South East European University – Tetovo

Contemporary Sciences and Technologies Faculty

Shpëtim S. Latifi

SOFTWARE DEFINED NETWORKING MODELS IMPLEMENTATION FOR TRAFFIC MANAGEMENT FOR ENTERPRISE NETWORKS AND INTER-DOMAIN ROUTING

Doctoral Thesis

Tetovo, 2017

COMMISSION FOR DEFENSE OF DOCTORAL THESIS

1. Prof. Aksenti Grnarov, President

2. Prof. Arjan Durresi, Supervisor

3. Prof. Betim Cico, member

4. Assoc. Prof. Lejla Abazi-Bexheti, member

5. Assoc. Prof. Halil Snopce, member

Contemporary Sciences and Technologies Faculty,
South East European University
Tetovo, 2017

To my wonderful parents.

ACKNOWLEDGEMENT

I would like to express my unique gratitude to my supervisor, prof. Arjan Durresi for his great support during the course of the research. His up to date expertise and comments have been crucial to me. I had been my privilege to work under his mentorship.

I would like to give a special thanks to prof. Betim Cico for the countless hours of work review, comments and advice; also for the motivation that he gave me during the course of the research. His effort and expertise have been invaluable to the completion of the thesis.Words cannot describe my gratitute.

I would like to express my gratitude to prof. Aksenti Grnarov for his mentorship and supervision during the early stage of my research. His knowledge of the matter helped clearly shape the focus of the research, and served as a great motivator.

Shpëtim S. Latifi

SOFTWARE DEFINED NETWORKING MODELS IMPLEMENTATION FOR TRAFFIC MANAGEMENT FOR ENTERPRISE NETWORKS AND INTER-DOMAIN ROUTING

Doctoral Thesis

ABSTRACT

Software Defined Networking (SDN) is meant to be a de facto paradigm in data networks. However for the time being, only larger campuses and data centers are utilizing the advantages of SDN, primarily taking advantage of easier network management, better resource utilization, better load and traffic balance on network devices and routes. But majority of networks, mainly small and campus networks, those that constitute the largest number of corporate networks while smaller campuses and enterprises are not yet widely deploying SDN. The future of SDN is therefore uncertain and it tightly depends on apps that are to be developed for various types of networks.

SDN allows for implementation of control function in the edge of the network, instead of in the routers, where it actually is, and the core only deals with delivering packets end-to-end. The core may easily remain legacy hardware and the network operators need not to know at all the one is implementing SDN from the own edge of the network. So it simplifies network troubleshooting and also there are no disruption periods during network convergence, due to the fact the one policy is implemented per specific flow, and each packet will be carried either by an old state, or by a new state. In traditional routing packets may well be lost during network convergence, or loops may appear.

The SDN gives an extra benefit to the network operators to more easily customize their networks based on their needs. The policies, traffic engineering, monitoring and security is easier to implement after getting a network state quickly from the Network Operating Systems, and moving the network to a virtual environment without much effort is perhaps the greatest benefit in this specific scenario.

In this thesis we design, implement and analyze models and applications of SDN for enterprise networks, as well as inter-domain routing. The results from emulated and simulated implementation show that SDN offer greater flexibility and allows for better traffic engineering than conventional techniques; it also enables quick transitioning to SDN enterprise networks from traditional ones and better overall networks management. In inter-domain routing, SDN imlpementatioons also give a

possibility to fight Denial of Service attacks, as well as better traffic engineering.

We also emulate an enterprise/university Software-Defined network using Python and Mininet, as well as a conventional network using ns-3 with the same number of nodes, topology, etc. Our tests show that Software-Defined Networks outperform conventional networks in parameters like minimum and mean delay for ICMP and HTTP packets, as well as minimum and maximum convergence time, and therefore are a best solution to transition corporate networks with hundreds or more nodes, mainly due to the fast propagation of routing policies from the central controller.

LIST OF ACRONYMS

RIP - Routing Information Protocol

OSPF - Open Shortest Path First

IGRP - Interior Gateway Routing Protocol

BGP - Border Gateway Protocol

SDN - Software Defined Networking

IXP - Internet Exchange Point

SDX - Software Defined Internet Exchange

OVS - Open vSwitch

POX - Open source development platform for Python-based software-defined networking control applications

NOX - Open source development platform for C++ based software-defined networking control applications

NS-3 - Network Simulator 3

GNS3 - Graphical Network Simulator 3

ARPANET – Advanced Research Project Agency Network

TABLE OF CONTENT

LIST OF FIGURES

I. CHAPTER - INTRODUCTION

1.1. Background

A computer network is a set of two or more computers, connected via one specific technology in order to share data and/or resources. A connection intuitively can be set up using cable media or no cable (wireless). In computer networks, a connected computer machine is called a network host (also network node). Similarly, in wireless networks, machines are called stations (STAs).

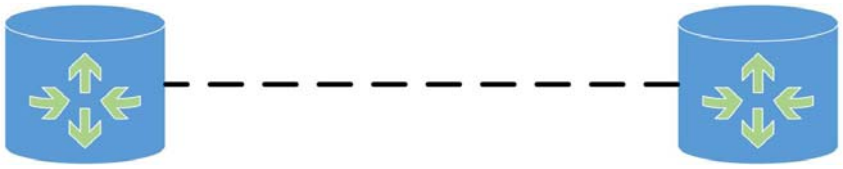


Figure 1. A basic network consisting of two routers

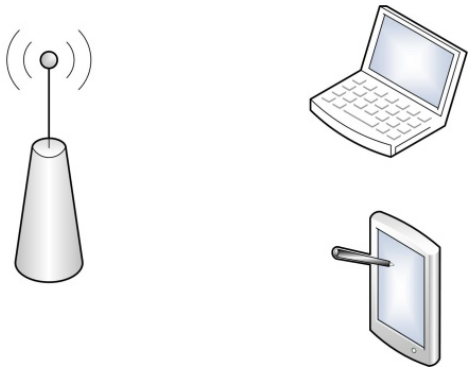


Figure 2. A basic wireless infrastructure network with two stations (STAs) and a wireless access point

The beginning of network connection dates back to late fifties and sixties of the previous century, with the military radar system of the U.S., as well as the semi-automatic business research environment (SABRE), when two so-called mainframe computers were connected. However, real computer networking started with ARPANet, the predecessor of today's Internet, in 1969. Four centers were connected as a beginning, namely University of California at Los Angeles, Stanford Research Institute, University of California at Santa Barbara at data rates of 50 kbps. This

network grew rapidly in the following period, and substantial advances and changes were made to the original network, especially with the underlying infrastructure of network models, such as OSI and TCP/IP.

If we are to make a general network classification and taxonomy, networks can be classified into circuit switched networks and packet switched networks. This categorization has to do mainly with the way how messages are passed onto the network and towards the destination, usage of network resources, switching and forwarding in the network, addressing, quality of service, etc. Data networks are generally packet switched networks, where the Internet Protocol (IP) serves as the main network layer routed protocol. Through IP, various technologies, media (physical layer) are used to connect devices, applications, services (application layer) into the global network called the Internet. At different ends of the Internet however, also routing is done differently, depending on whether we utilize datagrams or virtual circuits. As the networking technologies are growing, the number of hosts is also growing. With this the complexity of the networks is also growing. This in turn, makes the network management harder. In order to increase availability, security and reliability, network engineers employ numerous network engineering techniques. When doing so, they do not have full and instantaneous access and view of the network topology and current policies (on all routers), and therefore no way of direct and full management of the network itself, but rather they usually make configuration changes in each physical router. This costs time, effort and is impractical. Certainly, routers are managed remotely, but this does not decrease the level of work complexity a network administrator needs to undertake to properly configure her network.

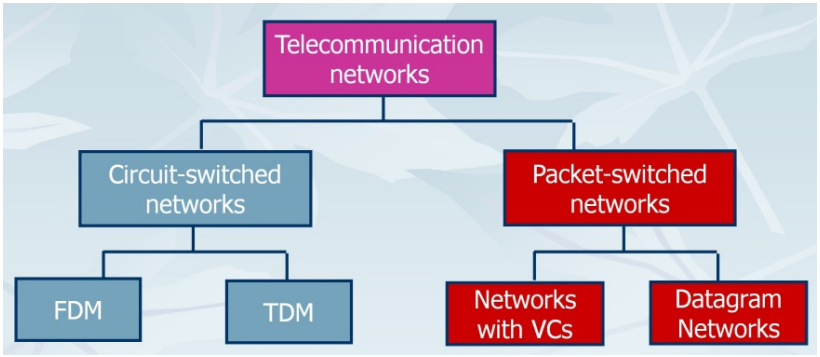


Figure 3. General taxonomy of telecommunication networks: circuit and packet switching

Software Defined Networking is the answer to the decades-long problem of

continuously increasing network complexity, and therefore increasing headaches for network administrators. The origins of software-defined networking began shortly after Sun Microsystems released Java in 1995. [1][2][3]
In April and May 2001, Ohio State University and OARnet, collaboratively ran the first SDN test and developed the first practical SDN use case for Internet2. After successful completion of tests, OARnet issued the following statement on May 8, 2001:

"We have witnessed the successful first step to the fulfillment of smart, interoperable networks through the deployment of Supranet Transaction Server. A technology first was accomplished as a new set of instructions was dynamically transmitted across the network, changing the behavior of the requesting computer. There was no need to take down any part of the system and there was no interruption of service. Our testing will continue and we anticipate further advancement of the next generation Internet through our partnership with Websprocket" – Pankaj Shah (Managing Director, OARnet). [4]

Hence, Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the control plane of the network and forwarding plane and functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. The OpenFlow protocol is a foundational element for building SDN solutions. [5]

1.2. Literature review and state of the art

During the first decade of this century, a numerous research articles and projects were carried out in order to propose ways to mitigate and reduce network complexity in a systematic manner, but separating data plane from control plane in the network. Furthermore, this decoupling would also allow for remote administrator and writing of easier third party applications for network and traffic management for network administrators. The real problem was how to access the router in order to install these policies in it, knowing that there was no such interface to interact with the router.

Hence, the most imporant step in addressing this question is the design and specification of the OpenFlow, which is explained above. In a paper of 2008 the idea for OpenFlow was laid out. The paper states that 'OpenFlow allows for a way for researchers to run experimental protocols in the networks they use every day. OpenFlow is based on an Ethernet switch, with an internal flow-table, and a standardized interface to add and remove flow entries. Our goal is to encourage networking vendors to add OpenFlow to their switch products for deployment in college campus backbones and wiring closets. We believe that OpenFlow is a pragmatic compromise: on one hand, it allows researchers to run experiments on heterogeneous switches in a uniform way at line-rate and with high port-density; while on the other hand, vendors do not need to expose the internal workings of their switches. In addition to allowing researchers to evaluate their ideas in real-world traffic settings, OpenFlow could serve as a useful campus component in proposed large-scale testbeds like GENI.

'Two buildings at Stanford University will soon run OpenFlow networks, using commercial Ethernet switches and routers. We will work to encourage deployment at other schools; and we encourage you to consider deploying OpenFlow in your university network too'. [6]

A paper which stresses the need and gives a specific idea for a network operating system is NOX. [7] This “operating system” provides a uniform and centralized programmatic interface to the entire network. Analogous to the read and write access to various resources provided by computer operating systems, a network operating system provides the ability to observe and control a network. A network operating system does not manage the network itself; it merely provides a programmatic interface. Applications implemented on top of the network operating system perform the actual management tasks. The programmatic interface should be general enough to support a broad spectrum of network management applications.

A very nice OpenFlow controller is proposed with Beacon. [8] Beacon is a Java-based open source OpenFlow controller created in 2010. It has been widely used for teaching, research, and as the basis of Floodlight. This paper describes the architectural decisions and implementation that achieves three of Beacon’s goals: to

improve developer productivity, to provide the runtime ability to start and stop existing and new applications, and to be high performance.

Another paper that focuses on tracerouting through SDN is SDN Traceroute. SDN traceroute can query the current path taken by any packet through an SDN-enabled network. The path is traced by using the actual forwarding mechanisms at each SDN-enabled device without changing the forwarding rules being measured. This enables administrators to discover the forwarding behavior for arbitrary Ethernet packets, as well as debug problems in both switch and controller logic. This prototype implementation requires only a few high-priority rules per device, runs on commodity hardware using only the required features of the OpenFlow 1.0 specification, and can generate traces in about one millisecond per hop [9].

Frenetic is a language for Software-Defined Networks proposed in 2013. [10] In the Frenetic project, authors are designing simple and intuitive abstractions for programming the three main stages of network management: (i) monitoring network traffic, (ii) specifying and composing packet-forwarding policies, and (iii) updating policies in a consistent way. Overall, these abstractions make it dramatically easier for programmers to write and reason about SDN applications. Frenetic provides a declarative query language for classifying and aggregating network traffic as well as a functional reactive combinator library for describing high-level packet-forwarding policies. Unlike prior work in this domain, these constructs are—by design—fully compositional, which facilitates modular reasoning and enables code reuse. This important property is enabled by Frenetic's novel runtime system which manages all of the details related to installing, uninstalling, and querying low-level packet-processing rules on physical switches. [11]

NetCore is a compiler and run-time system for SDN, where authors define a high-level, declarative language, called NetCore, for expressing packet-forwarding policies on SDNs. NetCore is expressive, compositional, and has a formal semantics. To ensure that a majority of packets are processed efficiently on switches—instead of on the controller, a new compilation algorithm for NetCore is presented and couples them with a new run-time system that issues rule installation commands and traffic-statistics queries to switches. Together, the compiler and run-time system generate

efficient rules whenever possible and outperform the simple, manual techniques commonly used to program SDN today. [12]

Procera ia a language for High-Level Reactive Network Control, where a control architecture is proposed for software-defined networking (SDN) that includes a declarative policy language based on the notion of functional reactive programming; the formalism is extended with both signals relevant for expressing high-level network policies in a variety of network settings, including home and enterprise networks, and a collection of constructs expressing temporal queries over event streams that occur frequently in network policies. [13]

Certainly, one of the tools used to emulate the SDN modeling of networks in Mininet, which is defined and proposed in Mininet. [14] Mininet is a system for prototyping large networks on the constrained resources of a single laptop. The lightweight approach of using OS-level virtualization features, including processes and network namespaces, allows it to scale to hundreds of nodes. The paper itself (or the system developed therein) does not offer specific solutions to the posed questions in the thesis, since it merely is a tool and an environment for each network administrator/programmer to create their networks, based on their own needs.

Based on the literature review, one can conclude that the research questions posed in this thesis are not fully answered, and the proposed models are not comprehensive in terms of both intra-domain and inter domain routing. SDN itself is a new phenomem in data networking, and therefore it is not widely deployed in networks, although almost all latest routers do implement OpenFlow as an interface that allows for a new way of interaction not only with a specific router, but rather with the whole network (the logical topology of the network). Most issues in traditional networks continue to persist, whereas the future of SDN directly depends on introduced models and applications.
The full list of literature and research papers that has been reviewed for this thesis can be found in the References section, following the last chapter.

From the literature review, we have raised a number of research questions which for the time being, in the field of SDN are not answered or at least are not completely

answered (please see the section below for the research questions). Hence, the basis and motivation for this research thesis. Based on the literature review, and the posed research questions, we have identified the problems that need to be addressed, as well as the research hypothesis are developed, as explained in the following section.

1.3. Problem statement and research hypothesis

Data networks have become increasingly complex nowadays. Even though technologies like Ethernet, IP protocol and packet forwarding are rather simple, control mechanisms like middleboxes, Access Control Lists (ACLs), firewalls, traffic engineering, VLANs, etc. have largely contributed to increasing their complexity. Primarily this is due to the lack of basic principles in networking. Networking still remains vertically integrated, where hardware comes with its proprietary software and is not open to innovation. Denial of Service has been another concern for the last decades, and in the current Internet it has not been fully addressed.

In the area of inter-domain routing, BGP (Border Gateway Protocol) lacks sufficient flexibility for performing various traffic engineering and traffic management operations. It's generally possibly to route only on destination IP address blocks, and it is only possible to exert indirect control over how switches and routers forward traffic, through local preference, etc. But generally it is difficult to introduce new network services, such as ability to arbitrarily route traffic flows through middleboxes.

Software-Defined Networks (SDN) instead decouples the data plane (which is and should remain the job of the physical routers) and control plane. The control plane in SDN is removed from the routers and switches, and instead is done in the edge of the network, thus allowing for third party software, open interface to devices regardless of hardware type and vendor, and easier management of networks. SDN is a new design model in networks rather than a new technology. It is a set of abstractions for the control plane rather than implementation mechanisms.

However for the time being, only larger campuses and data centers are utilizing the advantages of SDN, primarily taking advantage of easier network management, better resource utilization, better load and traffic balance on network devices and routes. But majority of networks, mainly small and campus networks, those that constitute the

largest number of corporate networks while smaller campuses and enterprises are not yet widely deploying SDN. The future of SDN is therefore uncertain and it tightly depends on apps that are to be developed for various types of networks.

The research questions that are discussed in the thesis, based on the abovementioned issues on one hand, and the literature review on the other, are:

a) Is SDN a feasible alternative to traditional campus and data-center networks?
b) Can network performance be improved with SDN in data centers with unpredictable traffic patterns?
c) Can SDN be implemented for traffic management between various Autonomous Systems (AS) successfully?
d) Does OpenFlow allow for better network security and traffic engineering for inter-domain routing?

This thesis analyses and tests possible models and design issues in SDN for intra-domain as well as inter-domain routing. The results from simulated implementations will be analyzed and compared to similar scenarios in traditional non-SDN topologies to see how problems of flexibility, traffic engineering, network management, network security, such as Denial of service attacks; ways to improve various network services in SDN are investigated, as opposed to traditional non-SDN techniques.

The following hypothesis can be raised in order to define the problem more clearly:

H1 – Large datacenters and campus networks are more feasible and can be managed more easily using Software Defined Networking models by creating logical overlays of network topologies;

H1a – Third party models and applications enable translation of existing network topologies to SDN deployments;

H2 – SDN models allow for an improved traffic management in enterprise networks over traditional (non-SDN) networks.

H3 - Software Defined Networking allows for improved security and easier traffic engineering for inter-domain routing between Autonomous Systems.

1.4. Structure

This doctoral thesis is organized in 7 chapters.
Chapter I introduces basic background information of current data networks and potential issues related to such organizing principle. The lireturare review is presented, as well as the main research problem is stressed and the research hypotheses are laid out. With that, the scientific importance of the thesis is explained and other organizational and structural components, such as research methodology and data gathering methods.

