version support in OpenFlow.

Floodlight can be run as a network plugin for OpenStack using Neutron. The Neutron plugin exposes a Networking-as-a-Service (NaaS) model via a REST API that is implemented by Floodlight. Once a Floodlight controller is integrated into OpenStack, network engineers can dynamically provision network resources alongside other virtual and physical computer resources. This improves overall flexibility and performance.

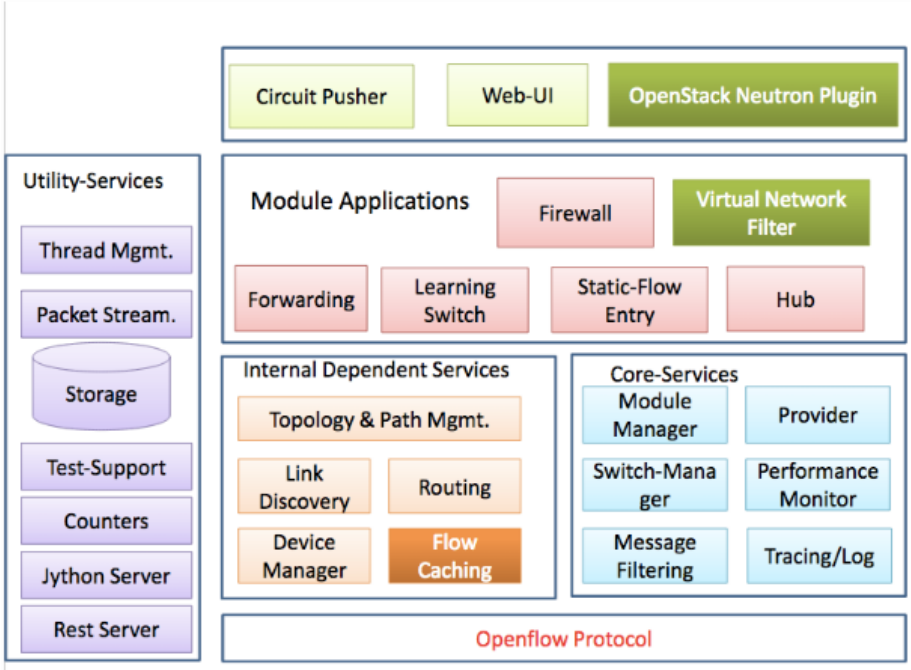

Figure 15: Floodlight Architecture diagram [37]

4.4 Trema

Trema is a programming framework for developing OpenFlow controllers that was originally developed and supported by NEC with subsequent open source contributions under a GPLv2 scheme (GNU General Public License), which is the most commonly used open source license. With Trema, the term framework is used to highlight the fact that the user has the freedom to configure and build an OpenFlow controller. The framework, designed to provide extensibility, includes all the necessary libraries and functionalities that are necessary to interact with OpenFlow. Unlike the more conventional OpenFlow-centric controllers that preceded it, the Trema model provides basic infrastructure services as part of its core modules that support the development of user modules. Trema is more a software platform for OpenFlow research than a production controller. Trema modules may be created either in C or Ruby. It provides a network emulator and libraries that can create simple OpenFlow based networks on a system. These features are an efficient way to provide development and testing environments for networks. According to [66], the appeal of Trema is that it is an all-in-one, simple, modular, rapid prototype and development environment that yields results with a smaller codebase.

The main API that the Trema core modules provide to an application is a simple, non-abstracted OpenFlow driver. Trema now supports OpenFlow version 1.3.X via a repository called TremaEdge. The base controller design is event-driven and is often compared to the explicit handler dispatch paradigm of other open source products. In addition, the core modules provide a message bus that allows the application modules to communicate with each other and core modules. Other core modules include timer and logging libraries, a packet parser library, and hash-table and linked-list structure libraries.

The Trema core does not provide any state management or database storage structure. These are contained in the Trema applications and could be the default of memory-only storage using the data structure libraries. The infrastructure provides a command-line interface (CLI) and configuration file system for configuring and controlling applications, managing messaging and filters, and configuring virtual networks. Network Domain Specific Language, a Trema specific language, is used for this purpose. There is also an OpenStack Quantum plug-in available for the sliceable switch abstraction.

The libraries in Trema can be categorized under multiple headings as listed below: protocol (i.e. OpenFlow), interfaces (i.e. OpenFlow application, switch, management), commonly used data structures (i.e. Linked list, doubly-linked list, hash table, timers), utilities (i.e. log, stats, wrapper), network protocols (i.e. CP, IP, UDP, ether and etherIP, ICMP and IGMP).

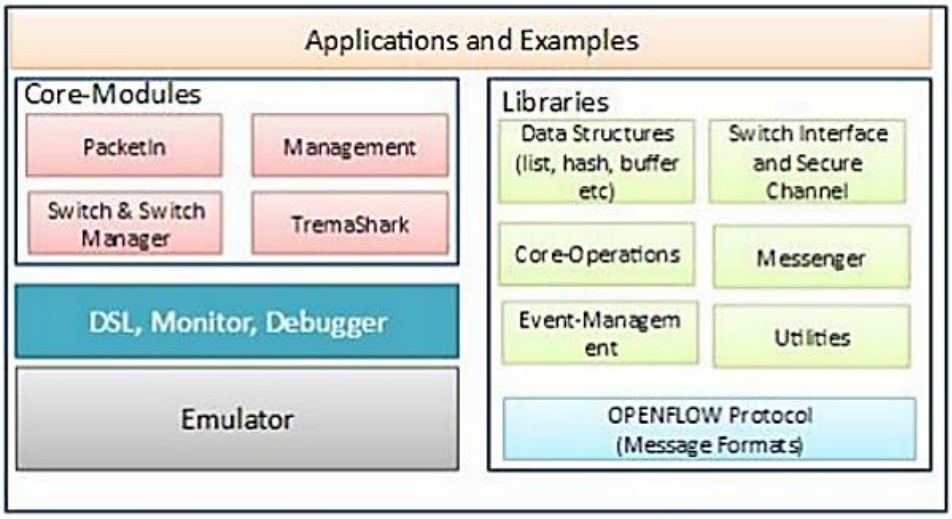

Figure 16: Trema Architecture diagram [37]

4.5 Ryu

Ryu is a component-based, open source (supported by NTT - Nippon Telegraph and Telephone Corporation Labs) framework implemented entirely in Python. It integrates with OpenStack and supports OpenFlow. It provides a logically centralized controller and a well-defined API that make it easy for operators to create new network management and control applications. The Ryu messaging service does support components developed in other languages. As in case of Trema, Ryu is more suitable for data centers, cloud infrastructures, and carrier grade networks.

Components include event management, messaging, in-memory state management, application management, infrastructure services and a series of reusable libraries [66]. Ryu applications are single-threaded entities, which implement various functionalities. Ryu applications send asynchronous events to each other. Each application has a receive queue for events to preserve the order of events. In addition, each application includes a thread for processing events from the queue. The thread's main loop pops out events from the receive queue and calls the appropriate event handler. Hence, the

event handler is called within the context of the event-processing thread, which works in a blocking fashion.

At the API layer, Ryu has an Openstack Quantum plug-in that supports both Generic Routing Encapsulation (GRE) based overlay and VLAN configurations. GRE is a tunneling protocol developed by Cisco Systems that can encapsulate a wide variety of network layer protocols inside virtual point-to-point links over an Internet Protocol network. OpenStack Neutron is a cloud networking controller and a networking-as-a-service project within the OpenStack cloud computing initiative. Neutron includes a set of application program interfaces (APIs), plug-ins and authentication/authorization control software that enable interoperability and orchestration of network devices and technologies within infrastructure-as-a-service (IaaS) environments.