Chapter II gives a more detailed explanation of network devices - layers at which each of them operates. Of particular interest in this regard is the operation of network routers, routing protocols within the routers. The basic functions of routers are explained such switching and forwarding in order to underline the problem of traditional model in hardware and software coupling, as well as problems associated with it.

Chapter III gives another insight in the trends, current Internet architectures and a wider overview of trends in networking in the past decade. Over 30 select research papers and other research work is analyzed, and various aspects are compared and contrasted. Also SDN development languages and environments are discussed in this chapter.

Chapter IV is composed of practical implementations of SDN scenarios and models. These implementations pose a comprehensive testing with many iterations and small modifications among each, of practical simulations and emulations of SDN scenarios, mainly for enterprise networks. The results are compared to traditional implementations. Some of most used network topologies and environments are used for the tests.

Chapter IV is composed of practical implementations of SDN scenarios and models, and more detailed testing of practical emulations of SDN scenarios for inter-domain routing, and the results are compared to traditional implementations. Some of most used network topologies and environments are used for the tests.

Chapter VI is a summary of the problem treated in this thesis, as well as it gives a summary of findings from literature review. It also reviews the proposed models to solve existing problems in networks, and discusses future research in the field.

Chapter VII lists all the literature and bibliography used as a reference for this thesis, as well as gives the codes used to build the network topologies and run the tests in specific environments in the appendix.

1.5. Importance of the study

One key characteristic of the Internet architecture is that it allows the system to evolve. In other words, the architecture is minimalistic and general, broad to support applications that are not developed yet; support technologies that are different from one another in many ways. This specific of the Internet has made it so robust that it has prevailed drastic technology improvements in all the years.
In today's networks we can easily says that topology is policy, meaning the actual physical location of routers, firewalls, etc. dictates how effective the network is, how well the ACLs work, what our broadcast domains are, etc. *When networks are moved to the cloud, we usually want to keep same policies*, but very few networks operators have an abstract expression of network policy, rather they have a network topology. Therefore SDN allows specifying a logical topology to the cloud. The cloud then ignores the physical topology and follows the logical topology based on the policy read based on the topology initially. Topologies differ, depending on real needs of network administrators, but this is not the scope of analysis of this thesis. It rather deals with model analysis and comparison of practical SDN and non-SDN implementations (simulation and emulation tests) and provides a quantitative

approach in testing various network parameters, protocols, applications and mechanisms.

This study gives a general contribution to the adoption of SDN principles in data networking, and most importantly, through specific and practical simulations and emulations it provides an original model towards which typical data center and campus networks can migrate, and also can be used for better traffic management.

1.6. Research methodology and data gathering

This study applies a quantitative research approach in order to support the assumptions and hypothesis of the study in regard to Software-Defined Networking. Specifically, network models of large datacenters are implemented in network simulators such as NS-3 [15] and GNS3 to implement given specific network topologies, as well as mininet environement is used to develop the corresponding implementations of SDN network topologies and compare network parameters on both implementations. This approach also tries to test and support implicitly the sub-hypothesis that translating existing network topologies into SDN implementations quickly in campus networks. Finally, traffic control for inter-domain routing, as well as in enterprise networks is implemented to support the second and third hypotheses.

This thesis is exploratory as well in the sense that it uses existing research papers to collect additional data relevant to the topic of Software-Defined Networking for campus networks and large datacenters.

1.7. Summary

This is the intruductory chapter to the doctoral thesis titled "Software defined networking models implementation for traffic management for enterprise networks and inter-domain routing".

It gives a general overview and definitions about data networks, a historical perspective of the Internet and its predecessors (such as ARPANet), as well as lays out the ground for the thesis and research content by giving brief introduction to Software-Defined Networking.

A significant importance is given to the literature review, where over 50 various

research papers in the field od SDN have been reviewed, and based on which the research is done. Furthermore, this chapter presents the thesis problem statement, as well as the three main research hypotheses of this research work. The research methodology is given and tools and techniques for data gathering.

CHAPTER II. ROUTING IN DATA NETWORKS

2.1. Introduction

This chapter describes the functioning of data networks in detail. It starts with a detailed description of network components, and how they operate, and then explain the functioning of network routers. In order to understand the rationale behind Software Defined Networking, one needs to unequivocally understand the problem with existing networking model, or more specifically the problem traditional networks. Hence, this chapter gives a detailed overview of the development of network hardware and software, and the issues related to the vertical coupling of both. As a comparison, it also explains the historic development of computer systems, as opposed to computer networks, to underline and emphasize the problem. More specifically, the chapter also analyses and describes specific problems in networking in context of the contribution of this thesis, to make clear the importance and relevance of this research. Specific traffic engineering problems in actual data networks are discussed, issues of middleboxes, and other mechanisms which violate basic networking principles, such as the end-to-end principle, but on the other hand mechanisms which are used on daily basis by network administrators to leverage existing problems.

2.2. Network devices

Just as we need to connect two or more computers into one network, we also connect multiple networks among themselves. The latter is called an internetwork, or internet. To do the former or the latter, we need special equipment (devices), which also run specific software to perform their tasks. These devices are designed to overcome obstacles to interconnection without disrupting the independent functioning of the networks.

'An internet is different from the Internet'. [16]

Network devices are categorized into four groups: repeaters, switches (and bridges), routers, and gateways.

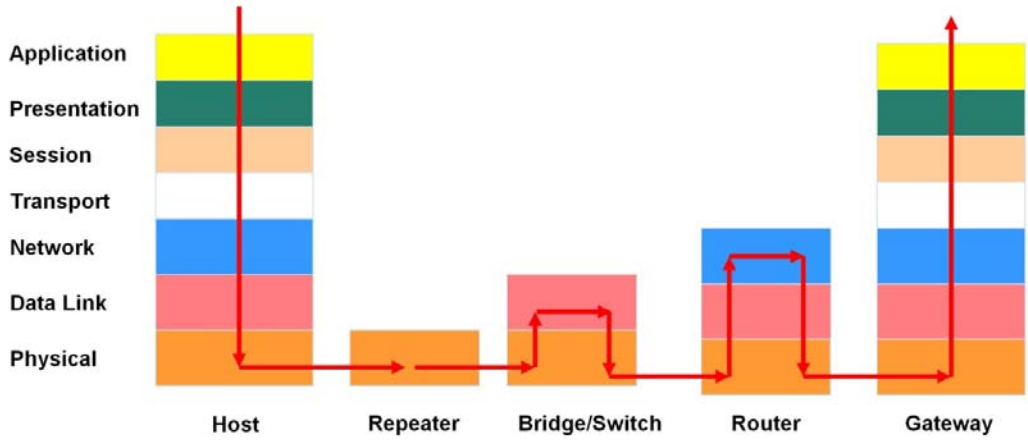


Figure 4. Connecting devices in the OSI model

The first two types are used to connect computers (hosts) in a network, whereas the last two types are used to connect networks in an internetwork.
Each type of network device sends and receives signals from network hosts through interacting with protocols at different layers of TCP/IP.

Repeaters are the 'dumbest' network devices operating at the physical layer of the OSI model. Figure 4 depicts the operation of a repeater. Each signal, when propagating through whatever medium (wired or wireless) and carrying data can only reach a specific and fixed distance before it attenuates so badly that the signal becomes unreadable. In order to prevent from endangering the integrity of the data, repeaters are installed on a network link just before the signal becomes too weak and then it regenerates it. This allows for the signal to traverse the medium as if it was a fresh signal (and message), thus reaching distant hosts, otherwise unreachable. It is worth noting that the repeater, besides extending the physical length of a network (not infinitely) does not change the nature or functionality of the network; it does not have any software that would allow for any intervention in the network, such as filtering, switching, error and integrity checking, selecting right path, etc. It merely strengthens incoming signal before putting it on outgoing link. The outgoing signal strength tends to be identical to the original strength, since each bit undergoes the process or reproduction in the repeater, as opposed to amplifying the signal.

In contrast to this, *bridges and switches* are network devices that operate at the data link layer of the OSI Reference Model, along with physical layer. This makes the

bridge (and the switch) a more intelligent device than the repeater. The bridge, not only does regenerate the signal (just like the repeater), but also employs added functionality by making forward/flood/discard decisions based on physical (Medium Access Control or MAC) address. They can relay frames between different LAN segments or LANs, and is able to control local network congestion, and therefore avoid other problems, such as faulty links.

Gateways are more sophisticated networking devices, which in contrast to all previous can potentially run on all layers of the network model (all seven layers of the OSI Reference Model, or all four layers of the TCP/IP model. Practically, gateways are a type of routers which in addition to the hardware capability similar to routers', they have specific software. This software enables gateways to forward packets from and to networks using different protocols. Hence, gateways are a type of translator, in often cases translating packet of one type (usually changing packet headers) to another. Also gateways are able to change frame formats completely, making it possible for example to send an http message to a cell phone recipient in the form of a text message (SMS). Another function of gateways is encryption: the trend in the past decade is to slowly but surely switch all Internet traffic from plain to encrypted traffic. Therefore, encryption gateways are used in those instances to enable this functionality of the network.
One concept that was mentioned here and will be discussed in the following paragraphs is the *router*.

2.2.1. Routers

Routers are another type of network devices, which are essential to this thesis, and the domain of Software Defined Networking. Routers are more sophisticated devices which, unlike repeaters and bridges, have the software capabilities of selecting a best path for incoming packets, given that multiple routes to a destination exist. Routers operate at the network layer of OSI Reference Model, and therefore not only do they understand physical addresses, but also IP addresses. This allows them to be able to route packets among different interconnected networks. Routing packets among different networks does not necessarily mean that the router is responsible for packet delivery to the destination. For example, it often happens that between the source and

the destination there are multiple routers en route. When the first router receives a packet, it only chooses a path to the next router (hop) in an attempt to reach its destination. When the next router receives the packet, the first one has already finished its job. In rare cases it happens that a router receives packets from one network and forwards them to another one which is directly connected and is the packet's destination – usually packets travel through multiple hops (routers). By default routers are single protocol devices. This means that if two networks are connected with each other (thus forming an internet) they will use same network layer protocol. In most cases this protocol is IP (Internet Protocol), but other possibilities also exist, such as Novell's IPX network protocol. In exceptional cases, routers can talk multiple different protocols, and such routers are called multiprotocol routers. In these cases, routers have more than one routing table, one for each of the network protocols, in order to make sense of the packets it receives and to be able to convert them into the respective protocol of the intended recipient network.

From the above mentioned one can understand that routers perform two basic functions: 1. To select a path towards destination for packets it receives (the process of receiving the packet on one ingress port and sending it to another egress port in networking is called *forwarding*), and 2. To make this decision based on some rule or policy (compare the destination address of each packet with a routing table, which has to be maintained and updated from time to time. This process is called *switching*). This is where things get complicated and these software operations typically are pre-installed by hardware vendors and little flexibility is allowed. In other words, hardware and software is vertically coupled (in a router) and code changes are not possible for third party software developers. More importantly, many issues that have arisen in the last decades have been only fixed using mechanisms, in the absence of logical partitioning of tasks in order to decrease the complexity of the problem (for instance to allow for code reuse). This is the essence of the problem in networks traditionally and this is what Software Defined Networking tries to solve.

2.3. Routing Protocols

There is a general confusion even among the IT community about routing and routed protocols. These two concepts although similar in writing, are different in function

and tasks. Routed protocols enable routers to forward packets and address intended recipients on the network correctly. In order to do this, networks and hosts are assigned addresses (numbers) by the routed protocol. Hence, a most well-known routed protocol is the IP protocol. It is a connectionless and unreliable protocol. It assigns an address to each network (and each host within each network) in order each one of them to be addressable. This address therefore has a network portion and a host portion within itself. This feature makes it even more practical (and possible) the routing process – routers do not need to know (and save in their routing tables) each host's address. Instead, they only save network addresses, and based on that information they perform routing. IP protocol does not perform error checking, and therefore it is also called best-effort delivery protocol.

On the other hand, routing protocols allow routers to communicate among themselves and enable them to construct paths to them. In order to perform routing, routers need to know how to reach certain remote network. Generally, they can learn this either from other routers (dynamically), or from network administrators telling the router which path to follow (statically). Examples of routing protocols are Routing Information Protocol (RIP), Open Shortest Path First (OSPF), Cisco Interior Gateway Routing Protocol (IGRP), Border Gateway Protocol (BGP), etc. Each routing protocol maintains a routing table, in which all learnt networks are kept. When routers exchange information among themselves to learn and construct paths, they possibly learn multiple paths to a certain destination. In that case, routers maintain the best path to the destination in the routing table, while removing from it invalid paths. Routing protocols are essential to dynamic routing. As networks change in terms of size, topology, performance, reachability, etc. routing protocols actively maintain routing tables to have an updated view of the current topology. When all routers have updated information about the topology, it is said that the network has converged.

To further understand differences among routing protocols, it is important to understand that the Internet can be seen as a set of autonomous systems (AS). Autonomous systems are in turn a set of routers which are administered by an organization, such as an Internet Service Provider (ISP). Autonomous systems may choose to use multiple routing protocols. There are over 3,500 registered autonomous systems in the Internet, and most of them have only one connection to the outside world (stub AS). In contrast there are two other types of AS: multihomed AS, which

have multiple connections to the outside world, but do not carry transit traffic, and transit AS that can carry transit traffic (along with local). The last type of AS intuitively are also multihomed.

Interior Gateway protocols use the IP address to construct paths for communication among one autonomous system, whereas Exterior Gateways Protocols use AS numbers to construct paths among different ASs. The former is called intra-domain routing, whereas the latter is called inter-domain routing.

2.3.1. Routing protocols for intra-domain routing

Routers are connected to multiple networks. Therefore, changes are that routers have multiple paths to certain remote networks. So the question is how routers select a best path to a destination? The goal of the routing protocols is to select the optimal path. But the term *optimal* is too general and allows for various parameters to be taken into account. Since it is already said that for intra-domain routing the IP address is used to construct paths to certain destinations, the selection of optimal path to each destination is done differently with different routing protocols. The parameters that can be taken into account are: number of routers that the traffic needs to pass (hops), delay, throughput, congestion level, policy, etc. Various routing protocols for intra-domain routing are devised, and each one of them factors in a various combination of these parameters. But generally, all of them perform routing within an autonomous system (AS). Three protocols are mostly used for this purpose: RIP – Routing Information Protocol, OSPF – Open Shortest Path First, and another one related to OSPF, called Intermediate System to Intermediate System or IS-IS (please note the dash sign to avoid confusion).

RIP protocol is in widespread use to date, especially in lower-tier Internet Service Providers and enterprise networks, and it simply takes into account the number of routers (hops) the message goes through. Messages do not exceed 15 hops and they are transmitted among neighboring routers in 30 seconds intervals. They are called RIP advertisements or RIP response messages, and they cannot contain more than 25 destinations with an autonomous system. This is usually more than enough for the routers to know all possible (and best) routes towards destination. Routing tables are maintained in the way that routers send regular advertisements to all neighbors. If however a router does not hear from a neighbor at least once in 180, then it considers

that particular neighbor as unreachable.

OSPF is another intra-domain routing protocol, and like RIP is also in widespread use, especially in upper-tier Internet Service Providers. It is a newer protocol than RIP and has been seen by many as a successor of RIP. It uses the Dijkstra algorithm and it has some advantages: it is a link-state protocol, which does not only send advertisements to its neighbors, but rather to all routers within the autonomous system. This way it creates a tree to each router in the AS, and puts itself as a tree root. It is important to stress that OSPF does not set a policy on how weights for links are assigned – that is for the administrator to decide. She may decide that each link's cost is 1, and this way it practically operates similar to RIP, counting hops and selecting the route with least hops. She can however assign weights differently – for example assign more weight (hence more changes to be selected) to the link with highest bandwidth. This way fiber optic links will always be chosen before coaxial or twisted pair links.

Another important thing about OSPF is that since this protocol sends advertisements to all routers in the AS, it causes a considerable amount of compute work initially, because it needs to draw a complete map of all routers in the AS based on all advertisements it received at once. But the topology it creates is very robust, and it sends advertisements only when there is a topology change, or at least once in every 30 minutes.

IS-IS is another link-state protocol and very similar to OSPF, developed by the long time forgotten Digital Equipment Corporation (DEC) [17]. It too builds a full network topology map of all routers in an AS using Dijkstra algorithm to compute best path in an AS, based on advertisements it receives from other routers. However, the main difference between OSPF and IS-IS that the OSI (or TCP/IP) layer they operate at. OSPF uses the IP protocol to send advertisements therefore it is a layer 3 protocol. Due to the popularity of IP, OSPF has also been used extensively. Whereas IS-IS is a layer 3 protocol, and therefore it is neutral towards layer 3 protocols. This gave the potential to IS-IS to have an advantage over OSPF (version 2) when IPv6 came around – OSPF was designed to be best suited in IPv4. But, OSPFv3 was developed to overcome this issue.

2.3.2 Routing protocols for inter-domain routing

While all the above mentioned protocols are used to intra-domain routing (within one specific Autonomous System), Border Gateway Protocol (BGP) is a de-facto protocol that is used for purposes of traffic routing among multiple Autonomous Systems. The latest version of GBP, BGP-4 is over a decade old now (since 2006) and is defined in the IETF's RFC 4271. This protocol version introduces mechanisms for reachability among autonomous systems, as well as dissemination of network information with all other routers within the Autonomous System. BGP gives mechanisms for loop prevention, and policy decisions are enforced on AS level. This feature sometimes is limiting though, as we will see when we use SDN applications for inter-domain routing, thus making SDN much more practical for this matter.

Generally BGP enables each subnet to be visible in the Internet - something otherwise not possible. It also allows for Classless Interdomain Routing (CIDR), thus eliminating the concept of IP address classes, and it also introduces means for route and Autonomous Systems path aggregation [18].

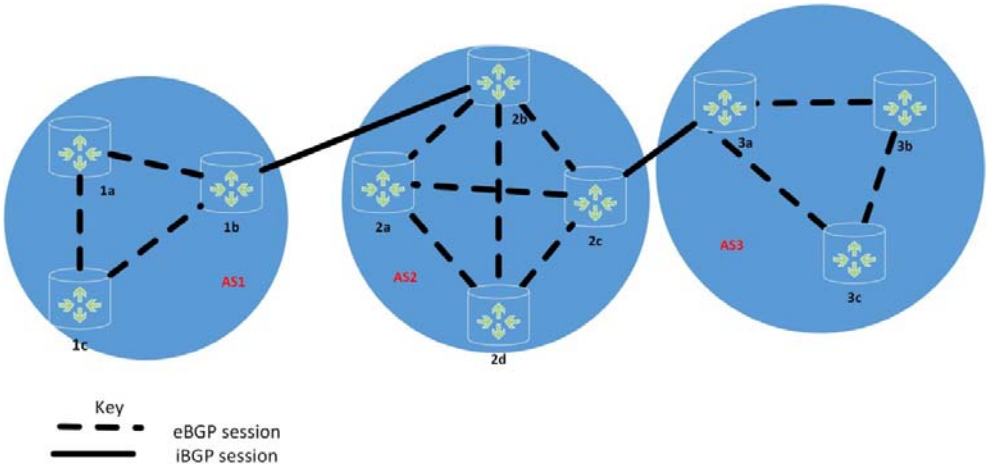


Figure 5. eBGP and iBGP sessions

As it is shown in the Figure 5, with the BGP protocol pairs of routers belonging to different Autonomous Systems are connected. This connection is done over TCP protocol with semi-permanent connections. When routers connect in this way to exchange routing information over BGP it is said that the routers have established a session. These sessions (connections) can be internal or external. External sessions (or eBGP sessions) are those by which routers of different Autonomous Systems are

connected. These routers are also gateway routers. Once they receive information from another Autonomous System, they forward it to other Autonomous Systems, but also to all other subnets within their own Autonomous System. The latter are called internal sessions, or iBGP sessions. BGP, as it is said above, uses CIDR to forward datagrams, so this means that if a gateway router is to advertise multiple subnets from one or more Autonomous Systems, they perform aggregation. On the other hand, an Autonomous System may receive multiple and different information about how to reach a specific subnet, each advertising a different path. To resolve any ambiguity, BGP uses longest-prefix matching to perform forwarding, thus selecting the more specific prefix, which inherently is the best path to that specific destination.
BGP is a very complex protocol and numerous books have been written about it. In general, it can be only mastered after a solid experience of work with it.

2.4. Complexity issues in data networking

In order to explain SDN and the rationale behind it, it is important to draw a comparison between networks and software systems.
A software system is a modular system based on abstractions to solve problems. A modular system allows for code reuse, change implementations and separate functions. In general, when one wants to solve a problem, the best way is to come up with abstractions, which in turn means to decompose the problem into its basic components, and then each components needs to have its own abstraction. These abstractions require an implementation to solve one specific task. Based on the complexity or the hardness of the task, it may again require going back the step one, until the implementations solve tasks that are easy to implement.
So historically, computer systems (for instance in the mid 70ties) were vertically intregrated in terms of hardware and software. The level of comlpexity in programming therefore was high, as software was tightly coupled with hardware (For example IBM mainframes that came with their own proprietary software, instead of general purpose modern operating systems of nowadays). This made programming much harder and much more complex than it is now, with high level programming languages.
Due to abstractions and modularity used in programming, the programming world has rapidly changed and simplicity has been abstracted. So when programs are written,

the programmer does not have to worry about memory management and addressing of bits, since it is solved by the operating system's memory management unit (that is defining a higher level abstraction for memory to deal with that piece of the problem), etc. Once the problem is divided in tractable pieces, the programmer is left to deal only with the inherent level of hardness to write the code that solves a specific task (the application itself), without worrying about related problems, such as memory management.

In comparison, the issue of complexity has not been addressed in data networks the same was it has in programming. The following section explains this right after it gives a brief introduction of the functions (and the planes) that routers have.

"Software-defined networking (SDN) is a modern approach to networking that eliminates the complex and static nature of legacy distributed network architectures through the use of a standards-based software abstraction between the network control plane and underlying data forwarding plane, including both physical and virtual devices." - NS3 SDN [19]

Software Defined Networking is not a revolutionary technology; it's an organizing principle in data networks. The rationale behind SDN is more important than its design. In 2008 the SDN elements like the Network Operating System (NOX) and OpenFlow switch interface were defined [20, 21, 22]. In 2011 the Open Networking Foundation was founded and now it has over 90 companies, among which Google, Cisco, Dell, IBM, Intel, Facebook, Verizon, Arista, Brocade, etc. [23]. Google publicly announced that they will use SDN for their interconnecting their data centers in 2012. SDN is now commercial and in production, although not so widely.

2.4.1. Data and control planes in networking

In data networking there are two planes: data plane, which processes packets with local forwarding state. The forwarding decision here is done based on the forwarding state compared to packet header.

The second plane in data networking is the control plane. It puts the forwarding state in the networking device, based on many possibilities and implementations. It can be computed using distributed or centralized algorithm, manually configured, etc, but

regardless of this, it is a completely different function as opposed to the forwarding plane [24, 25, 26].

Data Plane

The abstractions that we have for the data plane are basically known to every network engineer or even computer scientist.They are known as the protocol suites and the layering system used in data networking, namely TCP/IP. This layering model has been very successful in decomposing the problem in its basic components, and each implementation in this model solves a specific task. Applications are built on reliable end-to-end transport, built on best effort global packet delivery (network layer), which in turn is again build on best effort local packet delivery (link layer) on top of physical delivery of bits. Each layer is separate on top on the layer below [26].

The control plane on the other side doesn't have abstractions. We have a lot of mechanisms, which serve different goals, such as routing (a family of algorithms), isolations (VLANs, firewalls, traffic engineering, MPLS, etc.), but there is no modularity. And the functionality is limited. The network control plane is what happens when there are mechanisms without abstractions. So there is too many mechanisms without enough functionality. Each problem is solved individually and from scratch. This is not the way problems should be solved. Instead the problem should be first decomposed. That is the main reason that has lead to a great success and acceptance of Software Defined Networking [27, 28].

SDN imposes a big change in the industry just like computer industry was changed in the late 80es. At that time the computer industry was based on specialized hardware, specialized OS and specialized apps (usually all from one vendor, namely IBM). The computer industry was vertically integrated, close proprietary and relatively slow to innovations. With the microprocessor, an open interface led to many operating systems and a huge number of apps on top of these Operating systems. Hence, this industry moved from closed and very difficult for innovations, vertically integrated and proprietary, to horizontal, fast innovation and open. Networking too has for a long time worked in a same way, based on specialized hardware, software and features. SDN in essence offers the possibility to network programmers and third party app writers build anything they want on top of both router chips (data plane) and the Network operating system (now through OpenFlow, but it may be something else as

well) in the control plane, as well as on top of the Network Operating system due to the open interface it introduces (the control program).

Control Plane

The Control plane in a network should compute the forwarding state under three constraints [24, 26]:

1. The forwarding state should be consistent with specific low level hardware/software,
2. It should be done based on entire network topology, and
3. It should be implemented in every router.

To take care of these constraints, network designers should define specific abstraction for each problem component.

1. The compatibility with specific low level hardware/software needs an abstraction for a general **forwarding model** that hides the details of specific hardware/software;
2. Being able to make **decisions based on entire network** takes another abstraction for the network state, hiding the mechanisms to get it; and
3. Another abstraction that deals with the **actual configuration of each network device**, so they are configured in a much easier and straightforward way.

The Forwarding Model in SDN

For the forwarding abstraction, we want to hide details of the type of hardware or software the decision in used at. The device itself may by manufactured by any device manufacturer and still the model should work in a same way. **OpenFlow** is the current proposal for that [25, 29]. It is a standard interface to a switch so we can access the switch and we can store there flow entries through this protocol. It is a general language which should be understood by any switch. Conceptually this is a pretty easy and straightforward concept, although its practical implementation of design details (like header matching, allowed actions, etc.), may not be so [30]. The forwarding abstraction (implemented through the OpenFlow protocol) exploits the flow table in the routers and populates them with simple rules (if header x, forward to port y, etc.). It does a:

match + action,

in a similar way it is done today in networks. The set of actions in OpenFlow is rather small (forward packet to specific or set of ports, drop the packet, or send it to the control plane, and also define bit rate at which packet will be forwarded. The most interesting thing to look at with OpenFlow is rather the *action* part of the function, because the desired goal here it to use a minimal set of actions to be a good enough set of actions to do most work on one hand, and offer the possibility to chip vendors to implement it and program writers to offer special features that makes them unique in the market, on the other hand. Eventually the OpenFlow should offer protocol independence to build different types of networks, like Ethernet, VLANs, MPLS, etc, and new forwarding methods which are also backward compatible and technology independent [31, 32, 33, 34].

The Network State in SDN

In routing, we want to abstract the way how the distributed algorithms get the global network state, and instead only give a global network view, annotating things and information relevant to the network administrator, like delay, capacity, error rate, etc, so the network admin is able to program the switches the way they desire. It is implemented through the Network Operating System in SDN, which runs on servers on the network, and the information flows in both ways, which means the servers get the information about the network state by querying the switches to create the global network view. Based on what policy we want to implement in the switches, we do it in the opposite direction (router configuration). The Network Operating System gets the information from the network routers to create a global network view, and then a control program implemented on top of the Network Operating System implements policies about routing, access control, traffic engineering, etc. This is a major change in the networking paradigm, where the control program implements the policies into the routers [24].

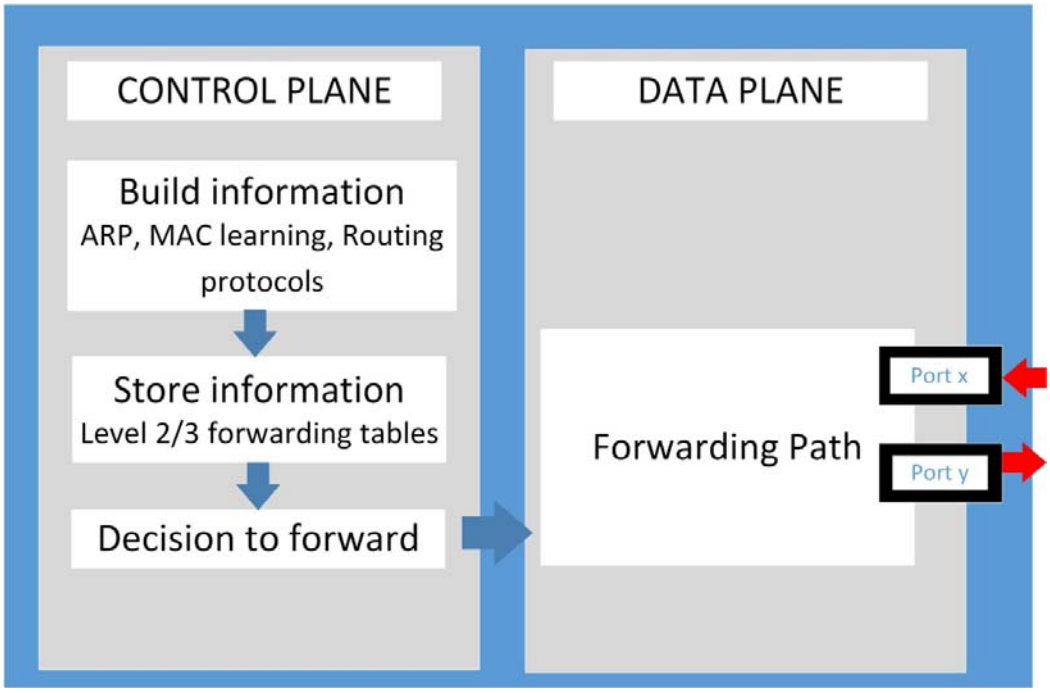


Figure 6. The Control Plane running on the edge, separately from the Data Plane in a SDN.

The control program needs to install the policies and flow entries intro each router in the network. But the control program needs to just express desired behavior, and not be responsible for specific statements. Another abstraction deals with writes of specific statements in routers. Rather than the control program dealing with the actual full network topology, it only deals with a virtual layer created on top of the full global network topology. This way each part of the problem is decomposed in a clean way, and each abstraction deals with its own task. That is, the control program expresses the desire for some specific network configuration on one or more routers, and the specification virtualization layer does the actual mechanics of implementing it on the actual routers in the network.

The SDN's achievement is not to eliminate complexity because the layers and the network operating system are still very complex and complicated. Its achievement however is to simplify the interface to the control program so that it has a simple job to specify what we want to do with the network. The hard part is the reusable code, and once it is done right, it will be used by any network programmer without the concern of knowing the details of its implementation. That part is implemented in the operating system and the control program, which is the reusable code. The

comparison here is with programming languages and compilers. The programmer needs not to know how the compiler is implemented, not should be familiar with the instruction set, because in programming those two problems have been decoupled a long time ago. This is something that has not yet been done in networks, and that is the major goal SDN accomplishes in the data networking field [28, 29, 35, 36].

2.4.2. Applications of SDN

In today's networks we can easily says that topology is policy, meaning the actual physical location of routers, firewalls, etc. dictates how effective the network is, how well the ACLs work, what our broadcast domains are, etc. ***When networks are moved to the cloud, we usually want to keep same policies***, but very few networks operators have an abstract expression of network policy, rather they have a network topology. SDN allows specifying a logical topology to the cloud. The cloud then ignores the physical topology and follows the logical topology based on the policy read based on the topology initially.

The function is evaluated on an abstract network and only the compiler needs to know the actual physical network topology. The major changes that SDN brings to the networking world in general is not the easier network management (which comes as a result of it), but rather primarily decouples the data plane from the control plane. They are now the same in terms of vendor, and place where they are implemented. This changes the business model in networking (hardware bought separately from software, which can be third party). But it also brings a clean interface which allows for much easier implementation of testing of networks in the split architecture.

1. Simplified Network Troubleshooting

SDN allows for implementation of control function in the edge of the network, instead of in the routers, where it actually is, and the core only deals with delivering packets end-to-end. The core may easily remain legacy hardware and the network operators need not to know at all the one is implementing SDN from the own edge of the network. So it simplifies network troubleshooting and also there are no disruptions periods during network convergence, due to the fact the one policy is implemented per specific flow and each packet will be carried either by an old state, or by a new

state. In traditional routing packets may well be lost during network convergence, or loops may appear. In SDN the expressions are high level and easily checked and corrected.

2. Network customization

The SDN gives an extra benefit to the network operators to more easily customize their networks based on their needs. The policies, traffic engineering, monitoring and security is easier to implement after getting a network state quickly from the Network Operating Systems, and moving the network to a virtual environment without much effort is perhaps the greatest benefit in this specific scenario. An example would be to improve load balancing in a network with many servers in different locations connected though a backbone. The load balancer used in networks today chooses a lightly loaded server to do the job. But since servers are in different location, the load balancer does not take into account to choose the lightly loaded path too. Ideally we would like to choose both, for a best result. SDN not only gives the possibility to do this, but more importantly, because the control plane now resides on the edge, it can be written by anyone and is vendor independent, it can be done in a very short period of time and tested in real time.

3. Third Party Apps in Networking

Also, network operators may hire third party people to develop special features for their networks, as well as remove unneeded features from routers. Removing some unused features from routers arguably increases their reliability of routers.
SDN offers a chance to increase the rate of innovation by moving the operation in software; standards will follow the software deployment instead of other way around. Another great opportunity here is experience, technology and innovation between vendors, universities and researchers.
As of now, there are already different domains where SDN has been or is under implementation, including data centers (Google for example), public clouds, university campus networks, cell phone backhauls [31, 32, 37, 38, 39], but also in enterprise Wi-Fi environments and home networks. There are already over 15 vendors offering SDN products, and probably the number will grow, including new jobs in

this field.

SDN is an opportunity to program the network infrastructure in an easier way, by offering network wide visibility as well as direct control through OpenFlow.

Another significant advantage in SDN is that OpenFlow offers the ability to innovate on top of the low level interface that today's controllers provide, by increasing the level of abstraction. Today's controllers do not have a complete network wide view primarily for scaling purposes, whereas in SDN the control program has a complete network wide view, and it can actually use the virtualized information they need for the network state. It is hard to compose different tasks in today's networks (like monitoring, access control and routing).

2.5. Summary

This chapter introduces basic networking and network routing concepts, such as network devices, and specifically routers. It also explains in more detail routing protocols, both protocols for intra-domain routing (within same Autonomous Systems), as well as protocols for inter-domain routing. In addition, this chapter elaborates the problems of ever increasing complexity in managing data networks, and the problem of lacking a well-defined abstraction for solving control tasks in data networks. The control plane rather does not have a structure, such as the data plane in networking, where the layered model decomposes a complex task into multiple less complex ones. In the end, a few typical scenarios of SDN implementations are given as common scenarios.

CHAPTER III. SOFTWARE DEFINED NETWORKING vs. TRADITIONAL NETWORKS

3.1. The skepticism towards a revolutionary paradigm explained

The reasons for having the current Internet architecture the way it was developed decades ago are very strong. The historical perspective of over five decades of evolution of this architecture is tremendously rich in research and innovations. While most of the proposed technologies over the years have failed and very few have been adopted, one thing is true – that is the Internet architecture is prone to change very slowly. Some of the arguments for this claim will be stressed shortly. While contrasting arguments for a need for quick and comprehensive change in the architecture (i.e. start from scratch, revolution) vs. arguments for slow change (evolution), one needs to stress the basic principles on which the current Internet was developed, and well as stress the major reasons why the current reality requires a different view, and poses the needs for a possibly new architecture.

This work tries to summarize the differences between the requirements for the Internet of 50 years ago and the nowadays Internet; the experience during this time that had led to the need for making a fundamental change in the current architecture, as well as discuss some of the already proposed solutions to some of the issues.

While there are sound reasons for a call about a new architecture that would address current issues with the Internet, two things must be considered in order to be able to say that the current Internet generation should be replaced by a new architecture: 1. how strong are the reasons for changing it?; this view takes into consideration the costs and benefits for replacing a whole architecture with a new one, and as long as patches can be provided for issues that do not concern a vast majority of Internet users anyway, chances are that patches are more feasible, and 2. if the idea for a completely new architecture is embraced, are there guarantees that these issues will be solved, and new issues related to it will not emerge? For example, one could argue that if intelligence is added to the core of the network, generality cannot be preserved, possibly limiting development of new applications, etc.