Ryu has an impressive collection of libraries, ranging from support for multiple southbound protocols to various network packet processing operations. With respect to southbound protocols, Ryu supports OF-Config, Open vSwitch Database Management Protocol (OVSDB), NetConf, SFlow (Netflow and Sflow) and other third-party protocols. Netflow is supported by Cisco and others and is specific to IP. The third-party libraries include Open vSwitch Python binding, the Oslo configuration library and a Python library for the NetConf client. The Ryu packet library helps you to parse and build various protocol packets, such as VLAN, MPLS, GRE, etc.

Ryu supports the OpenFlow protocol up to the latest version. It includes an OpenFlow protocol encoder and decoder library. Just like any SDN controller, Ryu can also create and send an OpenFlow message, listen to asynchronous events such as “flow removed”, and parse and handle incoming packets.

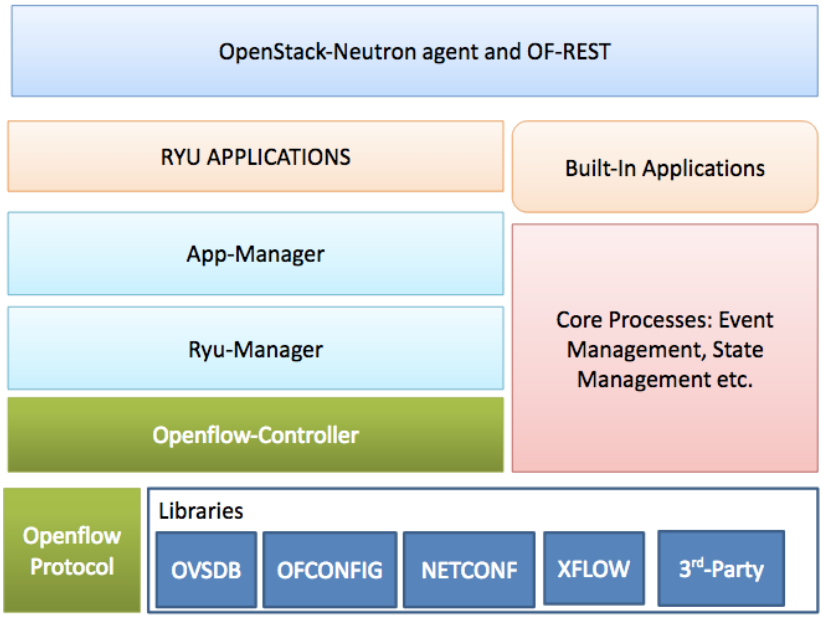

Figure 17: Ryu Architecture diagram [63]

4.6 Maestro

Maestro is an OpenFlow controller introduced in 2010 as an operating system for orchestrating network control applications [62]. It is developed in Java, which makes it highly portable to various operating systems and architectures. Maestro provides interfaces for implementing modular network control applications to access and modify the state of the network, and coordinate their interactions. Maestro is licensed under the GNU Lesser General Public License version 2.1. According to [62] Maestro, was designed as a simple single-threaded programming model for application programmers of the system. However, it enables and manages parallelism as a service to application programmers [56].

Maestro provides a view abstraction for grouping related network state into a subset, and for accessing the state in that subset. Each view is a Java class that can be dynamically created in Maestro. A view can contain any data structure to represent a particular subset of network state. For example, a view may be a hash table structure that holds all pair shortest path routing information for the network. The view is the minimal granularity at which Maestro synchronizes the concurrent execution of control components. Views are highly related to events in Maestro. An event is the basic data unit to exchange information between the underlying network and the control

components. Events can be generated from the switches in the underlying network. Events can also be generated by the control components, to affect the behavior of certain switches in the network. Each network control component is represented as one application in Maestro, which is a Java class that contains the code for the control function. All applications must extend the base abstract class application, and interact with Maestro via a simple and straightforward API.

Maestro was initially designed to meet OpenFlow requirements. Therefore, it offers similar functionalities as other OpenFlow controllers. In Maestro's case, low level components that remain fixed for a specific version of OpenFlow handle the details of reading and writing to socket buffers, and translating OpenFlow messages into and from high level data structures.

Maestro is a controller, which uses task batching so that worker threads pull a batch of tasks, so as to process multiple flow-requests in a single execution. The output batching technique is used to send packets out in which packets belonging to the same destination are grouped together and sent using a single socket system call [39]. Maestro handles most of the tedious and complicated job of managing work load distribution and worker threads scheduling.

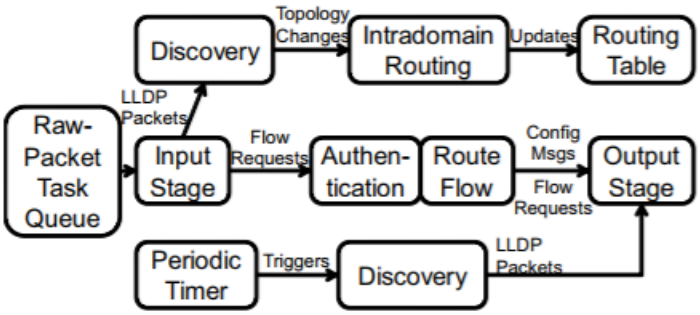

Figure 18: Maestro Architecture diagram [62]

4.7 Beacon

Beacon is an OpenFlow controller developed in Java and released in 2010. It has been used in several research projects, networking classes, and trial deployments [13]. It is a multi-threaded controller and considered more suitable for enterprise class networks and data centers. It runs on many platforms, such as Linux servers and Android phones. Beacon is licensed under a combination of the GPL v2 license and the

Stanford University FOSS License Exception v1.0. Today, Beacon is considered inactive.

Beacon's basic architectural component is the bundle. Bundles are jar files that may contain metadata, java classes, resources and others. Bundles share their resources, consume other packages, extend other bundles and run code. This architectural approach, makes Beacon a modular and configurable implementation since code bundles and are independent and can be installed at startup or at runtime. Beacon is provided with all the basic code bundles, such as OpenFlow OpenFlowJ (OF 1.0 Protocol) for the OpenFlow protocol, bundles for packet encoding and decoding (Ethernet, ARP, IPv4, LLDP, TCP, UDP), bundles for basic switch operations (Core, Learning Switch, Hub, 28 Device Manager) as well as topology and Layer 2 Shortest Path Routing clusters.

To maximize code re-use and facilitate development, Beacon leverages multiple libraries. The most significant library is Spring. Two main components of Spring are used in Beacon: the Inversion of Control (IoC) container, and the Web framework. Beacon uses Spring's IoC framework for wiring within and between applications. Spring's Web framework is used to map Web and REST requests to simple method calls, and to perform auto conversion of request and response data types to and from Java objects. IoC frameworks allow developers to list in an XML file or as Java annotations, which objects to create, how they are wired together, and then provide methods to retrieve the resulting objects [34]. As mentioned above Beacon, includes also the OpenFlowJ library for working with OpenFlow messages.

Beacon's API is designed to be simple and to impose no restrictions as far as available Java constructs, such as threads, timers, sockets, etc. are concerned. The API for interacting with OpenFlow switches is event based.

Beacon supports both event-based and threaded operation. It uses a static approach in which a fixed number of switches are assigned to a worker thread. Worker threads use static packet batching to serve the requests from the connected switches [39]. Once the packets are processed and ready to be sent, Beacon in its default mode uses write coalescing and allows only one write per I/O select loop to reduce the overhead of socket system calls for each individual OpenFlow message. Alternatively, an immediate mode can be enabled in which the controller attempts a socket write for every outgoing OpenFlow message waiting to be written to the switch to reduce per-

packet latency. Static partitioning and input batching improve its throughput in the default mode.