Historically, the aims for the DARPA Internet Architecture [40] were to have a suite of protocols (known as TCP/IP) for packet switched networks. Also, this includes

multiplexing techniques for interconnecting existing networks. Another alternative to packet switching was circuit switching. Many researchers would argue that circuit switching offers more in the sense of network delivery guarantees, delay guarantees, etc., but on the other hand the networks that were to be integrated in the network of networks were packet switching networks. Not only that, but another aim was to have a stateless network, which during failure would be able to reconstitute quickly, without interruption in the communication. Most of the goals set at the time of beginning of the Internet, are still very sound. For instance most of the new proposals for new architectures include the need for communication non-interruption even if some nodes fail, or are attacked. They also request that any new architecture should support different types of services, and be able to integrate variety of networks. But, being it a network developed primarily for military purposes, the importance of different goals was different from the one that is now, and some of them, understandably, were not even foreseen. For example, few people might have guessed that this will be an internetwork that will host billions of nodes worldwide. This bare fact imposes many issues – trustworthiness and security, network space and addressing availability, location, naming, the end-to-end principle, etc. As a consequence, the Internet evolution has addressed some of this issues, some successfully, some of them without any success, and some of them have introduced new issues in the picture (for example the introduction of private IPs has addressed the addressing issue, but has contributed in violation of the end-to-end argument).

The following sections discuss the abovementioned aspects one by one, with an explanation of the issue followed by ideas that have been proposed as solution to those. We will focus on key issues, without discussing details about things they might raise.

3.1.1. The principle of evolvable Internet

One key characteristic of the Internet architecture is that it allows the system to evolve. In other words, the architecture is minimalistic and general, broad to support applications that are not developed yet; support technologies that are different from one another in many ways. This specific of the Internet has made it so robust that it has prevailed drastic technology improvements in all the years.

While some proposals argue whether the New Internet should be built from scratch or

change the core of existing architecture [41, 42, 43], one thing that remains clear is that whatever it is, the New Internet should maintain the generality in mind [44], which will allow support for future technologies and applications. After all, this is an easy statement, but it is hard to implement, since it includes non-existent factors (factors that belong to the future, which are unknown). For example, how general should be this architecture? In the meantime, what are the compromises that need to be done for this benefit? If this generality involves too many options to extensibility, which most of the times will not be used, or are difficult to implement and include a big overhead, it will clearly be of no benefit.

From past experience and evidence, one could argue that the IP packets are one of the biggest advantages of Internet architecture [45]. Regardless of the technology or the medium, IP packets have driven the innovations of different applications over the years, and thus should not be replaced with a different implementation. IP packets may not be the most appropriate method that optimizes in a best way the use of a specific technology, but here comes in the argument for generality; clearly this is the price for being able to allow different networks to be integrated together. Technologies should (and do) change rather fast to suite the architecture, not vice versa. For instance, the best effort delivery of IP packets does not include any delivery or delay guarantees, but this has improved a lot over the years with different technologies, starting from 10Mpbs, 100Mbps, 1Gbps, to 10+Gbps. This is a proof of how technology can change so it will meet the rising needs better, and yet, it multiplexes packets in a general and mutual infrastructure, without having to change the architecture. By this, we want to show that there are things that can be done with the existing architecture that will address some issues; but there are still some major problems that are much more complicated than this. Some of these elements will be discussed in the following sections.

3.2. Issues with existing network architectures

This section addresses some of the issues of the current Internet architecture and the question raised by the National Science Foundation about the next Internet generation called Global Environment for Network Innovations [46] (GENI). Taken into consideration that the initiative has encouraged the research community to come up with different ideas on how the next Internet generation might look like, it is our

understanding that some of the proposed ideas have very strong arguments and theoretical ground for the suggestions that they make, and yet most of them fall short to take into considerations the risks that those ideas could bring (the issues will be treated one by one in the following section). Without getting into much detail, this research work tries to prove wrong some of them as too optimistic and too risky for a serious consideration until they show practical results in the ground.

3.2.1. Security

Perhaps one of the most significant reasons for the calls for change of the architecture is the security aspect. One could argue that the current architecture does very little when security is concerned, and concerns have been raised regarding security - Denial of Service attacks, as well as Internet fraud and spoofing (and sometimes all at once). It should also be mentioned that not only individual nodes are attacked, but the network itself as a whole can be a target; not only is the security an issue, but the trustworthiness as well. Over a billion users interact so much on the network nowadays, that in most cases they are not aware of the host node they are in contact with. This is a major philosophical change on how the Internet is now perceived. The original Internet design has not considered this matter. In fact, decades ago the assumption would have been that individual end nodes would be willing to communicate. This is clearly not the case today (who is willing to receive spam email?!).

However, this situation is not without a reason. The network does very little about security, because it is very transparent (although network transparency is broken on many occasions, which will be discussed next). Packets leave the network in the same way they enter it – they are not modified (the end-to-end principle which is now violated; this principle will be discussed in the next section). Many would argue that the stateless nature of the network is not sufficient, and instead routers should be able to keep some state, and also distinguish flows in the network. These demands might go against the principle of network simplicity, and clearly go against the end-to-end principle, and the use of datagrams. Basically, if one node is currently attacked on the network, the node is left on its own to deal with the attack, because no intermediate en route hop does anything to prevent this attack. In particular, when discussing Denial of Service attacks, here the attacker wastes victim‟s recourses (the victim is often a

server) to the point that no other (good) user is able to get any service from that server. One model suggests a proactive approach in defense, i.e. by offense [47]. What this model suggests is to isolate the attackers by requiring the good clients to increase their upload bandwidth. This, so called „speak up" model, does not guarantee any success, at best. It also makes the good clients busier with upload, which they might not desire. And it also adds total overhead to the network.

Traceback solutions are also proposed to solve the problem of Denial of service attacks [48, 49, 50]. This is also very difficult to rely on, because in such a case, the victim has the IP source address of the attacker, which might be spoofed. One of the flavors of such a technique is Probabilistic Packet Marking. The idea behind this is that the router marks the IP address packet in a probabilistic fashion. Since the attackers send a big deal of packets to attack the victim, eventually the victim should be able to reconstruct the graph to the attacker on basis of those marked packets. One weakness of the model is that the marking fields themselves may be spoofed (it should be as easy for the attacker to spoof the marking field, as the source address). Another weakness is that different spoofing methods, like enhanced random spoofing, and topology aware spoofing can be very successful as attack techniques, with almost no chance to be captured from the attacker. The weaknesses of this model are best described in [51].

These problems (which are much more difficult to deal with) have led the research community to propose completely new models, new architectures, which will try address these issues.

One proposal suggests that the layered architecture is not anymore an adequate foundation for networking architectures, due to the fact that layers have already been violated by middle boxes (NATs, firewalls, caches, proxies, etc.), thus it suggests a non-layered role based model architecture - RBA [41]. In this architecture, the communication would be organized in functional units – roles, instead of hierarchically. The proposal admits that role violations are possible, however.

This idea suggests a completely new architecture, which is very difficult to run for, especially because there is an IP layered architecture in place that has served well for many decades; there should be a good reason to change it now – the fact the role violations are possible here, it is very difficult to prove that this architecture is a better one.

But, if we disregard that fact for a moment, perhaps the biggest drawback is that in a

non-layered architecture there is basically a huge number of „layers". This type of architecture would run different protocols for different purposes (connection, routing, control, security, management, etc.), which will make the network very complex, that even if it still remains general, it will take too much time and effort to set up connections, deploy new applications and services, most of them users will not be able to adopt easily.

One good reason for the huge increase of Internet usage over the years is the simple and general architecture of the Internet. The hourglass model of TCP/IP is the simplest way of network deployment – all that is needed to use the network is the IP address. Everything else is up to the developer and the user. Just as a comparison, the telephone network, although much older than the Internet, has not been able to evolve into nothing else but a telephone network, exactly due to the complexity that exists in the network itself [52].

So, when discussing the need to change in order to meet new demands from the new era, one should inevitably admit the advantages of the current architecture like the IP model, and engage to keep them as such.

Having said this, this work still acknowledges the major issues of the current architecture, security being one of them, but instead, the solutions should be found within the existing architecture (be it existing or new methods). Two things that will contribute to the security are trustworthiness and stricter laws and law enforcement.

The trustworthiness is not a new concept, but has never been treated thoroughly and with participation from all sides concerned. Recently Microsoft has initiated a larger debate on the principle of end-to-end trust [53]. The strategy gives three directions in which a higher trustworthiness (as a broader concept than security) can be achieved. First is creation of a trusted stack where security is rooted in hardware and where each element in the stack (hardware, software, data and people) can be authenticated in appropriate circumstances. The second one involves managing claims relating to identity attributes, which means that identity claims need to be passed (sometimes names, sometimes an attribute like proof of age or citizenship). Some important technologies, such as public key infrastructure (PKI) and smart cards are now mature enough for broad deployment, and this is meant to ensure higher security in the Internet.

The Internet, in order to be safe, needs to support the option of identities based on in-

person proofing, enabling the issuance of credentials that do not depend upon the possession of a shared secret by the person whose identity is being verified. To some extent, government activities and markets themselves are driving „in person proofing" regimes. For example, governments are issuing (or considering issuing) e-ID cards for government functions.

This idea does not require any architectural change in the Internet.

Finally a good alignment of technological, social, political and economic forces so that we make real progress is needed. The nontechnical aspect (laws and regulations) is discussed later, and is unavoidable aspect of security, even in some future architecture. Besides, „intelligent" future Internet with higher security premises cannot guarantee non-existence of violations in authentication, eavesdropping from authorities (or higher managers in a realm), and preserve the principle of network generality. These issues are discussed next.

3.2.2. Network transparency

The network transparency has been widely discussed in recent years, and the fact that the network provides no security (hence is transparent) has led to the suggestion of controlled transparency [45]. The basic idea behind this proposal is that due to the fact that network users not always know their „interlocutor" on the other end, sometimes they need to protect themselves better than other times. In such cases, the network transparency would decrease, and (hopefully) the malicious user will not be able to utilize the fully transparent network to attack the good user, since the network will block the bad user. In cases where users are authenticated and there is trust among them, the network will continue to be fully transparent.

This idea sounds fine, but there are also questions that are associated with this idea, which make it more complex; we can only guess some of the questions that might be raised with respect to this idea. First, will this violate the end-to-end argument? (The end-to-end argument is discussed in the next section and the explanation to why the answer to this question is yes); second, are we willing to move from a „de facto" to a „de jure" violation of the network transparency? In other words, are we going to institutionalize the violation of the end-to-end argument by deploying the controlled transparency? The final question is how and who is going to perform the controlled transparency? Obviously, behind each network element there is someone (human

being, company, government, etc.).
To better address these questions let us stress the end-to-end principle that has been a very important characteristic of the Internet for many years first.

3.2.3. The 'End-to-end' principle

This principle clearly states that devices communicate directly among each other, and the network provides the means for that communication. The network itself is not and should not intervene in a whatsoever way in the content of the packets sent over it – it remains dumb. All application specific functions should be done on the end nodes, even if it is possible to implement such function in the network, it is not recommended to do so [51]. There are many reasons for this. This way the network remains simple and easy to manage. It is also easier to upgrade the network and different technologies on that network if it is kept simple. It also enables to deploy more new application on the network much easily without having to change the core of the network. This is the principle of generality, which is the main reason for the development of all applications so far over the Internet.
The Internet is not fully transparent today. The use of firewalls, NATs, caches, etc. has been violating this principle for some time now. Firewalls are used to protect one end system or a part of the network (possibly local network) in an on/off mode. All communications that seem insecure are rejected from and to the outside Internet. This is a primitive way of protecting a part of the network, and it still does not protect from internal attacks. Firewalls are clearly implemented in the network, between two end nodes that might want to communicate. However, the principle of evolvable Internet has not given a better answer or a solution to this issue so far in an effective manner. Firewalls are used largely nowadays, and therefore we maintain that the end-to-end principle can be violated in such cases, but this remains an exception, especially since firewalls do not add any complexity or intelligence to the network besides breaking the end-to-end communication.
NAT (Network Address Translator) boxes are another type of element that violates even more the end-to end principle. Originally, the goals for their implementation were to solve the problem of running out of public IP addresses. NATs allow end systems to be assigned private IP addresses, which are recognized and used only within a part of the network, but not on the outer side of the NAT. This way, packets

from different end nodes from the local network that need to leave this network, are assigned a new IP address from the NAT that is public and globally unique. Thus, NATs not only interrupt the clean end-to-end communication between two hosts, but they also change the content of the packets. Along with the IP address, the port numbers are also changed from the NAT. Therefore, hosts communication between them, actually communicate with the NAT, and the NAT is the „entrepreneur" that brings together the two ends nodes, without them being aware of this fact.

The IPv6 and its slow deployment is expected to solve the problem with the IP addresses, and its full implementation will not take much longer [54] (the IPv4 addresses are really running out now), and perhaps this will slow down the further implementation of NATs. As many developers and content providers cannot use NATs due to the fact that their servers cannot be seen from behind the box, it remains to be seen that these boxes will eventually be used less and less on the Internet.

Caches are also elements that violate the end-to-end principle in a more subtle way. One simple way is when one visits one website, some (or all) of the contents might be placed on a different region closer to the user, so future accesses are quicker at users" convenience. One example of this type of implementation is Akamai [42]. The simplified explanation of Akamai is that content is distributed over to different servers in different locations, making access to content quicker and more reliable. Another example is search engines – for a same keyword search, a user might get a different answer from one city (or country) than a user from a different one. This is done based on IP location, and is specific to different factors, like market, people"s interest, etc.

In both cases just mentioned above the user thinks that they are communicating with a single end system, but the reality is quite different. The type of cache orientated design is a change of the original pattern where exactly two end systems communicate with each other.

Current trends show that we cannot rely on the pure end-to-end principle, nor it seems to be the goal. While the deployment of IPv6 will address some issues, like the local IP addresses introduced with NATs (which is very important), one cannot hope for a pure end-to-end and fully transparent network.

This fact has led to some more drastic proposals for change in the core of the network. One idea proposes a new architecture, where this network transparency is as low as possible at a level, where users need to authenticate first before being able to access the network [41]. In other words, without explicit authentication, one cannot access

the network. From what was stated earlier, it is clear that some type of in-person authentication can be made even in the current Internet, without the need to modify the network core. While it is the willingness of all parties concerned that will decide if this trend will take off (ISPs, governments, businesses, users, etc.), it should remain clear that the authentication (which is not binary, but differs upon occasion) should only take place when necessary (like money transactions, personal data, etc). This brings up the other element that authentication brings - privacy. By privacy, we mean that while users should be responsible and accountable for their interactions on the network, there should still be a more subtle way of identity check (if necessary), instead of explicit disclosure. The privacy in general (or even anonymity), cannot be compromised a priori and in its social context is a highly appreciated value in many societies, and should be revealed only when it is necessary, but not always and for no reason.

Two other elements should be taken into consideration when we talk about changes in the network: 1. The social factor and 2. Necessity for simple network.

3.2.4. The social factors

By social factor we mean that there should be persistent effort to bring the Internet closer to the public. The expectations of big increase in access to the Internet are still not being met, and different parts of the world have very little or no access to the Internet – the tool that is meant to bring the world into every home. If, for some reason, these limitations come into picture, they can only discourage the usual users to utilize less the network, which might have substantial consequences.

3.2.5. Simplicity of the network

The necessary simplicity of the network assumes the simple IP model that encourages and enables a wide range of application to be developed on the network. One undisputable fact is that whatever change that might be made to the current architecture core with respect to increase of its complexity, it make it less flexible in terms of „runs over everything", regarding applications and technologies. This feature has been a driving machine of the Internet for many years, and has led to an economic and social revolution, and it should be given credit for that.

As far as the questions raised in this and in the previous sections are concerned, this research work maintains that the network should continue to be as transparent as possible. By this, we mean in cases where for practical reasons we might need some higher level violation of the end-to-end principle, as in the case of caches, a compromise can be made. This does not affect the privacy, and yet does not add state and much intelligence on the network – the routers will still do their basic job, which is routing, and the servers will not maintain any state. But, even in cases where this principle has to be violated, it should be done on a higher level, not in the network itself. One cannot guarantee that if the network core is changed (by adding state in the routers, leaving the best effort stateless approach, etc.), the generality will be preserved, and the network will remain as technology and protocol friendly as it is now.

As explained above, we keep a more conservative in its approach, and do not back the ideas for a revolutionary Internet, or architectures that are new and all different. Having stated this, some change is desirable, since the network should provide ground for new and future applications and services; some of them are not completely supported now (like Mobile IP), but there are advanced integrated solutions offered by software and telephone companies that address these problems fairly well (section 3.6); and some are still yet to come, thus the approach used so far – offering patches and repairing problems en route has been successful to a great extent, at least as far as current applications are concerned.

3.2.6. Quality of service

One issue that the current Internet architecture faces is lack of Quality of Service; that is the network does not make any guarantee about the delay and throughput rate of flows. Some would argue that due to its stateless environment, the network does not even recognize flows.

One proposal suggests is the future Internet should be able to provide QoS guarantees, and sometimes even total isolation [54]. Thus this idea suggests that both datagrams and circuits should be provided in Internet 3.0. Hence, in a shared wire circuits will be offered to those who are interested in delay and bandwidth guarantees, whereas datagram routing will be left for those interested in that.

While this approach is feasible from the technical aspect, some hidden risks are

associated with it. First, provided that circuits offer a better service (some guarantees), this must include the higher cost for the costumers interested in it (this is not bad at all – just as a comparison, there are people that fly business class at a higher rate, and people in economy class at a lower rate). Since circuits do not make best use of the resources that are offered, and instead sometimes they are even wasted (and cannot be allocated to someone else as long as the circuit is not disconnected), there is a risk that this might influence the datagram routing seriously. As a result, users that do not have quality of service support, might start to get way worse service that they do now. Practically, it does not cost much big businesses and corporations to buy out high rates of bandwidth and data rates over circuits, and individual costumers could be damaged by it. Thus the combination of circuits and datagrams might be a solution to the existing problem for quality of service, but only if there is a balance such that datagrams do not suffer. Since these two approaches do not use the recourses in a „fair" way (that is use the resources reasonably and responsibly), and weakest of two (which is datagrams) will be left on the mercy of the other one. Besides, having only datagrams (as it is now in the current Internet) sometimes requires higher bandwidth and better lines (even though datagrams utilize the network in a best way), let alone the fact that now datagrams have to share „fate" (medium) with another service, such as circuits. One can always argue that more bandwidth from the ISP can solve this problem, but it is not the ISPs that pay for that, it is the end users.

Thus, without serious analysis and numbers that might prove the opposite, we believe that the approach of shared medium for datagrams and circuits can cause more harm to the net, by satisfying only some users" needs. We maintain that only datagrams is still the best approach, and it should not be changed in the future Internet. Perhaps with some more investment on the Internet infrastructure, and more optical links deployment, the data rates will increase, at most users" convenience; for those users with higher needs for guarantees, the private line VPNs are also available; A network architecture should not be changed due to the need to replace private VPNs with core circuit switching on the Internet – this only shows that this is not a major problem, and even now there is a solution to that.

3.2.7. Mobility and naming

It is clear that IP addresses should serve a purpose, and it is addressing. Today, IP addresses express both network location and node identity [55]. IP addresses should represent location, and as one node moves from one place to another, IP addresses are assigned dynamically. This approach is acceptable and should therefore not be changed.

The original Internet architecture was designed to provide unicast point-to-point communication between fixed locations. In this basic service, the sending host knows the IP address of the receiver and the job of IP routing and forwarding is simply to deliver packets to the (fixed) location of the desired IP address. The simplicity of this point-to-point communication abstraction contributed greatly to the scalability and efficiency of the Internet. However, many applications would benefit from more general communication abstractions, such as multicast, anycast, and host mobility. These abstractions have proven difficult to implement scalably at the IP layer [56, 57, 58].

As the number of mobile devices is growing, the need to identify these kinds of devices in a unique way while in move is growing. While Mobile IP has attempted to solve the issue of mobility, it has been difficult to deploy it due to issues including scalability.

More specifically, the need for naming is much bigger with servers than it is with clients, since the demand and server access is much bigger. Basically, this is an issue that is already emerging – naming data and services, instead of nodes. The current model is a legacy of the first Internet, where Internet applications of that time, like file transfer and remote login were part of the host-to-host communication design [42]. Needs are changing lately, and instead, today most Internet applications include data and service access, and a service (like CNN, MSN, etc.) might include multiple hosts and multiple locations. Certainly, the existing architecture supports this kind of access, although issues like availability might arise.

Akamai is an example of a successfully developed idea of mirroring content (usually media content) stored on costumer servers. Although the domain name in this case remains the same, the IP address directs to an Akamai rather than the costumer server. This Akamai server is then picked based on the content and the actual location of the user‟s network. This example shows that content delivery on the current architecture is not as problematic as once thought, and with additional ideas put in practice, data

and service availability can be also reached.

A solution for naming replicas (mobile data and services) has been proposed in [59]. This idea introduces the idea of Human-Friendly-Names (HFN); a scalable HFN-to-URL resolution mechanism that makes use of the Domain Name System (DNS) and the Globe location service to name and locate resources. Using URNs (Uniform resource names) to identify resources and URLs (Uniform resource locator) to access them lets end users use one URN to refer (indirectly) to copies at multiple locations. To access the resource identified by a URN, a way to resolve that URN into access information, such as a URL is needed.

Because a URN refers to a resource rather than its location, users can move the resource around without changing its URN. A URN can thus support mobile resources by referring indirectly to a set of URLs that change over time. Because URNs identify resources to machines, they need not be human-friendly [60]. Unlike URNs, HFNs explicitly allow the use of descriptive, highly usable names. Replicating or moving a resource will not affect its name, for example, and a user can freely change the HFN without affecting replica placement.

A new idea of a network which proposes a completely new, called data oriented network can be found in [42].

As the demand for supporting mobile devices is rising, new ideas which not necessarily require a change in the network architecture are emerging. One example is the integrated Wi-Fi support in smartPhones. Arguably, Apple is in lead with this technology, offering the possibility to connect iPhone devices on available wireless networks, thus making possible direction communication with not only simple email servers, but Exchange servers as well using the push protocol for a direct connection with the server. These devices offer most of the necessary things for IP mobility nowadays: they are ready to support asymmetric protocols, they make use of proxies in case they are turned off or are in sleeping mode, thus enabling high energy efficiency at the same time, support for streaming data (voice and video) [61].

Currently, mobility is still best supported in the application level, although it is not yet clear if the trends will continue in this direction. [62] Thus, as one might argue that the network architecture should change in order to be able to provide for better mobility management, the other side of the coin is that the cell phone industry and the computer industry might integrate both networks into small computers and cell

devices, and make it much easier to deploy a whole range of services for these mobile devices. The applications for these devices are in an early stage, and in the following period we could witness a bigger exploitation of the possibilities in this field. It is clear that the emergence of P2P overlay networks has not been exploited enough, and much of the answers arguably may lie in the overlays. This is turn favors the application level mobility and multicast, which might take off in a near future. As demand is growing, especially for mobility, new ideas that can be built on top of current IP (as overlays) are coming in the picture, thus eliminating the need for a new architecture. One such proposal is „the Overlay-based Internet Indirection Infrastructure (i3) [63]. Authors propose a single new overlay network that serves as a general-purpose Internet Indirection Infrastructure (i3), which offers a powerful and flexible rendezvous-based communication abstraction; here applications can easily implement a variety of communication services, such as multicast, anycast, and mobility, on top of this communication abstraction. This approach provides a general overlay service that avoids both the technical and deployment challenges inherent in IP-layer solutions and the redundancy and lack of synergy in more traditional application-layer approaches.

3.2.8. Laws and regulations

Perhaps the experience in networking in the last decades, and certainly the experience from other fields of social life prove that one cannot claim absolute security and certainty in anything. Just like there is no guarantee that there won‟t be crime and robberies in the streets and banks, one cannot hope for a better situation in the networking world. Having acknowledged this, the networking community and the general public requires at least the enforcement of the principle of accountability for those who decide to violate the law. In other words, there is a need for identification on the network, such that authorities are able to identify possible illegal activities on the network, and take measures to sanction it. This discussion leads to the eternal debate of privacy vs. responsibility. In many countries, certainly including the United States, the principle of privacy is a strong value and has a high position in the value system [64]. In other words, entities in the network should be able to interact freely and without constraints, as long as they do not violate the law. This freedom includes the element of privacy (which not always means anonymity). And privacy is

determined based on the type of interaction that is performed on the net: individual subjects, people, businesses, etc. normally have less privacy as they move along the axis individual customer – content provider (with the latter one being less private). This is again, in accordance to social life – ordinary people enjoy more privacy than celebrities. Therefore, wiretapping and other sort of surveillance should be made possible on legal terms (technically it is possible), but it should be made known to the public who can do it, and under what circumstances (most people would not have against wiretapping of possible terrorists). It is clear that laws can be misused for this purpose, but the bottom line is that any kind of violation of this type should be dealt with outside the network. The opposite of this would be having people to authenticate every time they use the Internet, which may lead to privacy violations. Besides, this type of network would not encourage the growth of number of people with access to the network, but it will rather discourage it. The laws for cyber criminal do not have a long history, and they are too undergoing a process of evolution, with more precise definitions and sanctions for abusive behavior on the net, for Internet infringements, scams, etc. The level of legal progress is different in different countries, some being in the beginning phase of the process. It is governments‟ responsibility to act as closely as possible to harmonize the laws of this area, and certainly, for countries where it is not possible to treat cyber crime as illegal, other measures are still possible: credit card payments (say VISA) are not possible from certain type of countries that do not meet the standards and regulations for safe e-commerce. This sort of block is lifted once countries meet these standards. Clearly today's Internet needs to address issues like security in future, and as explained so far, we maintain that the best way to do this is by changes to the existing architecture (until accountability is ensured) and changes in the legal terms and better law enforcement. Technical changes only will never solve major problems, without causing other related harm.

From all the abovementioned analyses, one can conclude that the original Internet, dating from about 40 years ago has ever been evolving. As a result, some issues that could have not been foreseen are now appearing; some principles like end-to-end have been violated, and in the meantime new issues are emerging – identity and authentication, security, mobility, energy efficiency, etc. While recently there are calls for changes in the current Internet architecture, we stress the hidden risks and additional issue related to these, like network complexity, concerns about privacy, no

guarantees for security, and discouragement for Internet access to ordinary people, and discouragement for developing new applications. We maintain that the approach of patching the current architecture – firewalls, proxies, and other middle boxes are helpful and do not increase the complexity, thus do not endanger the evolvability and simplicity of the network. We believe that by better and more global laws the security issues will be minimized – this is what even the most advanced ideas for new architectures call for, since there are no guarantees for absolute security. The integration with cell phone networks and services can solve majority of existing issues with mobility, and energy efficiency. Models for naming, multicast and anycast are also proposed, and as the need to mobility is growing, together with the need for better identification of data and services instead of machines, these models will slowly take off. This industry is in the beginning steps, and there are some great solutions on issues related to mobile support to smartphone devices, and we believe this field is moving in the right direction, thus rejecting the need for major changes in the current architecture.

3.3. OpenFlow Protocol

OpenFlow is a communications protocol that gives access to the forwarding plane of a network switch or router over the network. [8][65] OpenFlow enables network controllers to determine the path of network packets across a network of switches. The controllers are distinct from the switches. This separation of the control from the forwarding allows for more sophisticated traffic management than is feasible using access control lists (ACLs) and routing protocols. Also, OpenFlow allows switches from different vendors — often each with their own proprietary interfaces and scripting languages — to be managed remotely using a single, open protocol. The protocol's inventors consider OpenFlow an enabler of Software Defined Networking (SDN).

OpenFlow allows remote administration of a layer 3 switch's packet forwarding tables, by adding, modifying and removing packet matching rules and actions. This way, routing decisions can be made periodically or *ad hoc* by the controller and translated into rules and actions with a configurable lifespan, which are then deployed to a switch's flow table, leaving the actual forwarding of matched packets to the

switch at wire speed for the duration of those rules. Packets which are unmatched by the switch can be forwarded to the controller. The controller can then decide to modify existing flow table rules on one or more switches or to deploy new rules, to prevent a structural flow of traffic between switch and controller. It could even decide to forward the traffic itself, provided that it has told the switch to forward entire packets instead of just their header.

The OpenFlow protocol is layered on top of the Transmission Control Protocol (TCP), and prescribes the use of Transport Layer Security (TLS). Controllers should listen on TCP port 6653 for switches that want to set up a connection. Earlier versions of the OpenFlow protocol unofficially used port 6633. [9][10][11]

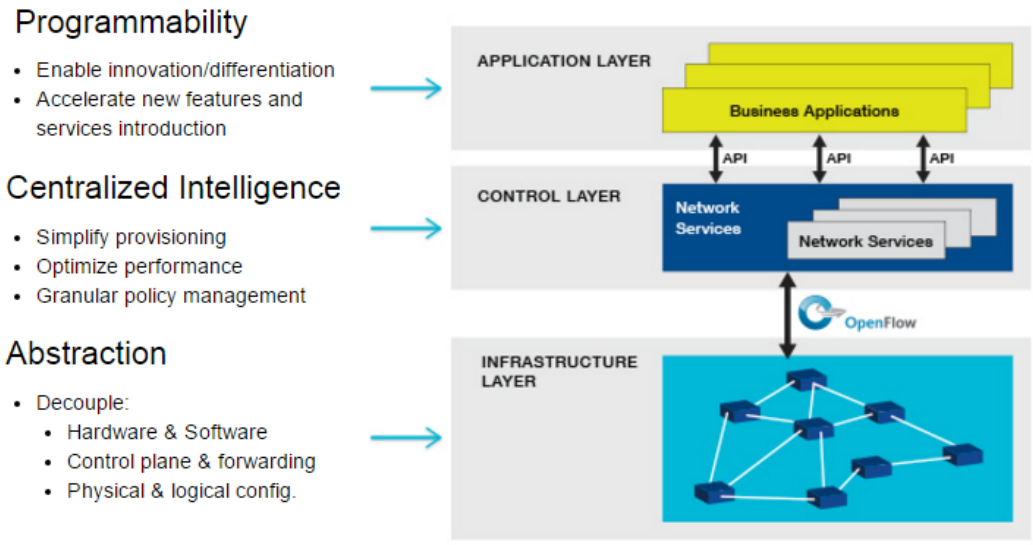


Figure 7. OpenFlow layered structure.

3.4. Deployments of OpenFlow

With all of the debate about OpenFlow and SDN progress, it's real-world deployments that will eventually tell the story. Many market participants have expressed frustration that commercial deployment is taking longer to develop than initially thought, but there are actual deployments taking place at both service providers and enterprises, so it's worth taking a look at these in more detail. Here are

some of the deployments that have been observed so far and what's driven them.

Google - Google, which manages one of the largest enterprise networks and cloud deployments in the world, is a big fan of OpenFlow. Engineers with direct experience with OpenFlow point out that it's a key element of the Google architecture. In a summary report on Google's infrastructure from 2014, Google technical expert Bikash Koley wrote that OpenFlow support is key to Google's SDN infrastructure, which is primarily built with white box switches and commodity hardware. "The only way to get well defined control and data plane APIs on routing hardware at that time was to build it ourselves," wrote Koley, telling the now well-known story of Google's homegrown network.

It's important to note that the scale at which Google is using OpenFlow is quite wide: It's using it both inside the data center and to interconnect data centers, which is a wide-area network application. The company also uses OpenFlow to model and implement traffic engineering of the network across data centers, a use case that would work in some of the world's largest service providers. [12][67]

Geant - Géant, Switzerland Technology firm Géant, whose network connects more than 10,000 research institutions such as the CERN physics lab in Switzerland, said it is deploying SDN using a combination of Infinera's packet-optical technology, Corsa Technology's programmable switching and routing platform, and the Open Network Operating System (ONOS) SDN controller, which includes OpenFlow as a southbound protocol. Géant says its network can use this SDN and OpenFlow technology to handle traffic at different layers, including an SDN Layer 3 domain that also routes traffic to the general Internet.
This is accomplished with an ONOS application known as SDN-IP, which provides a mechanism to share routing information between legacy and SDN domains. The company says its use of OpenFlow-based controllers is key to the programmability of the network as well as its capability to interact with the packet-optical network.

Other instances where OpenFlow is deployed involve a long list of companies, campus networks, etc., some of them being: Cornell University, REANZZ, New Zealand, TouIX, France, T-Mobile, Huawei, NEC, Oracle, Vodafone, Yahoo, ZTE,

Ricoh, Juniper, Hewlet-Packard, Intel, Microsoft, Brocade, etc. In total there are hundreds of them. [68]

CHAPTER IV. SDN MODELS FOR ENTERPRISE NETWORKS

4.1. Intruduction

The main reseach of this thesis is based on completed papers and presented in conferences and in order to answer the research questions and research hypotheses.

This chapter includes the list of papers and practical implementations of various models in order to address issues and problems about intra-domani routing and inter-domain routing, as well issues related to both cases, such as traffic engineering, transition issues from traditional to SDN networks, etc.

The papers are given in its original form, and are a result numerous networks simulations and emulations for both traditional and SDN networks implementations.

4.2. Separating network control from routers with Software Defined Networking

Data networks have become increasingly complex nowadays. Even though technologies like Ethernet, IP protocol and packet forwarding is rather simple, control mechanisms like middleboxes, Access Control Lists (ACLs), firewalls, traffic engineering, VLANs, etc. have largely contributed to increasing their complexity. Primarily this is due to the lack of basic principles in networking. Networking still remains vertically integrated, where hardware comes with its proprietary software and is not open to innovation.

Software-Defined Networking (SDN) instead decouples the data plane (which is and should remain the job of the physical routers) and control plane. The control plane in SDN is removed from the routers and switches, and instead is done in the edge of the network, thus allowing for third party software, open interface to devices regardless of hardware type and vendor, and easier management of networks. SDN is a new design model in networks rather than a new technology. It is a set of abstractions for the control plane rather than implementation mechanisms;

SDN in essence offers the possibility to network programmers and third party app writers build anything they want on top of both router chips (data plane) and the Network operating system (now through OpenFlow, but it may be something else as

well) in the control plane, as well as on top of the Network Operating system due to the open interface it introduces.

Software Defined Networking is not a revolutionary technology; it's an organizing principle in data networks. The rationale behind SDN is more important than its design. In 2008 the SDN elements like the Network Operating System (NOX) and OpenFlow switch interface were defined [69, 70, 71]. In 2011 the Open Networking Foundation was founded and now it has over 90 companies, among which Google, Cisco, Dell, IBM, Intel, Facebook, Verizon, Arista, Brocade, etc. [23]. Google publicly announced that they will use SDN for their interconnecting their data centers in 2012. SDN is now commercial and in production, although not so widely.

In order to explain SDN and the rationale behind it, it is important to draw a comparison between networks and software systems. A software system is a modular system based on abstractions to solve problems. A modular system allows for code reuse, change implementation and separate functions. To solve a problem we then should come up with abstractions, which in turn means to decompose the problem into its basic components, and then each component needs to have its own abstraction. These abstractions require an implementation to solve one specific task. Based on the complexity or the hardness of the task, it may again require going back the step one, until the implementations solve tasks that are easy to implement.

In data networking there are two planes: data plane, which processes packets with local forwarding state. The forwarding decision here is done based on the forwarding state compared to packet header.

The second plane in data networking is the control plane. It puts the forwarding state in the networking device, based on many possibilities and implementations. It can be computed using distributed or centralized algorithm, manually configured, etc, but regardless of this, it is a completely different function as opposed to the forwarding plane.

The abstractions that we have for the data plane are basically known to every network engineer or even computer scientist.

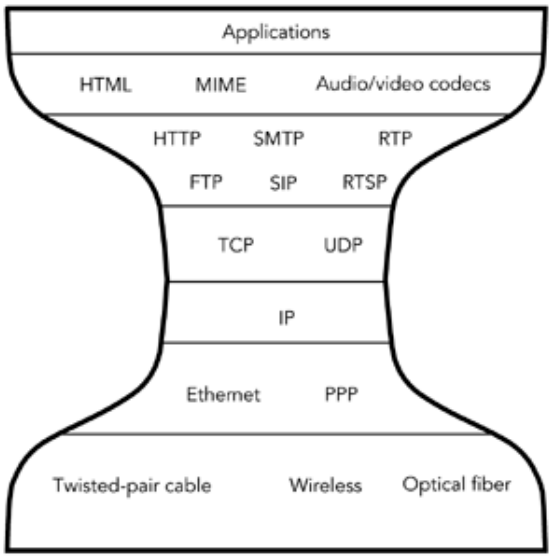


Figure 8. Abstractions and layering of data plane as a reason for Internet success

They are known as the protocol suites and the layering system used in data networking, namely TCP/IP. This layering model has been very successful in decomposing the problem in its basic components, and each implementation in this model solves a specific task. Applications are built on reliable end-to-end transport, built on best effort global packet delivery (network layer), which in turn is again build on best effort local packet delivery (link layer) on top of physical delivery of bits. Each layer is separate on top on the layer below [26].