4.8 NOX

As already mentioned in paragraph 3.1, NOX was the first SDN controller. It was initially developed by Nicira Networks and was the first to support OpenFlow protocol. It was given to the research community in 2009 and has been the basis for many research projects on SDN. It was subsequently extended and supported via On.Lab activity at Stanford University with major contribution from the University of California, Berkeley. Some popular NOX applications are SANE (an approach to represent the network as a file system) and Ethane. Today NOX is considered inactive. Different versions of NOX have been introduced during these years. These are new NOX, NOX-MT and POX. New NOX only supports C++. It has fewer network applications compared to NOX, but is much faster and has a much cleaner codebase. NOX-MT, introduced in [2] is a slightly modified version of the NOX controller that uses optimization techniques to introduce multi-threaded processing and to improve the rate and response time of NOX. These optimization techniques include I/O batching to minimize the overhead of I/O and others. POX is the newer, Python-based version of NOX. The idea behind its development was to return NOX to its C++ roots and develop a separate Python-based platform. It also has a Python OpenFlow interface, reusable sample components for path selection, topology discovery, and others. The primary target of POX is research. Since many of the research projects are short-lived by nature, the focus of the developers of POX is on good interfaces rather than maintaining a stable API [11]. Generally, the NOX controller provides a full OpenFlow API using C ++ and Python languages, uses asynchronous inputs / outputs (I/O) and is geared to operating on Linux systems, Ubuntu and Debian.

NOX is used both as a stand-alone controller and as a component-based framework for developing SDN applications. It is built on an event-based programming model and adopts a simple model of programming interfaces that revolves around three pillars: events, namespace, and network view. Events may be generated either directly by OpenFlow messages or by NOX applications a result of processing low-level events or other application-generated events [63]. As far as namespaces and the network view are

concerned, NOX consists of number of basic applications that construct the network view and maintain a high-level namespace that can be used by other applications. These applications are responsible for functionalities such as user and host authentication. The inclusion of high-level names and their bindings in the network view allows any application to convert a high-level name into low-level addresses, allowing applications to be written in a topology independent manner. To perform such conversions, high-level declarations can be “compiled” against the network view to produce low-level lookup functions that are enforced per-packet [63]. This architecture allows the writing of applications for NOX regardless of the topology and form of the underlying network. Due to the fact that the network database must be consistent and available in all the processes of the controller, records and write-offs from it bring some cost to the controller's speed and performance. Thus, applications in NOX need to register in the network database only when a change in the network is detected and not for every received package from the application. In the case of NOX, as well as computer systems, a malicious application, in terms of access to the database, can lead to a serious downgrade of the overall performance of the network.

Large-scale networks have a very variable character and are not static, unlike home networks. Flows may be recorded and deleted by the switches, users may enter or leave the network, the various links may fail or be reconnected, and devices may be added or removed from the network. An event represents a low-level or high-level event in the network. Typically, the event only provides the information. Information process is deferred to handlers. Many events roughly correlate to something which happens on the network that may be of interest to a NOX component. These components, typically, consist of a set of event handlers. NOX events can be broadly classified as core events and application events (do not apply for all NOX versions). Core events map directly to OpenFlow messages received by controlled switches. In addition to core events, components themselves may define and throw higher level events (application events), which may be handled by any other events. Though NOX does not contain any such application events, considering it has a minimal set of applications, NOX classic has various events such as host event and flow in event by authenticator application and link event by the discovery application.

NOX mainly provides support modules specific to OpenFlow. The NOX core provides helper methods and APIs for interacting with OpenFlow switches, including a connection handler and an event engine. Additional components are available,

including host tracking, routing, topology (LLDP), and a Python interface implemented as a wrapper for the component API. Programmatic interfaces also exist for control over the network and high level services.

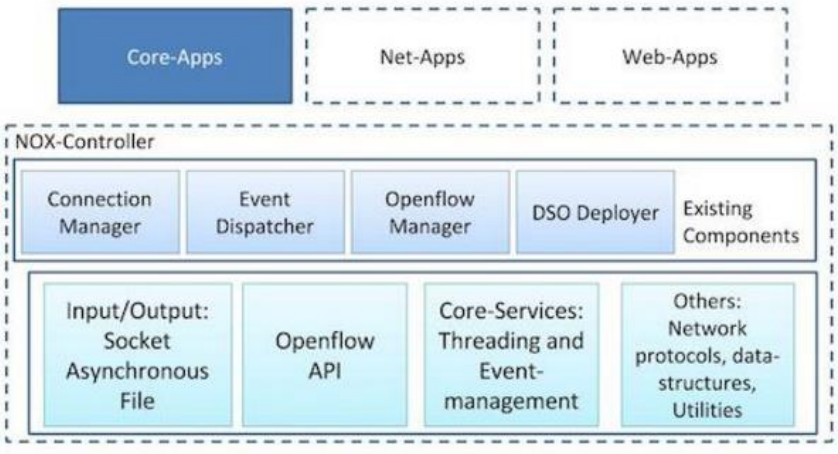

Figure 19: NOX Architecture Diagram

4.9 Summary

As a last step, a summary table presenting the basic features of the above described open source controllers is provided. The chosen criteria concern southbound and northbound communication, OpenFlow and OpenStack support, programming language and others.

- As can be observed, most controllers have a Web based GUI.
- Most controllers are centralized and support multi-threading.
- Most of the controllers under consideration are implemented in Java, due to the fact that it may be used across all platforms and architectures and that it supports memory management.
- As expected, no specific protocol is supported as far as Northbound interface is concerned. Only the most recent implementation support REST API. ODL presents multiple interfaces, such as, the Yang interface and supports Group Based Policies that can be accessed by a REST API. On the other hand, ONOS presents an Intent Framework with Graphical User Interface (GUI) and a REST API, among others.

- All controllers support OpenFlow. However, it should be noted that especially not active controllers (i.e. NOX) support only specific versions of OpenFlow.
- As far as other southbound protocols are concerned, ODL is the controller which is compatible with the widest variety of protocols.

Controller / Attribute	ODL	Floodlight	Trema	Ryu	Maestro	Beacon	ONOS	NOX
Year	2013	2013	2011	2013	2010	2010	2014	2009
GUI	Web based	Web based Java	N/A	Y	N/A	Web based	Web based	Pythons +qt4
Programming Language	Java	Java	C Ruby	Python	Java	Java	Java	C++
Platform Support	Linux MAC Windows	Linux MAC Windows	Linux	Linux	Linux MAC Windows	Linux MAC Windows	Linux MAC Windows	Linux
OpenStack	Y	N	Y	Y	N	N	Y	N
Southbound APIs	OpenFlow NetConf BGP PCEP CAPWAP LISP SNMP OVSDB	OpenFlow	OpenFlow	OpenFlow NetConf OVSDB SFlow	OpenFlow	OpenFlow	OpenFlow NetConf	OpenFlow
Northbound APIs	Rest API	Rest API	Ad-hoc API	Rest API	Ad-hoc API	Ad-hoc API	Rest API	Ad-hoc API
Multithreading Support	Y	Y	Y	N	Y	Y	Y	N
Centralized	N	Y	Y	Y	Y	Y	N	Y
Modularity	high	medium	medium	poor	poor	poor	high	poor
Documentation	good	medium	poor	medium	poor	poor	good	medium

Table 4: Open Source Controllers Summary Table

5. Controller Efficiency

As concluded from the preceding analysis, there are currently several controllers created by different vendors, universities and research groups that aim at fulfilling SDN's advantages such as reduced complexity, high performance, security and others. In some cases, controllers are under continuous development that result in different controller versions. Due to the above and the importance of the controller within the SDN architecture, the need to assess and benchmark controllers against different efficiency indicators has arisen. Controller efficiency is a term used to refer to the different parameters such as performance, scalability, reliability and security that characterize a controller. Almost all existing studies focus on one or more of these parameters. For example, in different studies performance is defined by various metrics, latency, throughput, etc. Similarly, there are various metrics used in existing works defining scalability and security. Comparison is performed by using several methods and tools such as controller benchmarking, which is a commonly used procedure for the performance analysis of SDN controllers.

The current chapter elaborates on the main attributes of controller efficiency: performance, scalability and security. Their basic principles are presented along with respective research efforts.

5.1 Performance

Performance is the most commonly tested attribute of an SDN controller. It is related to flow establishment (or set up) time. This is the process of determining the action a switch should perform for a specific flow of packets. The above process might be a bottleneck if we take into consideration that controllers need to handle many flows initiated from many network elements. Therefore, the number of flow setups per second an SDN controller can support is important. The most common performance metrics for an SDN controller in the literature are throughput and latency. Throughput is the number of transactions per second that an application can handle. There are different forms of measuring this parameter such as bits per second or packets per second. Latency is the time required for a packet to arrive at its destination through the network. Latency may be measured either using the time needed for a packet to reach its

destination or the time round-trip time. The latter is the most commonly used. The goal is to obtain maximum throughput and minimum latency.

In case of SDN and particularly controllers, the above metrics are influenced by many factors. For example, the number of controllers used to handle an amount of traffic, the processing power of the switches that are attached to the controller, the I/O performance of the controller affect performance. Another important factor is the controller's operating mode, which can be set at proactive or reactive. In the first case flow tables are prepopulated. Therefore, there is no setup delay and no limit on the number of flows per second that the controller can support. In the case of the reactive mode, this processes flows that do not match to a specific flow table entry. In this case delays are expected.

Understanding the performance of the SDN controller is a requirement for its implementation in production networks. As already mentioned, many existing studies focus on the characteristics of controller performance. In most cases simulation and experimentation are used as performance evaluation techniques. There are also studies that have used analytical modeling to conclude whether one or more controllers are enough to ensure high performance.

Currently, controller vendors implement several techniques to handle performance issues since there is a need for the controller to be able to support different loads in different environments. However, performance should not be achieved at the expense of other capabilities, such as security or modularity.

5.1.1 Benchmarking

As already mentioned, benchmarking is a commonly used procedure for the performance analysis of SDN controllers. In general, benchmarking is defined as the act of running a computer program or other operations, in order to assess the performance of an object. It is used in both hardware and software. In the context of SDN controllers, benchmarking has been used from researchers to evaluate controller's performance.

The most popular OpenFlow controller benchmark is Cbench that was introduced by R. Sherwood et al [56]. It is a program for testing OpenFlow controllers by generating events for new flows. In more detail, Cbench emulates several switches which connect to a controller and send packet-in messages. It supports latency and

throughput modes. Latency mode measures the OpenFlow controller's request processing time under low-load conditions. In latency mode, each switch maintains exactly one outstanding new flow request, waiting for a response before soliciting the next request. In throughput mode, each switch maintains as many requests as buffering will allow. Thus, throughput mode measures the maximum flow setup rate that a controller can maintain. Cbench has been reused or in some cases enhanced by researchers to identify performance improvements for OpenFlow controllers. According to [56], conclusions may be based on different environment and system configurations (i.e. number of forwarding devices, network topology, network workload, type of equipment).

Except for Cbench, other frameworks have been used to facilitate the evaluation of performance aspects of controllers. Examples are Hcprobe and EstiNet. EstiNet, introduced in [6], simulates a network, where each host uses the real-world Linux operating system. It supports two modes. In simulation mode, an open source OpenFlow controller can directly run on a node in the simulated network to control switches. In emulation mode, the controller application can run on an external machine, different from the machine used to simulate OpenFlow switches. EstiNet can accurately simulate the properties of the links that connect simulated OpenFlow switches. These properties include link bandwidth, link delay, link downtime, medium access control and others. Hcprobe, introduced in [1], is written in Haskell and emulates OpenFlow switches and hosts, which are connected to a controller. Its main features include packet generation, an API for custom tests design and the introduction of a domain-specific language that is used for creating tests.

5.1.2 Related Work

Most comparative studies regarding SDN controllers refer to performance. Most of them use Cbench or other benchmarking tools to assess metrics related to throughput and latency. In most cases, popular OpenFlow controllers have been chosen since they are open-source and helpful documentation exists. In the following paragraphs, a summary of related work may be found. What should be noted is that in some cases in the literature, performance and scalability are considered identical. Therefore, some of the below experiments refer also to scalability, that is going to be further analyzed in the following paragraph.

In [46] the performance of the SDN controllers POX, Ryu, ONOS and OpenDaylight has been measured using the Mininet tool. Mininet is a network tool which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and SDN. In this case performance was indicated by the Round Trip Time (RTT) of ICMP packets between two hosts. This refers to the interval from the moment a message is dispatched from the switch to the controller until the corresponding message is received by the switch. A tree topology consisting of 4 layers of Open Virtual switches and 16 hosts was used. Two modes of switches were used: Hub mode (every packet is flooded to all ports), and L2 Switching mode. In the latter case, the SDN controller when receiving a packet, associates the source MAC address with the switch port from which the packet arrived. In both modes, performance tests were conducted using the ping and iperf (measures the maximum network throughput a server can handle) commands between specified nodes. The main conclusions regard both the type of topology and the type of controller used each time. In hub mode, RTT differentiations among controllers are minor due to the rather simple topology. In switch mode, the minimum RTT was observed for ONOS while the maximum for POX. ONOS performance has also exceeded the other controllers in the case of the iperf tests.

In [39] the performance of NOX, Beacon, Maestro and Floodlight was measured in terms of traffic handling. Firstly, 32 switches (100.000 MAC addresses per switch) that sent packet to a controller were simulated. The maximum throughput in this case was exhibited by Beacon due to its partitioning and batching technique. Maestro's throughput was lower since it employs adaptive-batching technique and improves its latency by compromising its throughput performance. Floodlight had the lowest performance. As a next step, performance was measured by simulating various numbers of switches. In this case, decreasing performance as the number of switches increased was observed. Beacon had again the best performance because of its static switch partitioning, packet batching technique and low synchronization overhead. To measure latency, a constant number of threads was used. In this case, Maestro had the best performance due to its workload adaptive batching technique that dynamically changes the batch sizes. NOX-MT, Beacon and Floodlight showed a degradation in performance compared to Maestro because of their static packet batching design. Based on their findings, writers propose that controllers designed for high throughput should use static switch partitioning and packet batching. Moreover, controllers designed for delay-

sensitive control plane applications should use workload adaptive packet batching and task batching to reduce per-packet latencies. Finally, to further improve latency, controllers should send each outgoing control message individually.

In [45] ONOS, Floodlight, ODL, POX and RYU are compared in terms of throughput and latency. Throughput was measured by usinh a varied number of switches with a fixed number of hosts. Then a varied number of hosts was used. In this case, ODL achieved the highest performance. It also had the most variation regarding the increase of the number of switches, while the other three controllers achieved similar results around the maximum value. In general, the increase in the number of hosts had close to no impact in the number of responses. Finally, what is also pointed out is that in case of ODL and ONOS, some tests returned a considerable amount of failed tests. Latency was also measured in a similar way. On the one hand one switch with a fixed number of hosts was simulated. On the other a varying number of switches was used. When the number of switches increased, ODL clearly achieved better results compared to the other three controllers. In general, Ryu and ODL stood out, followed closely by ONOS. On the other hand, POX showed higher latency. In case of switch scalability, ODL achieved the best results when emulating four or more switches while the other controllers had a similar performance, despite the high rate of flows per second by Ryu for a lower number of switches. Finally, regarding switch scalability throughput tests, ODL clearly achieved better rates than any other controller tested while POX placed second.