The control plane on the other side doesn't have abstractions. We have a lot of mechanisms, which serve different goals. For example routing (a family of algorithms), isolations (VLANs, firewalls, traffic engineering, MPLS, etc.), but there is no modularity. And the functionality is limited. The network control plane is what happens when there are mechanisms without abstractions. So there is too many mechanisms without enough functionality. Each problem is solved individually and from scratch. This is not the way problems should be solved. Instead the problem should be first decomposed. That is the main reason that has lead to a great success and acceptance of Software Defined Networking [36, 72].

SDN imposes a big change in the industry just like computer industry was changed in the late 80es. At that time the computer industry was based on specialized hardware, specialized OS and specialized apps (usually all from one vendor, namely IBM). The computer industry was vertically integrated, close proprietary and relatively slow to

innovations. With the microprocessor, an open interface led to many operating systems and a huge number of apps on top of these Operating systems. Hence, this industry moved from closed and very difficult for innovations, vertically integrated and proprietary, to horizontal, fast innovation and open. Networking too has for a long time worked in a same way, based on specialized hardware, software and features. SDN in essence offers the possibility to network programmers and third party app writers build anything they want on top of both router chips (data plane) and the Network operating system (now through OpenFlow, but it may be something else as well) in the control plane, as well as on top of the Network Operating system due to the open interface it introduces (the control program).

The network control plane

The Control plane in a network should compute the forwarding state under three constraints [24]:

1. The forwarding state should be consistent with specific low level hardware/software,
2. It should be done based on entire network topology, and
3. It should be implemented in every router.

To take care of these constraints, network designers should define specific abstraction for each problem component.

1. The compatibility with specific low level hardware/software needs an abstraction for a general **forwarding model** that hides the details of specific hardware/software;
2. Being able to make **decisions based on entire network** takes another abstraction for the network state, hiding the mechanisms to get it; and
3. Another abstraction that deals with the **actual configuration of each network device**, so they are configured in a much easier and straightforward way.

The Forwarding Model in SDN

For the forwarding abstraction, we want to hide details of the type of hardware or software the decision in used at. The device itself may by manufactured by any device manufacturer and still the model should work in a same way. **OpenFlow** is the current proposal for that [6, 29]. It is a standard interface to a switch so we can access the

switch and we can store there flow entries through this protocol. It is a general language which should be understood by any switch. Conceptually this is a pretty easy and straightforward concept, although its practical implementation of design details (like header matching, allowed actions, etc.), may not be so [30]. The forwarding abstraction (implemented through the OpenFlow protocol) exploits the flow table in the routers and populates them with simple rules (if header x, forward to port y, etc.). It does a:

match + action,

in a similar way it is done today in networks. The set of actions in OpenFlow is rather small (forward packet to specific or set of ports, drop the packet, or send it to the control plane, and also define bit rate at which packet will be forwarded. The most interesting thing to look at with OpenFlow is rather the *action* part of the function, because the desired goal here it to use a minimal set of actions to be a good enough set of actions to do most work on one hand, and offer the possibility to chip vendors to implement it and program writers to offer special features that makes them unique in the market, on the other hand. Eventually the OpenFlow should offer protocol independence to build different types of networks, like Ethernet, VLANs, MPLS, etc, and new forwarding methods which are also backward compatible and technology independent [31, 32, 33].

The Network State in SDN

In routing, we want to abstract the way how the distributed algorithms get the global network state, and instead only give a global network view, annotating things and information relevant to the network administrator, like delay, capacity, error rate, etc, so the network admin is able to program the switches the way they desire. It is implemented through the Network Operating System in SDN, which runs on servers on the network, and the information flows in both ways, which means the servers get the information about the network state by querying the switches to create the global network view. Based on what policy we want to implement in the switches, we do it in the opposite direction (router configuration). The Network Operating System gets the information from the network routers to create a global network view, and then a

control program implemented on top of the Network Operating System implements policies about routing, access control, traffic engineering, etc. This is a major change in the networking paradigm, where the control program implements the policies into the routers [24].

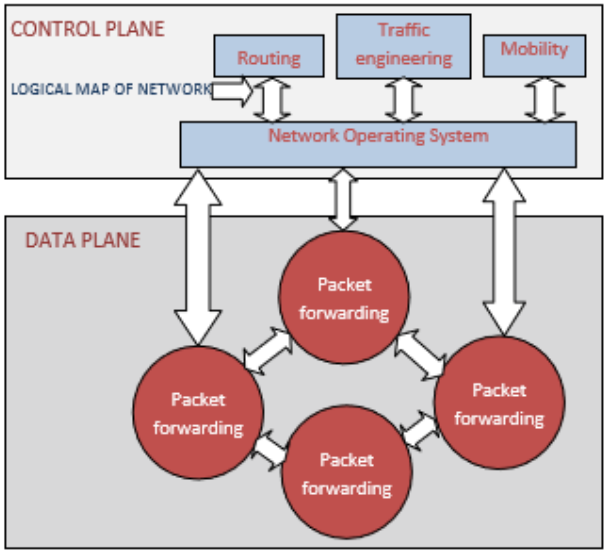


Figure 9. The Control Plane running on the edge, separately from the Data Plane in a SDN

Router Configuration in SDN

The control program needs to install the policies and flow entries intro each router in the network. But the control program needs to just express desired behavior, and not be responsible for specific statements. Another abstraction deals with writes of specific statements in routers. Rather than the control program dealing with the actual full network topology, it only deals with a virtual layer created on top of the full global network topology. This way each part of the problem is decomposed in a clean way, and each abstraction deals with its own task. That is, the control program expresses the desire for some specific network configuration on one or more routers, and the specification virtualization layer does the actual mechanics of implementing it on the actual routers in the network.

The SDN's achievement is not to eliminate complexity because the layers and the network operating system are still very complex and complicated. Its achievement

however is to simplify the interface to the control program so that it has a simple job to specify what we want to do with the network. The hard part is the reusable code, and once it is done right, it will be used by any network programmer without the concern of knowing the details of its implementation. That part is implemented in the operating system and the control program, which is the reusable code. The comparison here is with programming languages and compilers. The programmer needs not to know how the compiler is implemented, not should be familiar with the instruction set, because in programming those two problems have been decoupled a long time ago. This is something that has not yet been done in networks, and that is the major goal SDN accomplishes in the data networking field [35, 36, 73, 74].

Applications of SDN

In today's networks we can easily says that topology is policy, meaning the actual physical location of routers, firewalls, etc. dictates how effective the network is, how well the ACLs work, what our broadcast domains are, etc. ***When networks are moved to the cloud, we usually want to keep same policies***, but very few networks operators have an abstract expression of network policy, rather they have a network topology. SDN allows specifying a logical topology to the cloud. The cloud then ignores the physical topology and follows the logical topology based on the policy read based on the topology initially.

The function is evaluated on an abstract network and only the compiler needs to know the actual physical network topology. The major changes that SDN brings to the networking world in general is not the easier network management (which comes as a result of it), but rather primarily decouples the data plane from the control plane. They are now the same in terms of vendor, and place where they are implemented. This changes the business model in networking (hardware bought separately from software, which can be third party). But it also brings a clean interface which allows for much easier implementation of testing of networks in the split architecture.

Simplified Network Troubleshooting

SDN allows for implementation of control function in the edge of the network, instead of in the routers, where it actually is, and the core only deals with delivering packets

end-to-end. The core may easily remain legacy hardware and the network operators need not to know at all the one is implementing SDN from the own edge of the network. So it simplifies network troubleshooting and also there are no disruption periods during network convergence, due to the fact the one policy is implemented per specific flow, and each packet will be carried either by an old state, or by a new state. In traditional routing packets may well be lost during network convergence, or loops may appear. In SDN the expressions are high level and easily checked and corrected.

Network customization

The SDN gives an extra benefit to the network operators to more easily customize their networks based on their needs. The policies, traffic engineering, monitoring and security is easier to implement after getting a network state quickly from the Network Operating Systems, and moving the network to a virtual environment without much effort is perhaps the greatest benefit in this specific scenario. An example would be to improve load balancing in a network with many servers in different locations connected though a backbone. The load balancer used in networks today chooses a lightly loaded server to do the job. But since servers are in different location, the load balancer does not take into account to choose the lightly loaded path too. Ideally we would like to choose both, for a best result. SDN not only gives the possibility to do this, but more importantly, because the control plane now resides on the edge, it can be written by anyone and is vendor independent, it can be done in a very short period of time and tested in real time.

Third Party Apps in Networking

Also, network operators may hire third party people to develop special features for their networks, as well as remove unneeded features from routers. Removing some unused features from routers arguably increases their reliability of routers.
SDN offers a chance to increase the rate of innovation by moving the operation in software; standards will follow the software deployment instead of other way around. Another great opportunity here is experience, technology and innovation between vendors, universities and researchers.

As of now, there are already different domains where SDN has been or is under implementation, including data centers (Google for example), public clouds, university campus networks, cell phone backhauls [31, 32, 37], but also in enterprise Wi-Fi environments and home networks . There are already over 15 vendors offering SDN products, and probably the number will grow, including new jobs in this field.
SDN is an opportunity to program the network infrastructure in an easier way, by offering network wide visibility as well as direct control through OpenFlow.
Another significant advantage in SDN is that OpenFlow offers the ability to innovate on top of the low level interface that today's controllers provide, by increasing the level of abstraction. Today's controllers do not have a complete network wide view primarily for scaling purposes, whereas in SDN the control program has a complete network wide view, and it can actually use the virtualized information they need for the network state. It is hard to compose different tasks in today's networks (like monitoring, access control and routing).

Programming in SDN

SDN and OpenFlow have made possible to program the network. In a SDN there is a logically centralized controller and an arbitrary number of switches under its control. The controller by default is a smart and slow device, as opposed to the fast and dumb switches in the network, which only manage packets (based on the policy coming from the controller). The controller pushes the policies to switches through the OpenFlow API.
There are three aspects which dictate the way how SDN are programmed: 1. The data plane abstraction is very simple and the architecture is centralized with direct control over it. 2. The programming interface of OpenFlow API is relatively low-level with a number of limitations. The functionality when programming through OpenFlow is limited and tied to hardware, and the programmer needs to manage the resources explicitly, which in routers are scarce (similar to doing register allocation in Assembly language coding). 3. Probably the most difficult thing with OpenFlow programming is when combining different modules at the same time (like routing, monitoring, load balancing, etc.). The programmer would be able to do this much easily in a high programming language.
A new programming language, called Frenetic has been developed, allowing for

network programming at a higher level of abstraction. Frenetic is a SQL-like query language, which allows composition of different modules possibly at the same time [36]. The following example illustrates how traffic statistics ban be collected:

```
Select(bytes)*
Where(in:1 & scrport:25)*
GroupBy([dstip])*
Every(30)
```

Figure 10. Count number of bytes with TCP port 25 coming in port 1, grouped by destination IP address every 30 seconds.

```
#Repeating between two ports of a repeater
def repeater():
rules=[Rule(in:2, [out:1]),
        Rule(in:1, [out:2])]
register(rules)
```

Figure 11. A repeater forwarding traffic from one port to another

```
#Monitoring web traffic
def traffic_monitor():
q = (Select (bytes)*
Where (in:1 & srcport:25) *
Every (45))
q >> print
```

Figure 12. A traffic monitor colleting incoming traffic data from port 1 with TCP port 25 every 45 seconds

```
#The previous modules composed into one in Frenetic
def main():
repeater()
traffic_monitor()
```

Figure 13. Composition of two modules in Frenetic

Figure 13 illustrates how composition of modules can be done in Frenetic, something

which is not possible directly through OpenFlow interface.

Future Work

SDN is a new set of abstraction with many unanswered questions related to practical implementations. The issues of mobility, security and privacy will have to be addressed in the future, as new control programs emerge on top of existing physical infrastructures. The third party apps developed by different network programmers will have to be fully validated and checked for security holes before they are implemented in actual networks. But SDN also allows for rapid prototyping at software speeds, not having to wait for vendors to come up with new features in networks. This is where we will focus our research work in the future. We will try to implement new routing policies in existing network environments using SDN and Frenetic in a real network, thus adding extra features to our network that the actual routers do not have. We will also focus on mobility, as most network devices nowadays are mobile.

From the abovementioned, we can conclude that networking over the years has been vertically integrated, where hardware comes with its proprietary software and is not open to innovation.
Software-Defined Networks (SDN) instead decouple the data plane (which is and should remain the job of the physical routers) and control plane. The control plane in SDN is removed from the routers and switches, and instead is done in the edge of the network, thus allowing for third party software, open interface to devices regardless of hardware type and vendor, and easier management of networks. SDN is a new design model in networks rather than a new technology. It is a set of abstractions for the control plane rather than implementation mechanisms.

SDN is merely a set of abstractions for the control plane. It is not a set of mechanisms. It involves a computing function and the Network Operating System deals with the distribution of state.

SDN in essence offers the possibility to network programmers and third party app writers build anything they want on top of both router chips (data plane) and the Network operating system (now through OpenFlow, but it may be something else as

well) in the control plane, as well as on top of the Network Operating system due to the open interface it introduces.

4.3. Emulating enterprise network environments for fast transition to Software-Defined Networking

Software-Defined Networking has introduced a new reality in data networking world, not present before. The concepts it introduces are of the same importance for networking as are high level programming languages and compilers for the general purpose computer systems – they basically decouple the hardware from the software in a network switch, and thus it makes the hardware platforms software independent. This is leading towards many benefits, among which better network control and management, a possibility to build logical network overlays and thus manage all the network switches (from a centralized controller rather than distributed) following the end-to-end path, and not dealing with individual switches at a time. It has also opened a way to the research community and third party vendors to offer enterprise specific solutions, based on consumer needs, thus bringing the prices of switches down.

Arguably, for the research community the biggest benefit might be the possibility to rapidly prototype [75] simulations/emulations of SDN network environments using real world parameters (data rates, size and number of nodes, means for analysis and troubleshooting) before they are applied on testbeds or real networks, and compare its advantages to conventional non-SDN networks.

Also, open-source platforms like POX [76] and NOX [77] offer the possibility to write control applications for Software-Defined Networking controllers, such as the OpenFlow controllers. OpenFlow has gained a huge deal of use in Virtual Machines, but also in mission critical networks. Google's internal network now runs completely on OpenFlow.

In this section we prototype an enterprise/university Software-Defined Network in a virtual environment. For this purpose, we use Python and Mininet [78] CLI to write the code for network creation and configuration (note: that there are at least two other ways of doing it, the first is by doing it directly in Linux, and the second way can be done using only Python programming language) making it easier to write and create

topologies (including virtual switches, network controllers that pushes the rules and policies into the virtual switches) due to a higher level of abstraction. This way we enable implementation of routing policies within a network in dumb switches (dumb switches being those network devices implementing only a forwarding plane, but not a control plane) through pushing the rules from the virtual controller.

The remaining section is organized as follows: first we present the network model created in Python and run in Mininet, then we discuss the results and findings from the emulated network in terms of data rates, delays, scaling, and advantages compared to conventional non-SDN networks. Lastly we present work conclusions and discuss future work.

Proposed model description

Our network model consists of 50 network switches that are used for packet forwarding, a central controller, and a host connected to each switch. The switches are connected linearly (linear is the Mininet term for bus topology) among themselves for simplicity, but they can be rearranged based on one's needs and requirements. Every switch is connected to the central controller, which pushes the routing policies into them, but this also is not a mandatory requirement, since rules can be pushed towards nodes even through intermediary nodes in the same way (following a tree structure, mesh, partial mesh, star, etc.).

We have also implemented a virtual conventional network of same size (using ns-3) to depict some disadvantages of distributed [79, 80] control implemented in conventional networks and also for analysis and comparison purposes.

We run the code in Mininet, which is a network emulator used to create virtual Softare-Defined networks with virtual switches, controllers, hosts, etc. The virtual switches of Mininet, needless to mention, support the OpenFlow protocol, as well as a Python API to create, manage and experiment with networks.

The version we use is Mininet 2.0 with Ubuntu 12.10 in VirtualBox v.4.3.8. The whole virtual platform runs on a Windows 8 HP laptop, with an i3 2310 processor, 4GB DDR3 RAM and a 1Gbps Realtek PCIe Ethernet adapter for Internet connection.

We have used Wireshark to analyze propagation of packets, ICMP and OpenFlow (OF) packets, as well as HTTP requests sent from one selected node in the network to another. Then the same procedure is repeated with the virtual conventional network and the results are compared against each other. As far as the routing tables and policy implementations are concerned, here the comparison was done using a different technique due to the difficulty to measure the time for total network converge in a conventional network. Namely, it was pretty straightforward to measure the convergence time (the time to propagate the rule from the controller to the switch plus the time for the rule to install in the routing table of the switch) in our virtual Software-Defined Network. Whereas it is more subtle to measure convergence time in a conventional network (even more so in a virtual one), due to the fact that the network control is distributed and we only know that the convergence is reached by sending a series of packets from one to another host and tracking the amount of time it takes for packets to start following a different path (or in our case reporting an unreachable host due to our bus topology).

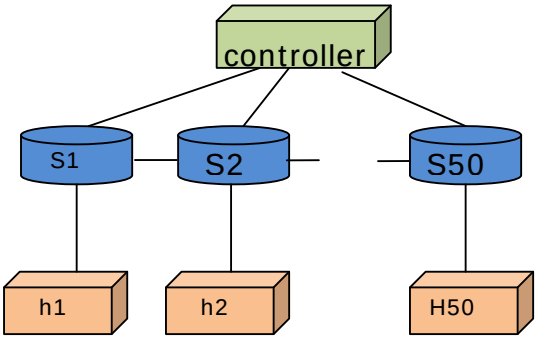


Figure 14. The virtual SDN network topology

At last, it should be also noted that the topology that we have used slightly favors SDN vs. conventional networks in terms of delay as far as control packets are concerned due to the direct link of each switch to the controller, but in terms of data packets parameters both networks are totally equal.

Figure 14 depicts the created virtual SDN network in Python. An identical topology with the same number of nodes has been created in ns-3 network simulator, to analyze and compare the results between the proposed Software-Defined Network and the

conventional network. Based on the results, we propose the abovementioned network model.

Results

The test consists of measuring min, max and means propagation delay of ICMP and HTTP packets from one random host to another using Mininet during 600 seconds. The same test is done by picking the same hosts in the ns-3 environment, and for the same amount of time. We have also changed the routing policy in the Mininet controller using the Python API to push a new policy into the OVS-switches and measure the convergence time in Wireshark. Whereas in the ns-3 environment, we kept sending ICMP packets from one host to another until the host became unreachable and that is our reported time to achieve convergence in the conventional network.

After the tests have been run on both Mininet and ns-3 we have reported results for control packet and data packet propagation delays from the controller to the OVS-switches. Compared to the conventional virtual network data packet delays, we see a slight advantage of the Mininet platform, as shown in Figure 3 and 4. In a real network scenario, this phenomenon is interesting to be analyzed – if the same results persist, then we could conclude that the OpenFlow switches can process data packets slightly faster due to the fact that they do not spend CPU time in calculating optimal paths and installing new routes in the routing tables, but they only spend CPU time to install the already calculated path from the central controller. The only disadvantage for the SDN, as Figure 15 depicts is for the maximum packet delay, but this is only for the first packet that goes through the switch until the switch is trained, and therefore it can be considered as negligible.

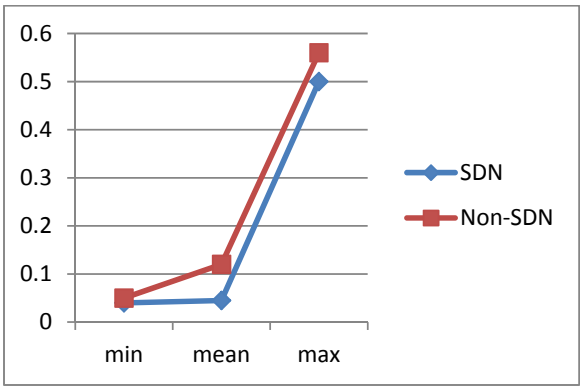


Figure 15. Propagation time in milliseconds for ICMP packets in the SDN and conventional network

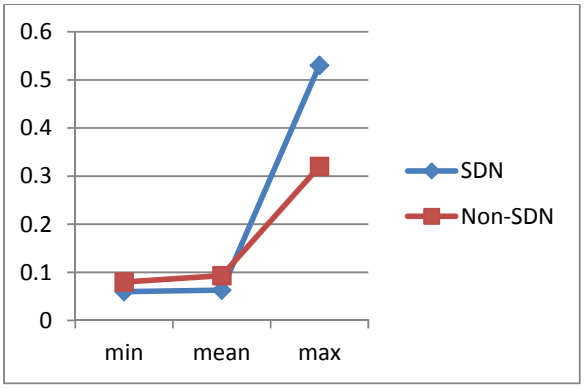


Figure 16. Propagation time in milliseconds for HTTP packets in the SDN and conventional network

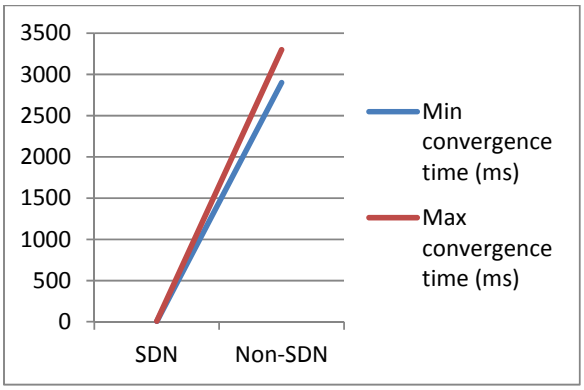


Figure 17. Propagation time in milliseconds for the SDN and conventional network until network convergence is reached

Nevertheless, the real difference lies in the fact the all SDN switches are updated with one push command using the OpenFlow interface from the central controller to each of the hosts. This is a real advantage in achieving a full network convergence in much shorter time, than we see this convergence in a conventional network. As it can be seen in Figure 17, the time difference between both scenarios is tremendously in favor of our Software-Defined Network, with only about 7ms convergence time versus ~3200ms for the conventional network. Given the fact that enterprise and datacenters may employ more than 100 switches (which we used in our scenario), this difference may become even bigger.

Software Defined Networks are emerging as a new paradigm in data networks. While most corporate networks and datacenters (like Google, Cisco, Big Switch, etc.) are already on OpenFlow protocol utilizing SDN, a number of emulation platforms have emerged, such as Mininet. In this section we emulate an enterprise/university Software-Defined network using Python and Mininet, as well as a conventional network using ns-3 with the same number of nodes, topology, etc. Our tests show that Software-Defined Networks outperform conventional networks in parameters like minimum and mean delay for ICMP and HTTP packets, as well as minimum and maximum convergence time, and therefore are a best solution to transition corporate networks with hundreds or more nodes, with good scaling capabilities [81, 82, 83] mainly due to the fast propagation of routing policies from the central controller.

As we have seen in our models, we can conclude that Software-Defined Networks can perform faster in enterprise/university networks, as far as both data packets and control packets are concerned. This, having in mind the fact that we have used two different platforms for the test, namely the well known ns-3 simulator for the conventional networks, and Mininet to emulate the virtual OpenFlow network. This is due to the fact that Software-Defined Networks have specific hardware and software requirements and practically it is impossible to carry out tests and compare SDN and non-SDN networks in a same virtual environment. However, both test have been carried out on the same platform, with the same hardware configuration. Also, the tests have been carried out in a specific node arrangement, which represents one specific topology, and similar tests can be carried out for different types of topologies. Based

on the tests for ping, HTTP, and network convergence, our conclusion is that our Software-Defined Networking model performs faster in all cases, except for the maximum delay of ICMP and HTTP packets. However, this is only valid for the very first packet that goes through the OVS switch, until the switch is trained.

Based on our conclusions on one hand, and the positive trend in networks towards SDN [84], we can anticipate that SDN will be around in the future and that as new control and data applications become available, SDN will start to become part of more and more corporate, enterprise and university campuses. Accordingly, in the following section we plan our future work.

Our future work will rely on the results we have gained by the previous implementations of emulated SDN, as well as current research work performed by us and other researchers from the field. We are strongly confident that SDN as an organizing principle of data networks will have a future in major networks with hundreds or thousands (or even more) nodes, therefore based on this, our research will continue in two similar directions: 1. We will continue to run more exhaustive and different tests on various scenarios, topologies and data traffic (especially video and voice), until we have analyzed thoroughly all aspects and differences, as well as possible problems that might come up in a virtual SDN environment, and 2. We will propose to redesign our university campuses' network, which is similar in number of nodes to the one presented in this work and transition to SDN.

4.4. Improving network performance of emulated data centers with unpredictable traffic patterns using Software Defined Networking

SDN was created in response to demands from large data centers, which faced problems coping with very unpredictable traffic patterns. These patterns would cause high demands for particular resources that could not be met with existing network infrastructures. Practically, two solutions were viable in such scenarios: to either scale the infrastructure to meet the peaks, which is expensive and the resources would stay underutilized most of the time, or build the network so it can reconfigure itself to meet the appropriate demands.

SDN is a programmatic method where the costumer can alter the network to meet the peaks and demands when rapid changes happen, so that social networking sites and large data centers [83] that are dispersed in different locations are able to meet the specific demands.

It offers many benefits, among which better network control and management, a possibility to build logical network overlays and thus manage all the network switches (from a centralized controller rather than distributed) following the end-to-end path, and not dealing with individual switches at a time. It has also opened a way to the research community and third party vendors to offer enterprise specific solutions, based on consumer needs, thus bringing the prices of switches down.

For the research community the biggest benefit might be the possibility to rapidly prototype simulations/emulations of SDN network environments using real world parameters (data rates, size and number of nodes, means for analysis and troubleshooting) before they are applied on test beds or real networks, and compare its advantages to conventional non-SDN networks. Also, open-source platforms like POX and NOX offer the possibility to write control applications for Software-Defined Networking controllers, such as the OpenFlow controllers. OpenFlow has gained a huge deal of use in Virtual Machines, but also in mission critical networks. Google’s internal network now runs completely on OpenFlow (B4, Google's SDN-powered WAN, provides connectivity among data centers. Traffic includes asynchronous data copies, index pushes for interactive serving systems, and end user data replication for availability. Well over 90% of internal application traffic runs across this network).

The following section of this section outlines the issues related to implementing an SDN, as well as explains the implementation process of an emulated data center, including a real SDN controller in the network to measure network performance.

When implementing Software Defined Networking, from the programming aspect, the implementation process involves three major steps. First, a controller needs to read or monitor network state and various events occurring in the network (topology changes, failures, security events, etc.). The second step is to compute the policy based on the state that the controller sees from the network, i.e. the role of the decision plane in deciding what the forwarding behaviour of the network should be in response to various states that it reads from the network switches. The third step is to write the network policy back to the switches by installing appropriate flow table state into the switches.

Consistency problems can arise in two steps: first, the controller may read state from the network switches at different times, resulting in an inconsistent view of the network-wide state, and second, the controller may be writing policy as traffic is actively flowing through the network, which can disrupt packet along the end to end path, or packet that should be treated consistently, because they are part of the same flow.

Both reading and writing network state can be challenging, because OpenFlow rules are simple match-action predicates, and it is very difficult to express complex logic with these rules.

Another problem when implementing an SDN is the fact that it is intended for larger data centers and campuses, which is not always very practical and feasible to use and test. Therefore, more and more applications are emerging, which enable to connect and run real SDN controllers to emulated platforms, thus offering similar characteristics to real networks by shortening the implementation time drastically. Mininet [85] is such a platform. It enables us to create emulated network topologies on a single PC for experimentation using a virtual machine running on a PC, while running real switch software that can run real application code through command line interfaces and Python to interact with it; Mininet can actually run any program that can run on Linux, therefore we can create a virtual network and connect an OpenFlow controller to real OpenFlow switches that are running in the virtual network

environment.

For the purpose of the implementation, we have used the following equipment: a Lenovo Yoga 2 Pro laptop, with i7 processor, 8GB DDR3 RAM and 512GB SSD running Windows 8.1, Mininet 2.2.0 on Ubuntu 14.04 32 bit (recommended for Windows) on Oracle VM VirtualBox, connected to a HP 2920 PoE Switch, Open vSwitch, and HP SDN controller.

In order to perform the connection a number of steps need to be taken, not necessarily in the order given here.

First we open and perform the necessary updates in the Virtual Machine with the controller, which is part of the VM. The authors have had problems accessing the controller using Ubuntu 12.04.03, so it is recommended to use Ubuntu 12.04. Also sometimes there are chances that when trying to access the controller in the VM from the browser, using the local IP address, it may not open. In such cases it is needed to perform the java update on the VM (Fig.18).

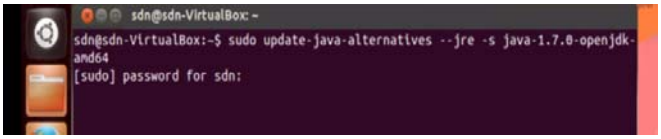


Figure 18. Performing updates on the virtual machine

Next we have moved to performing the switch configuration, connection is to the USB port of the laptop. It is also recommended that the switch is updated. To start the switch configuration we need two vlan-s, one to communicate with the Internet, and the other one for configuration. The following figure shows the vlan configurations done in the switch. After that, we run OpenFlow to further configure the switch, so that the controller talks to the configuration vlan.

Figure 19. Switch configuration

In Mininet we then create a topology with our controller as part of it. Later we access and manage the Open vSwitch. The easiest way to do it is to just install and use Mininet, since it incorporates Open vSwitch installed (the command to check whether it is installed is sudo ovs-vsctl show), and make take the Open vSwitch to the controller, possibly through a bridge, again from Mininet. Once all address configurations are done, the controller should be able to talk to the Open vSwitch and the topology created in Mininet.

From here on, we can move onto the creation of the topology, based on our needs. As it was previously explained, SDN is mostly suited for larger networks, with hundreds or thousands of nodes, and therefore the topology we need to use in order to achieve useful test results should be more complex and non-trivial.

Our sample SDN consists of 20 Open vSwitches that are used for packet forwarding, the SDN controller, all managed from the virtual environment from Mininet. The switches are connected linearly (linear is the Mininet term for bus topology) among themselves for simplicity, but they can be rearranged based on one's needs and requirements. Every switch is connected to the SDN controller, which pushes the routing policies into them, but this also is not a mandatory requirement, since rules can be pushed towards nodes even through intermediary nodes in the same way (following a tree structure, mesh, partial mesh, star, etc.). We used Wireshark to analyze propagation of OpenFlow (OF) packets, as well as HTTP requests sent from one select node in the network to another. Our test consisted of checking the system stability over a period of time of 12 hours of network operation, with constant file transfers from one virtual machine to another.

As depicted in Figure 20, the system proved to be stable, while receiving round trip times (RTT) of less than 20ms in over 90% of time.

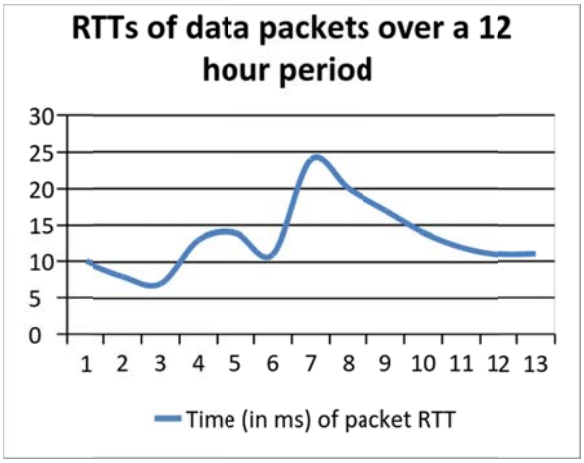


Figure 20. Results of network stability over a 12 hours test time in implemented SDN.

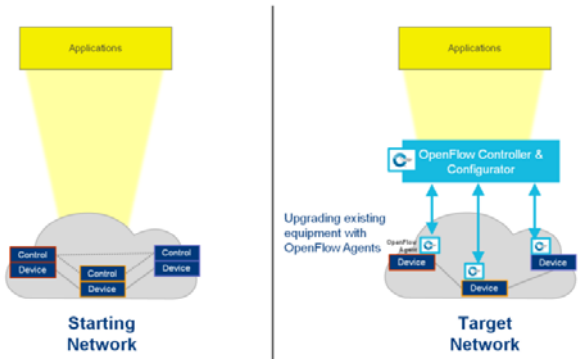
In this test, we have implemented a sdn network using an HP SDN controller, open v-switch and Mininet to monitor network stability for a larger campus network. the results show that for unpredictable/random traffic, sdn offers great stability, with relatively short RTTs in over 90% of time tested.

Based on our conclusions on one hand, and the positive trend in networks

towards SDN, we can anticipate that SDN will be around in the future and that as new control and data applications become available, SDN will start to become part of more and more corporate, enterprise and university campuses.

Our future work will rely on the results we have gained by the previous implementations of emulated SDN, as well as current research work performed by us and other researchers from the field. We are strongly confident that SDN as an organizing principle of data networks will have a future in major networks with hundreds or thousands (or even more) nodes, therefore based on this, our research will continue by running more exhaustive and different tests on various scenarios, topologies and data traffic, until we have analysed thoroughly all aspects of our implementations.

As a reminder, Google's global user based services [86] (Google Web Search, Google+, Gmail, YouTube, Google Maps, etc.) require significant amount of data to be moved from one region to another, making these applications and services very WAN-intensive. Google concluded that the delivery of such services would not be scalable with the current technologies due to their non-linear complexity in management and configuration. As a result, Google has decided to use SDN for managing WAN as a fabric as opposed to a collection of boxes. Hence, the future of SDN is already here, and the transition for most data center and large networks has begun. Figure 21 depicts the transition (starting and target network outlooks) as described by the Open Networking Foundation.



Picture 21. Starting network outlook and target network outlook of major networks of data centers and campuses. The picture is courtesy of ONF [86]

4.5. Summary

This chapter consists of a collection of research papers and publications in the area of Software-Defined Networking. Each of the papers is given in its almost original form as they are submitted. Each one deals with practical implementation of SDN to model or simulate/emulate various networks inquiries, policy and management issues and are compared to traditional networks. While this represents original work, they treat issues like separation of network control from routers with OpenFlow, emulate enterprise network environments for fast transition to SDN, or deal with network performance issues of emulated data centers with unpredictable data patterns in SDN in order to improve performance. Lastly, it tackles the issue of improving traffic control by implementing and analysing parameters for inter-domain routing, as oposed to previous implementations that deal with intra-domain routing and traffic management.

The following chapter gives the final conlusions of the thesis, as well as discusses future work.

CHAPTER V. SDN MODELS FOR INTER-DOMAIN ROUTING

5.1. Introduction

This chapter includes practical implementations of various models in order to address issues and problems about inter-domain routing, as well issues related to traffic engineering, transition issues from traditional to SDN networks, etc.

Numerous networks simulations and emulations tests for both traditional and SDN network models implementations were carried out.

5.2. Improving traffic control using Software Defined Networking for inter-domain routing

SDN was created in response to demands from large data centers, which faced problems coping with very unpredictable traffic patterns. These patterns would cause high demands for particular resources that could not be met with existing network infrastructures. Practically, two solutions were viable in such scenarios: to either scale the infrastructure to meet the peaks, which is expensive and the resources would stay underutilized most of the time, or build the network so it can reconfigure itself to meet the appropriate demands.

SDN is a programmatic method where the costumer can alter the network to meet the peaks and demands when rapid changes happen, so that social networking sites and large data centers, that are dispersed in different locations are able to meet the specific demands.

It offers many benefits, among which better network control and management, a possibility to build logical network overlays and thus manage all the network switches (from a centralized controller rather than distributed following the end-to-end path, and not dealing with individual switches at a time. This allows for easier traffic engineering, management and security. Techniques for easier prevention of Denial of Service attacks can be proposed in SDN, as it is explained in the section below.

SDN has also opened a way to the research community and third party vendors to offer enterprise specific solutions, based on consumer needs, thus bringing the

prices of switches down.

For the research community the biggest benefit might be the possibility to rapidly prototype simulations/emulations of SDN network environments using real world parameters (data rates, size and number of nodes, means for analysis and troubleshooting) before they are applied on test beds or real networks, and compare its advantages to conventional non-SDN networks. Also, open-source platforms offer the possibility to write control applications for Software-Defined Networking controllers, such as the OpenFlow controllers. OpenFlow has gained a huge deal of use in Virtual Machines, but also in mission critical networks. Google's internal network now runs completely on OpenFlow (B4, Google's SDN-powered WAN, provides connectivity among data centers). Traffic includes asynchronous data copies, index pushes for interactive serving systems, and end user data replication for availability. Well over 90% of internal application traffic runs across this network.