In [23] the aim is to evaluate NOX, Floodlight, POX and Trema controllers in both reactive and proactive mode. As already mentioned in this thesis, in proactive mode the path from source to destination is pre-defined and there is no need for the controller to build it upon switch triggering. A real and a simulation network (through Mininet) are used for testing. A varying number of switches has been created to test controller performance. As expected, all controllers had better performance (reduced with increasing number of switches) when running in proactive mode. The improvement is due to the fact that a proactive controller receives less request messages from switch because the path is already set. However, it should be noted that the proactive approach requires the controller to know the traffic flows in advance to configure the paths before it is used. The reactive approach reduces controller performance but requires less configuration effort.

In [26] Floodlight and OpenDaylight controllers are compared in terms of delay and loss in different topologies (single topology, linear, tree) and network loads (zero, half of the bandwidth, all the bandwidth. The results showed that OpenDaylight outperforms Floodlight in low loaded networks and for tree topologies in mid-loaded networks in terms of latency. Maximum latencies in all three topologies belong to Floodlight, but there is no significant difference between the results for minimum latency. It is noticeable that the maximum latency in Floodlight is much more than in OpenDaylight. This means that Floodlight needs more time to find the route and send a decision for newly flows. Networks that use Floodlight as their controller have more latency while the traffic is low. For mid-loaded networks, in single and linear topologies, there is no difference in the latency results for Floodlight and OpenDaylight.

In [22] NOX, Maestro, Floodlight, Beacon are under evaluation. The authors focus in a different aspect of performance, which concerns datacenters and the way controllers are implemented on multi-core processors. These are compared to many-core processors, where many simpler cores are used. Having this in mind, throughput, latency and performance per Watt (through respective tools) are measured. What should be noted in this case is that the controller responds to packet-in messages reactively. Their main conclusion is that the performance of existing controllers does not truly leverage the concurrency level of modern machines. The performance results obtained on the many-core platform represent a lower portion of network capacity than on the multi-core platform. As far as power consumption is concerned, measurements indicate that controllers also fail in exploiting the energy savings of multi-core processors.

In [1] NOX, POX, Beacon, Floodlight, MUL, Maestro and Ryu are examined in terms of throughput and latency. Throughput is measured with varying switches, hosts and CPUs. It should be highlighted that the performance of POX and Ryu does not depend on the number of switches, since they do not support multi-threading. For multi-threaded controllers, adding more switches leads to better utilization of available CPU cores, so their throughput increases until the number of connected switches becomes larger than the number of cores. The results of performance tests showed that Beacon controller achieved and had a potential for further throughput scalability when the number of CPU cores increased. The scalability of other multi-threaded controllers was limited by 4-8 CPU cores. Another conclusion was that the number of hosts in the network was irrelevant. Specifically, the average response time of the controllers demonstrated insignificant correlation with the number of connected hosts. The average

throughput achieved with different number of switches influenced the performance of most of the controllers. This was caused by the specific details of the Learning Switch application implementation, namely, the implementation of its lookup table. Maestro's scalability is limited to 8 cores, as the controller does not run with more than 8 threads. The writers conclude that the difference in throughput scalability between multithreaded controllers depends on two major factors: the first one is the algorithm for distributing incoming messages between threads, and the second one is the mechanism or the libraries used for network interaction. Beacon shows the best scalability. NOX, MuL and Beacon pin each switch connection to a certain thread. This strategy performs well when all switches send approximately an equal number of requests per second. Maestro distributes incoming packets using the round-robin algorithm, so this approach is expected to show better results with an unbalanced load. Due to the round-robin packets distribution algorithm, Maestro shows better results on a small number of switches, but the drawback of this approach is that its performance decreases when many switches (256) is connected. Floodlight relies on the Netty library, which also associates each new connection with a certain thread. To measure latency a single switch is used. The switch propagates requests to the controller only after a reply to a previous message has been received. The smallest latency was demonstrated by the MuL and Beacon controllers, while the largest latency is typical of python-based controller POX. The minimum response time was observed in the case of MUL and Beacon

In [4] the tests were conducted for these controllers: ONOX, NOX, Beacon, Maestro, Ryu, Libfluid_Raw, POX, MuL, Floodlight, IRIS, Libfluid_Msg and ODL. Performance concerned throughout and latency with a varying number of switches and threads bound to the controller instance. The results showed that the controllers coded in the C language offered the highest performance (Mul, Libfluid_msg) and the next best performance was offered by the Java coded controllers (Beacon, Iris and Maestro). Furthermore, in latency mode, Maestro, using the adaptive batching mode, gave the best latency results. In addition, it was noted that controllers coded in C and Java offer higher performance when varying the number of threads. However, POX, coded in Python, does not show any significant difference when using multithreading because Python's support of multithreading is not very efficient.

5.2 Scalability

Scalability is one of the major concerns for both traditional networks and SDN. Scalability, as a network property, is generally difficult to explain and in some cases it is necessary to define the specific requirements to conclude on a system's scalability. Generally, scalability is the capability of a network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth. The traditional LAN networks that form a multi-tier architecture do not scale in an optimal way, especially when it comes to east-west traffic. In the case of SDN, scalability concerns the decoupling of the control and data planes. It is obvious that controller scalability is directly related to their performance. The changes in throughput and latency when adding more switches and hosts to the network or adding more CPU cores to the server where the controller runs define the scalability of the controller.

An SDN enables IT organizations to move to a scale-out model of networking whereby they add networking functionality when needed, and the SDN controller enables them to manage the networking functionality as if it was one device. In terms of the SDN controller, this needs to effectively scale to control extra devices and meet the demands of a distributed environment. There are several factors affecting controller scalability. One of them is the number of switches that the controller can support. Another is the fact that network broadcast overhead will limit the scalability of the solutions implemented. Moreover, controller scalability may be limited by the proliferation of flow table entries. The controller architecture should also allow for horizontal scaling in clustered and cloud environments. Finally, another aspect of scalability is the ability of the SDN controller to create an SDN that can span multiple sites. This capability enables the movement of VMs and virtual storage between sites. Run-time extensibility is also needed to avoid the lengthy release cycle of legacy systems. In terms of expandability, the controller architecture must allow for plugins to be developed independently of each other and of the controller infrastructure, and it must support short system integration times. For example, in cases of open source controllers (i.e. ODL), there can be many active and proposed projects concurrently. These projects need to be autonomous, since they and are being developed by independent teams, with little coordination between them.

5.2.1 Related Work

In [12] solutions regarding SDN scalability are proposed. Most of the research efforts addressing scaling limitations of SDN can be classified in three categories: data plane, control plane, and hybrid. While targeting the data plane, proposals such as DevoFlow and Software-Defined Counters (SDC) reduce the overhead of the control plane by delegating some work to the forwarding devices. In terms of controller solutions, DIFANE, Onix, HyperFlow, Kandoo, and others are mentioned. These are considered efforts for designing and deploying high performance controllers since techniques such as buffering, pipelining and parallelism are used. Also related to scalability, the notion of elasticity in SDN controllers is mentioned. Elastic approaches refer to the dynamic change of the number of controllers and their locations based on traffic requirements. Finally, hybrid solutions that aim at splitting scalability between controllers and network elements are proposed. In these cases, authoritative switches are placed between controllers and network elements.

In [108] emphasis is given on the modelling of scalability and a metric is proposed to quantify scalability. Three control plane structures (hierarchical, centralized, decentralized) are studied by implementing numerical experiments. It is concluded that the scalability of hierarchical and decentralized structures outweighs the others.

5.3 Security

Today, traditional networks are considered vulnerable in terms of security. Many areas such as infrastructure, software and network protocols may be affected by external intrusions. Security issues multiply due to growth of Internet throughput, to mobile networking development and others. Thus, network solutions have become more complex. Malfunctions in these areas result to the violation of basic security principles, confidentiality, integrity and availability. In thecase of SDN solutions, despite proven advantages in areas such as performance, modularity and programmability, security is currently an area where more research and improvements are needed. According to [12], there are many areas in SDN that are prone to security incidents. Some of them are: faked traffic flows in the data plane, which can be used to attack forwarding devices and controllers, vulnerabilities of network devices that may consequently affect the whole network, attacks on the control plane and network applications that could re-

program the entire network and, finally, attacks in administrative stations. Researches have shown that SDN networks are subject to a variety of security problems such as spoofing, tampering, denial of service and others.

Due to the centralized design of SDN, a secure controller would result in the security of a whole network. The controller is considered the main component of SDN, providing multiple levels of access to network users, applications, and devices based on requirements. In addition, there are multiple critical data structures or data stores comprised by the controller. These data stores may include topology information, user details, access control, and policy information. For the above reasons, the controller remains a potential high value target of the security attackers. Initially, the fact that a centralized controller could end up to be a single point of failure was the main concern in terms of security and reliability. However, with the evolution of SDN, the vulnerability of the centralized controller is no longer the main security issue. Controllers have a set of processes which implement and perform important networking functions (i.e. topology management, load balancing, authentication mechanism, access control, etc.). All these may be affected from a security incident. Other issues of concern are the integration with network applications policy conflict resolution between multiple applications in the SDN, and the complexity of multiple controller deployments [12]. Moreover, communication channels between a switch and a controller or between an application and a controller are prone to incidents.

To sum up, security for an SDN controller may be translated into several requirements. Firstly, encryption of communications is expected. Moreover, to provide security to the network, an SDN controller must be able to support authentication, authorization and other types of administrative controls. Multi-tenant isolation is another example of security. This is the capability of the controller to ensure that each tenant that is sharing the infrastructure has complete isolation from all the other tenants. Finally, SDN controllers need the ability to respond to malicious attacks (i.e. Denial of Service). This could be achieved with a modular structure in which individual modules are responsible for certain functionality and the main process is offloaded. However, in designs where the core process of a controller acts as a single monolithic module, it becomes a bottleneck, and if not handled properly, may lead to a DoS attack. This might be achieved by being able to both rate-limit the control communications and to alert at a real-time the network administrators regarding the event

Since no controller implements all the security features, [48] recommends a few best practices which we think are necessary for controller security. These are isolation of the user process, monitoring of resources, intrusion detection system, secure communication channel (recent versions of TLS should be used), use of safe mode or partial restart (for recovery purposes) and conflict resolver. Monitoring of resources refers to both Northbound and Southbound interfaces and is proposed to pushed on to data path for an earlier detection. According to [48], monitoring could be extended to a system, which could monitor malicious traffic or system upon notified by a monitoring agent and neutralize the same upon detecting the anomaly. A conflict resolver is necessary for a system if it participates in a multiple-domain or multiple-controller network. There could be a conflict of trust between switches, domains, and controllers. Currently, ODL and ONOS have a flow conflict resolver based on the user role. If there is a flow conflict between two users, then a user with higher privilege overrides the other change. However, this needs to be further extended to controller level conflicts.

There is no single SDN controller that currently delivers security, robustness, and resilience in parallel. [47] By secure, robust, and resilient, it is meant that the controller is designed to reduce the risk of intrusion/attack at the network control layer, can withstand errors in control layer logic, and can recover quickly from disruption and maintain an acceptable level of service in the face of faults.

5.3.1 Related Work

Although initially, the security domain was not analyzed in depth, several comparative studies have been published during the last years. Different methodologies are used for this purpose. One of the most popular ones is the STRIDE method (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege). This is a threat classification model developed by Microsoft, used to help reason and find threats related to an information system. As in the case of performance, OpenFlow controllers have been chosen since they are open-source and helpful documentation exists. In the following paragraphs a summary of related work may be found.

In [1], one of the first attempts to review the security of SDN controllers has been made. NOX, POX, Beacon, Floodlight, MUL, Maestro and Ryu were taken into consideration. Controller security was examined by using malformed OpenFlow

messages that are sent to the controller to test its reaction and tendency to failure. The expected behavior for the controller when receiving a malformed message is to ignore it and remove it from the incoming queue, without affecting other messages. If the controller processes the malformed message without indicating an error, this could possibly cause incorrect network operation. Malformed OpenFlow header and body messages have been generated for this purpose. In the case of the header, tests with invalid packet length, OpenFlow version and message type were conducted. In the case of the body, protocol type and port status were modified. To examine how the controller parses the queue of incoming messages, the field denoting packet length was modified to depict different values. Ryu was the only controller whose operation was not affected in this case. In case of invalid OpenFlow version, POX, MuL and Ryu closed the connection with the switch upon receiving a message with invalid protocol version. Other controllers process messages regardless of an invalid protocol version. Similar behavior was observed in the case of an incorrect message type. In general, these tests highlighted several possible security vulnerabilities of the tested controllers.

In [47] a detailed analysis of ONOS, ODL (Helium version), Rosemary, Ryu and SE-Floodlight (Security Enhanced version) is attempted. Based on documentation and code verification, security attributes (figure) have been divided into three categories: Secure controller design, Controller Interfaces and Security Services.

Controller	ONOS	OpenDaylight (ODL)	ROSEMARY	Ryu	SE-Floodlight
IV. Secure Controller Design					
- A. Control Process (Application) Isolation	x	x	✓(micro-NOS)	x	✓(Privilege-Based)
- B. Implementation of Policy Conflict Resolution	✓(Data-Store)	x	x	x	✓(Algorithm)
- C. Multiple Controller Instances - Resilience	✓(Clustering)	✓(Clustering)	x	x	x
- D. Multiple Application Instances - Resilience	x	x	x	x	x
- E. Secure Storage	✓	✓	✓	✓	✓
V. Secure Controller Interfaces					
- A. Secure Control Layer Communication	x	✓(D-CPI)	x	✓(D-CPI)	✓(D-CPI, A-CPI)
- B. GUI/REST API Security	x	✓(weak)	n/a	x	x
VI. Controller Security Services					
- A. IDS/IPS Integration	x	✓(Defense4All)	x	✓(Snort)	✓(BotHunter, Sec. Actuator)
- B. Authentication and Authorization	x	✓	✓	x	✓
- C. Resource Monitoring	x	x	✓	x	x
- D. Logging/Security Audit Service	✓	✓	✓	✓	✓

Figure 1: [79] Security Attributes of SDN controllers

Control process isolation refers to the ability of separating the application processes running at the controller to provide logical segmentation, to support authentication of individual applications and to apply levels of authorization. As demonstrated in the figure, Rosemary and SE-Floodlight, cover the process isolation requirement. In

Rosemary each application is run within an independent instance of Rosemary effectively sandboxing the application to protect the control layer from any vulnerability or malicious operation of the application. SE-Floodlight uses the northbound API to separate the application and control processes. Privilege-based OF operations are enforced with privileges defined by Role and Group Access Control Lists and credential files for internal Java-based Floodlight modules. Groups define permissible OF operations. On the other hand, ONOS and ODL, which are based on the OSGi framework, are not able to provide protection at the level of the control processes that support the application, since the respective attribute is not part of Apache Karaf. Of the five controllers analyzed in this work, only ONOS and SE-Floodlight implement policy conflict resolution. In ONOS, the application describes its network requirements in the form of "intents" and ONOS translates these intents with respect to the network configuration. With multiple controller instances, the provision of multiple application instances must also be considered. None of the studied controllers provides a mature solution for handling network state coordination across multiple app instances. Finally, as far as security storage is concerned, after reviewing the controllers, writers concluded that standard security practices are applied e.g. default permissions on log files to allow owner read/write privilege but read-only to others. Additional measures are applied to the individual controllers.

As far as controller interfaces are concerned and taking into consideration that the southbound interface (mentioned as D-CPI) is the only standardized interface, writers conclude that ODL, SE-Floodlight and Ryu are compatible with security requirements. The criterion that drove the above conclusion was SSL/TSL support.

In addition to protecting the control framework and interfaces, security services have been also evaluated. An Intrusion Detection System/Intrusion Prevention System (IDS/IPS) is provided in cases of ODL, Ryu and Floodlight. Relevant components, (Defense4All, Snort, BotHunter and Sec.Actuator) are used for this purpose. Defense4All is an application for detecting and mitigating DDoS attacks. The application communicates with the OpenDaylight Controller via the north-bound REST API to monitor the behavior of protected traffic and divert attack traffic to selected Attack Mitigation Systems (AMSs). Snort installs a flow to mirror incoming packets to the Snort network interface. A set of custom rules are generated and a packet matching a custom rule generates a Snort alert that generates an event alert in Ryu. The code can be extended for intrusion prevention. BotHunter monitors traffic to identify

communication patterns consistent with coordination-centric malware, BHResponder decides whether the identified asset should be quarantined from the network, and SDN Security Actuator links with SE-Floodlight to implement the quarantine i.e. generate OF rules to redirect suspicious traffic flows. Authentication, Authorization, and Accounting (AAA) are important aspects of the controller design, providing effective access control to users, applications, and resources. Although ONOS does not explicitly implement an AAA functions, the applications register with CoreService for a unique App ID to use ONOS services. The AAA project was launched in ODL mid-2014. The identity of users is authenticated, and user access to resources is authorized and recorded. The user is authenticated to the controller with a username/password combination and receives an access token so as to access protected resources on the controller. Access to specific resources is determined by the user role and permissions. Rosemary provides a similar AAA system in which applications are authorized to access specific controller resources. For privileged system calls, an application authorization module determines whether the application is authorized by investigating its signed key. In SE-Floodlight, authorization roles are assigned to applications during an application authentication procedure, which involves generation of a runtime credential to uniquely identify the application. The authorization role includes a set of associated permissions. The credential is added to each message produced by the application. Without the credential, the application will not run. All controllers have been identified as compliant with logging and auditing requirements.

In [48] the STRIDE method is used to analyze the security levels of four individual controllers: ODL, ONOS, Ryu and Rosemary. As expected, the authors concluded that none of the controllers is completely secure from threats and free from vulnerabilities. It is noted that the secure modes which are available in some of the controllers are only optional. As far as the controllers under evaluation are concerned, ONOS is vulnerable to known security threats. The multi-threaded architecture of Ryu may lead in DoS because of a lack of authorization for the addition of flow rules and event consumption. Rosemary handles resource access in an effective manner with the help of a finely-grained access control mechanism. Overall, ODL is recommended for the following reasons. The code is continuously checked for security vulnerabilities and improved by the security experts as it is an open-source project. Secondly, the modular structure of ODL has a performance benefit over other studied controllers. Finally, ODL has provided security mechanisms both for NBI and SBI including AAA and SNBI.

In [41], the behavior of POX and ODL controllers in case of Distributed Denial of Service (DDOS) attacks is analyzed. The Hping3 tool was used to simulate a DoS attack. Hping3 is a packet generator and analyzer for the TCP/IP protocol. The aim was to flood a controller with a bunch of packets using the Packet-In event. These packets were received in the switch with different IP addresses, which resulted in a missed flow that was automatically forwarded to a controller. In order to measure the effect of the attack, the bandwidth between the two hosts was measured. ODL proactively installs default flow tables in the data plane, but these flow tables cannot prevent flooded packets from reaching the controller. From our investigation, as the time for waiting for response from the server increases, it affects the bandwidth between the hosts. Additionally, th etime taken to receive a response was too large compared to the real time used in the iperf command. Using the l3_learning component in POX controller, no default flow table is installed by controller unless a controller learns about them when a Packet_In event is triggered. Results showed that as the number of flooding host attacks increases, a negative impact on the bandwidth between these hosts is seen. Low bandwidth was exhibited from the results on ODL and POX controller after launching a DoS attack. The authors suggest that these attacks may be prevented by implementing a packets rate limiter to prevent any traffic that isolates the SDN architecture. However, this should be implemented carefully, especially if a network consists of many hosts accessing the same server at a time. To avoid this, a flow aggregator may be used depending on the target host or application.

In [44] the STRIDE method and vulnerability testing tools were used to evaluate security features of ODL controller. Vulnerability tools (i.e. NMAP, Nexpose) revealed a number of vulnerabilities, such as missing flags in case of some cookies, unencyrpted credentials, open oports and “smuggling attacks”.

6. Conclusions

In this thesis, a number of SDN controllers and their capabilities have been studied. In the beginning of the thesis, a background study on SDN was made to show how this architecture can serve existing and future network requirements in terms of aspects such as programmability and flexibility. The rest of the thesis focused on SDN controllers. Firstly, the history, types, capabilities and main components of controllers have been described. Then a number of open source controllers were analyzed and compared in terms of their architecture components and basic features. Finally, emphasis was given to SDN controller efficiency its attributes (performance, scalability, security). Terms have been defined and relevant studies concerning open source controllers have been presented. Taking into consideration the above, the following conclusions can be drawn:

- SDN is a concept that exists for about 20 years. However, only the last few years it has become prevalent in the network community. This is due to the increasing needs in terms of network traffic and programmability that have been driven by the development of other areas, such as mobile devices, network virtualization and others.
- Due to that fact, that SDN concept is not considered mature enough and many areas have not been thoroughly researched. These refer to various areas of an SDN architecture such as the northbound interface, network security and others.
- The controller is the main building block of SDN. Its necessity and importance has led to the introduction and continuous development of several implementations. These belong to both the open-source community and to commercial vendors. In some cases, vendors have made some of their implementations available to open-source community for it to contribute.
- SDN controllers vary in many areas. On the one hand, this is considered positive since a lot of use cases and different requirements may be covered. On the other hand, proper controller selection is a multi-faceted problem that depends on the various factors.

- The analysis of some open source controllers has revealed that most of them are aligned as far as some basic features such as southbound support, multi-threading and modularity are concerned.
- Comparative analysis of northbound APIs is not comprehensive enough to provide significant insight.
- The aforementioned controllers are also aligned regarding some basic architecture components. What differentiates them is the way these are implemented and the extra features each of them has.
- Most of the research in controller capabilities is restricted to controller efficiency. This includes performance, security and scalability. Other parameters such as mobility and controller capacity have not been addressed to a large extent.
- Existing research regarding controller efficiency is considered immature. Analysis of open-source controllers have been performed using different controller versions and experiment set ups. Therefore, such a comparison cannot lead to valid conclusions.
- Most commercial controllers restrict their comparative analysis as far as their features are concerned and therefore, the way these are suitable to specific use cases cannot be analyzed within the scope of this thesis

Bibliography

[1] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, Ruslan Smeliansky, "Advanced study of SDN/Openflow controllers", Conference: Proceedings of the 9th Central & Eastern European Software Engineering, 2013

[2] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, Rob Sherwood "On Controller Performance in Software-Defined Networks", 2012

[3] Rahamatullah Khondoker, Adel Zaalouk, Ronald Marx, Kpatcha Bayarou, "Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers", Conference: ICCSA, 2014

[4] Ola Salman, Imad Elhajj, Ayman Kayssi, Ali Chehab, "SDN Controllers: A Comparative Study", 2016

[5] SDX Central, "The Future of Network Virtualization and SDN Controllers", https://www.sdxcentral.com, 2016

[6] Shie-Yuan Wang, Hung-Wei Chiu, Chih-Liang Chou, "Comparisons of SDN OpenFlow Controllers over EstiNet: Ryu vs. NOX", ICN 2015: The Fourteenth International Conference on Networks, 2015

[7] Ian F. Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, Wu Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks", Elsevier Computer Networks 71 (2014) 1–30

[8] Brandon Heller, Rob Sherwood, Nick McKeown, "The Controller Placement Problem", HotSDN'12, 2012

[9] Hyojoon Kim, Nick Feamster, "Improving Network Management with Software Defined Networking", IEEE Communications Magazine, 2013

[10] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks", Communications Surveys and Tutorials, IEEE Communications Society, Institute of Electrical and Electronics Engineers, 2014, 16 (3), pp.1617 - 1634

[11] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky and Steve Uhlig, Member, "Software-Defined Networking: A Comprehensive Survey", https://arxiv.org/abs/1406.0440, 2014

[12] SDX, "SDN Controllers Report", https://www.sdxcentral.com, 2015

[13] Siamak Azodolmolky, "Software Defined Networking with Open Flow", 2013

[14] Verizon, "SDN-NFV Reference Architecture", http://innovation.verizon.com, 2016

[15] Pritesh Ranjan, Pankaj Pande, Ramesh Oswal, Zainab Qurani, Rajneeshkaur Bedi, "A Survey of Past, Present and Future of Software Defined Networking", Volume 2, Issue 4, International Journal of Advance Research in Computer Science and Management Studies, 2014

[16] Andrei Bondkovskii, John Keeney, Sven van der Meer, Stefan Weber, "Qualitative Comparison of Open Source SDN Controllers", 2016

[17] Zuhran Khan Khattak, Muhammad Awais and Adnan Iqbal, "Performance Evaluation of OpenDaylight SDN Controller", 2014

[18] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart and Amin Vahdat, "B4-Google", SIGCOMM'13, August 12–16, 2013, Hong Kong, China

[19] Bing Xiong, Kun Yang, Jinyuan Zhao, Wei Li, Keqin Li, "Performance evaluation of OpenFlow-based software-defined networks based on queueing model", Elsevier Computer Networks 102 (2016) 172–185

[20] Michael Jarschel, Frank Lehrieder, Zsolt Magyari, Rastin Pries, "A Flexible OpenFlow-Controller Benchmark", 1st European Workshop on Software Defined Networks, 2012

[21] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, Shriram Krishnamurthi, "An API for Application Control of SDNs", SIGCOMM'13, August 12–16, 2013, Hong Kong, China

[22] Stephen Mallon, Vincent Gramoli, and Guillaume Jourjon, "Are Today's SDN Controllers Ready for Primetime?", https://arxiv.org/abs/1608.05140, 2016

[23] Marcial P. Fernandez, "Evaluating OpenFlow Controller Paradigms", ICN 2013 : The Twelfth International Conference on Networks, 2013

[24] Guillermo Romero de Tejada Muntaner, "Evaluation of OpenFlow Controllers", 2012

[25] Pankaj Berdey, Matteo Gerolaz, Jonathan Harty, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantzy, Brian O'Connory, Pavlin Radoslavovy, William Snowy, Guru Parulkary, "ONOS: Towards an Open, Distributed SDN OS", HotSDN'14, August 22, 2014, Chicago, IL, USA

[26] Shiva Rowshanrad, Vajihe Abdi and Mamijeh Keshtgari, "Performance Evaluation of SDN Controllers: FloodLight and OpenDaylight", IIUM Engineering Journal, Vol. 17, No. 2, 2016

[27] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, Ramana Kompella, "Towards an Elastic Distributed SDN Controller", HotSDN'13, August 16, 2013, Hong Kong, China

[28] https://www.opendaylight.org

[29] http://zhengcai.github.io/maestro-platform

[30] Zheng Cai, “Using and Programming in Maestro”

[31] Eugen Borcoci, Radu Badea, Serban Georgica Obreja, Marius Vochin “On Multi-Controller Placement Optimization”, ICN 2015: The Fourteenth International Conference on Networks, 2015

[32] https://openflow.stanford.edu/display/Beacon/Home

[33] David Erickson, “How Beacon works”, 2011

[34] David Erickson, “The Beacon OpenFlow controller”, HotSDN’13, August 16, 2013, Hong Kong, China

[35] Zheng Cai Alan L. Cox T. S. Eugene Ng, “Maestro A System for Scalable OpenFlow Control”, https://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf

[36] https://en.wikipedia.org

[38] https://thenewstack.io

[39] Abhishek Rastogi, Abdul Bais, “Comparative Analysis of Software Defined Networking (SDN) Controllers – In Terms of Traffic Handling Capabilities”, Multi-Topic Conference (INMIC), 2016 19th International

[40] Abhinandan S Prasad, David Koll, Xiaoming Fu, “On the Security of Software-Defined Networks”, Conference: EWSDN, At Bilbao Spain, 2015

[41] Huseyin Polat, Onur Polat, “The Effects of DoS Attacks on ODL and POX SDN Controllers”, ICIT 2017 The 8th International Conference on Information Technology

[42] Nguyen Tri-Hai, “Vulnerability Analysis of Controllers in SDN”, https://www.researchgate.net, 2017

[43] Ruslan. L. Smeliansky “SDN is it a solution for network security?”, https://www.researchgate.net, 2013

[44] Izzat Alsmadi, William Unger “Evaluating security of SDN controllers”, Conference: Software Defined Networking (SDN) for Scientific Networking Workshop, At Austin, 2015

[45] Pedro Bispo, Daniel Corujo, Rui L. Aguiar “A Qualitative and Quantitative assessment of SDN controllers”, 20171nlemalional Young Engineers Forum (YEF~ECE), 2017

[46] Alexandru L. Stancu1, Simona Halunga1, Alexandru Vulpe1, George Suciu1, Octavian Fratu1, Eduard C. Popovici1 "A comparison between several SDN Controllers", TELSIKS 2015 conference

[47] Sandra Scott-Hayward, "Design and deployment of secure, robust and resilient SDN controllers", Network Softwarization, 2015 1st IEEE Conference

[48] Ramachandra Kamath Arbettu, Rahamatullah Khondoker, Kpatcha Bayarou, Frank Weber, "Security analysis of ODL, ONOS, Rosemary and Ryu SDN Controllers"

[49] "SDxCentral: Future of Network Virtualization and SDN Controllers Report", https://www.sdxcentral.com, 2016

[50] Thomas Nadeau D., Ken Gray, "SDN Software Defined Networks", 2013

[51] Siddharth Valluvan, T. Manoranjitham, V.Nagarajan "A study of SDN Controllers", International Journal of Pharmacy & Technology, 2016

[52] Taimur Bakhshi, "State of the art and recent research in SDN", Wireless Communications and Mobile Computing Volume 2017 Article ID 7191647

[53] E. Haleplidis, K. Pentikousis, S. Denazis, J. Hadi Salim, D. Meyer, O. Koufopavlou "SDN Layers and Architecture Terminology", Internet Research Task Force (IRTF), 2015

[54] Paul Goransson, Chuck Black "Software Defined Networks, A Comprehensive Approach", 2014

[55] "OpenFlow Version Roadmap", Ching-Hao, Chang and Dr. Ying-Dar Lin,2015

[56] R. Sherwood, and K. Yap, "Cbench Controller Benchmarker", 2011

[57] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, Nick McKeown, "ElasticTree: Saving Energy in Data Center Networks", 7th USENIX conference on Networked systems design and implementation Pages 17-17

[58] Kok-Kiong Yap, Masayoshi Kobayashi, David Underhill, Srinivasan Seetharaman†, Peyman Kazemian and Nick McKeown, "The Stanford OpenRoads Deployment", WiNTECH'09, 2009

[59] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker "Modular SDN Programming with Pyretic", https://www.cs.princeton.edu, 2013

[60] https://wiki.onosproject.org

[61] http://www.projectfloodlight.org

[62] J.E. van der Merwe, S. Rooney, L. Leslie, "The Tempest-a practical framework for network programmability", IEEE Network, Volume: 12, Issue: 3, May/Jun 1998