As a reminder, Google's global user based services require significant amount of data to be moved from one region to another, making these applications and services very WAN-intensive. Google concluded that the delivery of such services would not be scalable with the current technologies due to their non-linear complexity in management and configuration. As a result, Google has decided to use SDN for managing WAN as a fabric as opposed to a collection of boxes. Hence, the future of SDN is already here, and the transition for most data center and large networks has begun.

The following section of this section outlines the practical applications related to implementing an SDN in inter-domain routing and advantages, compared to conventional inter-domain routing.

SDN changes the way we manage networks, datacenters, backbones, enterprises, etc. Most of these are applications of SDN are used in internal networks, enabling software technologies such as network virtualization, traffic engineering, etc. It is interesting to view how SDN relates and how it fundamentally changes how traffic is delivered in between independently operated networks. BGP (border gateway protocol) lacks sufficient flexibility for performing various traffic engineering and traffic management operations. It's generally possibly to route only on destination IP address blocks, and it is only possible to exert indirect control over how switches and

routers forward traffic, through local preference, etc. But generally it is difficult to introduce new network services, such as ability to arbitrarily route traffic flows through middleboxes.

There are a variety of wide-area services on the other hand, and it would be extremely valuable to network operators had there been a way to implement them. One possibility is application specific peering, or providing the ability to autonomous systems to exchange traffic only for specific applications, such as video streaming. Another application that could be significantly enhanced is the ability to block denial-of-service traffic, such as the ability to block traffic in upstream autonomous systems, much closer to the source. Other applications include direct client request to different data centers, depending on where they are originating, steering through network functions, and various inbound traffic engineering options, such as splitting incoming traffic over multiple incoming peering links.

SDN allows routers and switches to match packets on multiple header fields rather than just their destination IP address. It also allows a network controller to control entire networks with a single program, possibly even including remote autonomous systems, as opposed to just immediate neighbors, and provides mechanism for direct control over packet handling rather than indirect control, via various routing protocols. It also allows and gives the ability to perform many different actions on packets, beyond simply just forwarding them.

Given the previously given and known SDN capabilities, SDN deployment in inter-domain routing can be applied at Internet exchanges (IXP). An SDN deployment even at a single IXP can benefit tens of hundreds of providers, and this is possible without requiring any of the providers to deploy any new equipment. ISPs have also tremendous interest in innovation. Practically, the number of IXPs is growing globally, with some 400 IXPs worldwide nowadays.

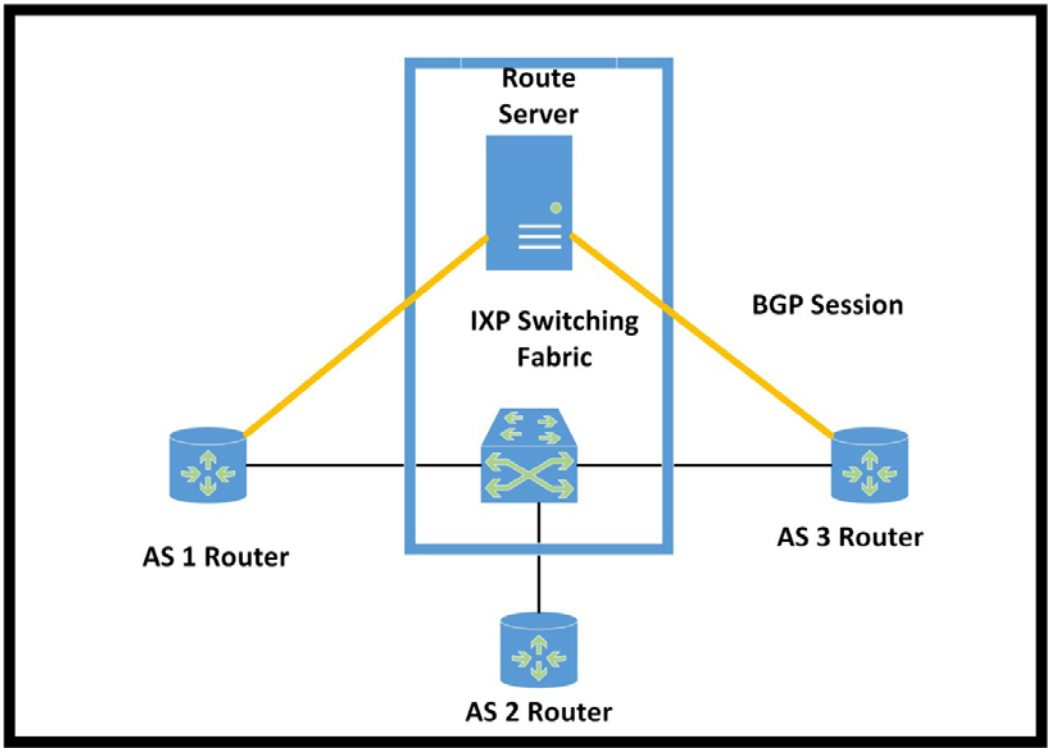


Figure 22. Conventional IXP layout

In conventional IXPs, participant autonomous systems exchange BGP routes with a route server. The ASes then make their own forwarding decisions and forward traffic to each other over a common layer 2 switching fabric. The primary function of a route server in an IXP is to avoid having each of these autonomous system make pairwise connections with one another. Instead, all of them can communicate directly with the route server. So, the idea of Software Defined Inter-domain exchange point (SDX) is to turn a route server into a smart SDN controller, and make the layer L2 switch an SDN capable switch that can take commands from the SDN controller concerning the forwarding table entries that one can have installed for each participant.

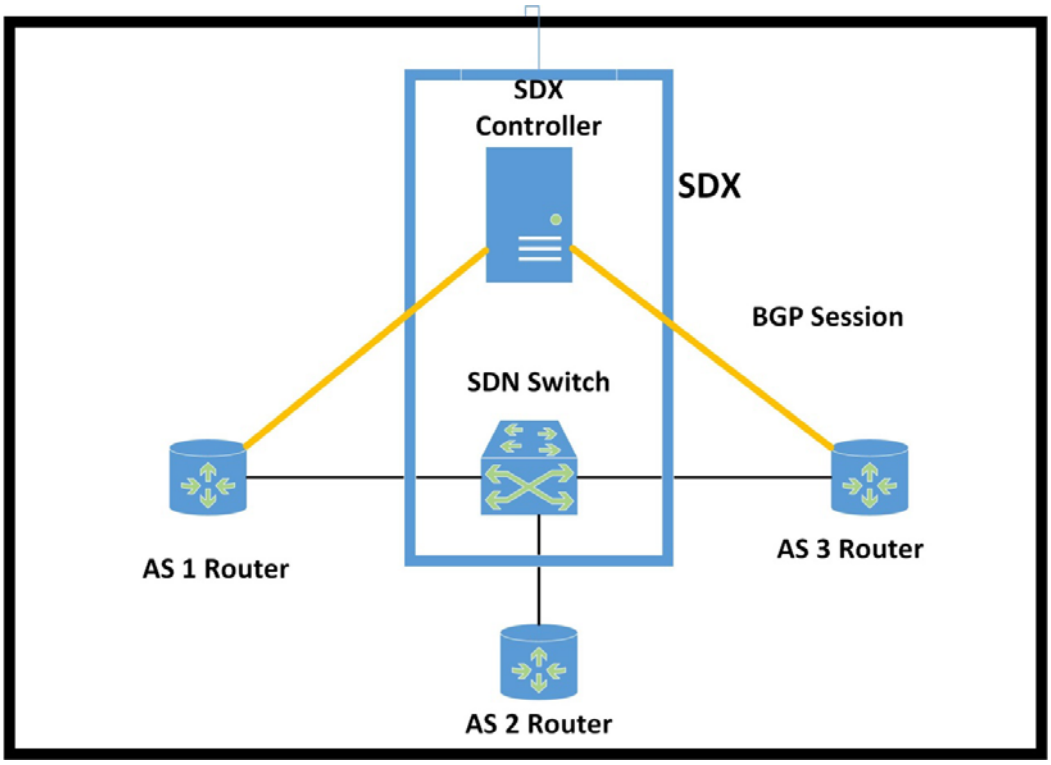


Figure 23. SDX layout: SDX= SDN + IXP

This seemingly simple change enables a wide variety of new applications. One can be prevention of Denial of Service attacks upstream. In the example in Figure 23 we have used AS C with two ingress routers at the exchange point. Let's suppose that AS C want to ensure that port 80 traffic enters its network via interface C1. This is very difficult to do with BGP, where you are not able to forward traffic in custom ways, based on specific fields in packet headers. On the other hand, an SDN enabled switch in an IXP makes it possible for a participant to install a rule that pertains only to a specific portion of packet header (this application however is not implemented in this deployment).

Denial of Service attacks are one of the most significant and serious problems currently facing the Internet users. For many users these attacks are an irritation, even if they understand the reason for the poor performance they occasionally observe. But for the Internet to achieve its full potential, it has to be able to offer highly reliable services, even in the face of hostility. The biggest DoS problem is caused by

distributed denial of service (DDoS) attacks, where the attacker compromises a large number of systems and then uses these as zombie systems to attack the victims. DDoS attacks of sufficient scale provide the firepower needed to overwhelm almost all victims. They can also be combined with spoofing or reflection to make the attack even more difficult to defend against. Currently most DDoS attacks do not bother to spoof the source addresses because, as no automatic push-back mechanism is widely deployed, it takes so long to shut down each zombie that there is no need to hide their identity [87].

DDoS is principally an issue due to widespread exploitation of software vulnerabilities, which permit the control of large numbers of compromised systems. To gain sufficient scale, such exploitation is typically automated using worms, viruses, or automated scanning from already-compromised hosts (so called "bots"). Fast-spreading worms are extremely hard to combat in the current Internet Architecture. Although viruses and bots are a serious issue, their spread rate is slower, which permits a wider range of defense options. A number of solutions have been proposed in the last over 10 years to fight DDos, and the fact the this is still an interesting and live topic, helps conclude that DDoS is still an issue in the current Internet, and therefore not fully addressed.

In SDN there is the ability to block denial-of-service in an significantly enhanced way, such as to block traffic in upstream autonomous systems, much closer to the source. For example if AS 1 is receiving Denial of Service attack traffic from AS 3 (Figure 24), the victim AS 1 can install drop rules potentially at multiple SDN enabled exchange points, thus preventing this traffic much closer to the source (a following section gives the sample code we used to install a drop rule at an exchange point).

In comparing to conventional defenses, SDX-based DDoS defense allows for an AS to remotely influence the traffic by remotely installing drop rules further upstream, and these rules can be more specific than in standard Access Control rules. For example these drop rules can be based on multiple header fields, such as source address, destination address, port number, etc. Furthermore, these drop rules can be coordinated across multiple IXPs therefore making it difficult for attack traffic to enter via any path.

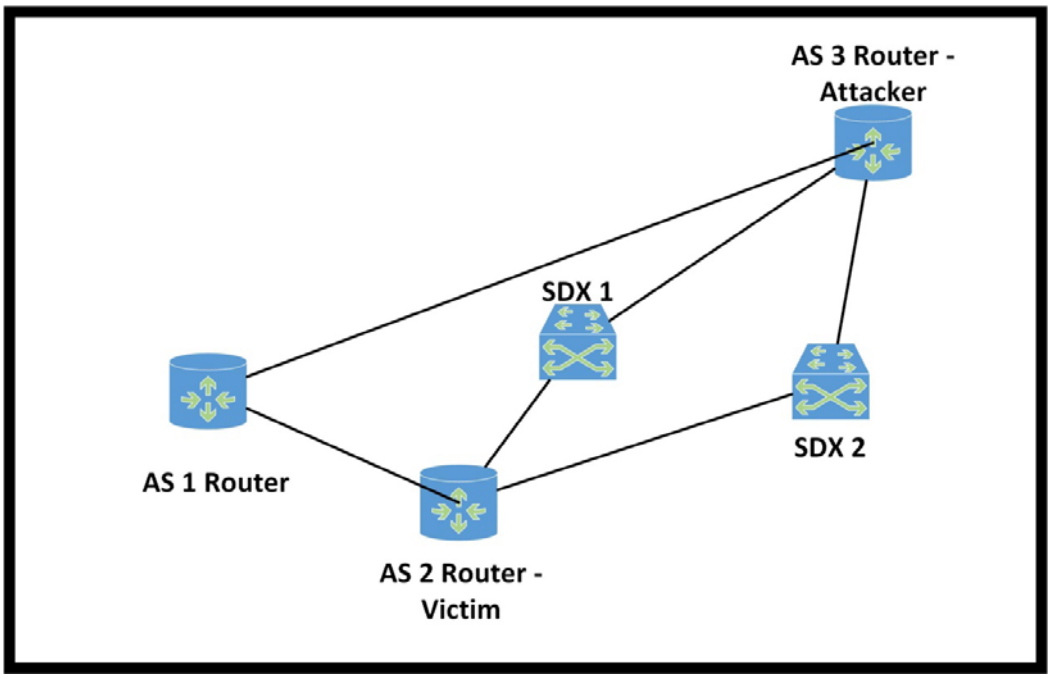


Figure 24. SDX preventing attack: AS 2 under attack originating from AS 3

In order to run experiment on testbed, or simulate it, users will need a set of tools. We have used a software router called Quagga (both Quagga and Mininext are SDX dependencies) in our case. Previously users need to have installed Mininet '(Mininet)' on Linux. To get Quagga, type:

$ sudo apt-get install quagga

Then users need to install Mininext (which is an extension of Mininet), using:

$ sudo apt-get install help2man python-setuptools

$ git clone https://github.com/sdn-ixp/sdx.git

The drop rule installation at the exchange point, given that the topology has been previously configured [88] is done through the following code:

$ curl -d '{"switch": "[DPID]", "name":"drop-flow", "src-ip": "IP1", "dst-ip":"IP2","active":"true"}' http://127.0.0.1:8080/wm/staticflowentrypusher/json

$ curl http://127.0.0.1:8080/wm/staticflowentrypusher/list/[DPID]/json | json_pp -t dumper

After applying the rule, we can see that it is applied and effective at the exchange point. This specific rule is given based on destination and source IP address. Note that actually the drop rule can be adjusted and modified based on port number or as a combination of multiple header fields (IP version, etc.).

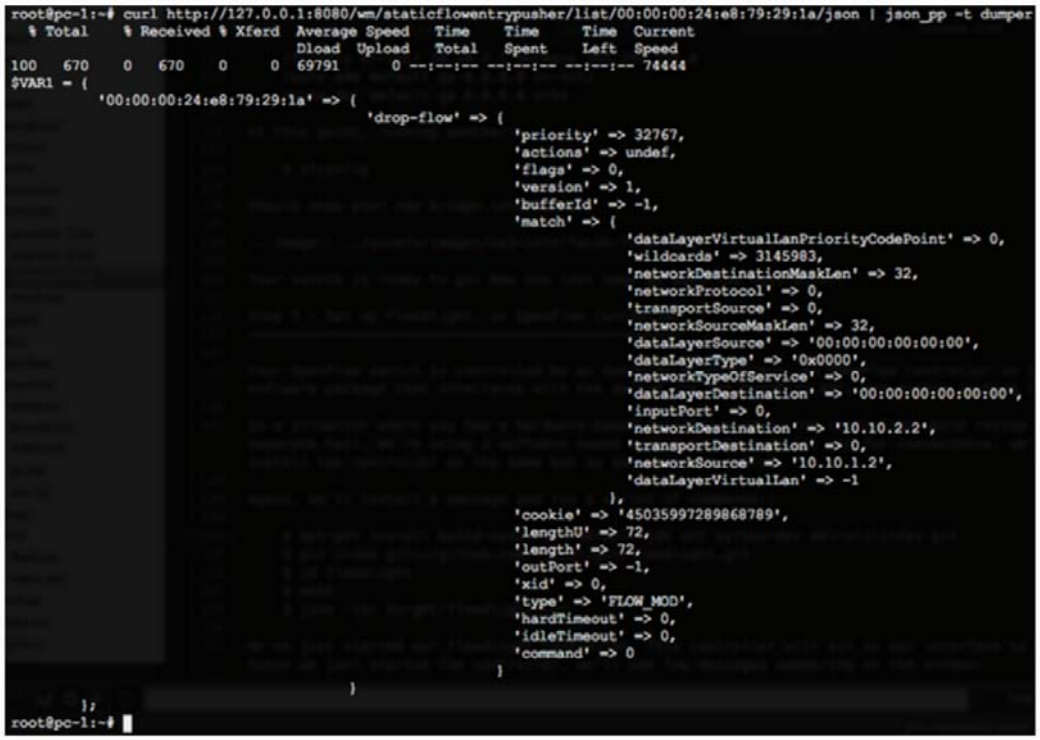


Figure 25. Drop rule installed at exchange point

The Internet is changing, and there are new challenges for content delivery and IXPs are playing an increasingly dominant role. SDN lets providers innovate and add new capabilities to the system.

In this model implementation, we have analyzed results gained from testing new policies for traffic engineering for inter-domain routing using Princeton's SDX platform, which is available. It can be seen that using SDN for interdomain routing proves not only useful, but much more practical for applications such as attack prevention and inbound traffic engineering, with more specific rules applying to specific traffic. In this implementation, we tested the suggested robustness of the proposed SDX, as a SDN Internet Exchange Point, by installing drop rules at one exchange in based on source and destination IP address. This proves to be easier, and more straightforward compared to traditional IXPs utilizing BGP.

Based on our conclusions on one hand we anticipate that SDN will be around in the future and that as new control and data applications become available, SDN will start to become part of more and more corporate, enterprise and university campuses.

Our future work will rely on current research work to test inter-domain routing applications in the SDN realm, as well as proposed specific solutions based on user needs. We are strongly confident that SDN as an organizing principle of data networks will have a future in major networks with hundreds or thousands (or even more) nodes, therefore based on this, our research will continue by running more exhaustive and different tests on various scenarios, topologies and data traffic, until we have analyzed thoroughly all aspects of our implementations.

5.3. Summary

This chapter consists of a collection of research papers and publications in the area of Software-Defined Networking. Each of the papers is given in its original form and in the format as they are submitted. Each one deals with practical implementation of SDN to model or simulate/emulate various networks inquiries, policy and management issues and are compared to traditional networks. While this represents original work, they treat issues like separation of network control from routers with OpenFlow, emulate enterprise network environments for fast transition to SDN, or deal with network performance issues of emulated data centers with unpredictable data patterns in SDN in order to improve performance. Lastly, it tackles the issue of improving traffic control by implementing and analyzing parameters for inter-domain

routing, as opposed to previous implementations that deal with intra-domain routing and traffic management.

The following chapter gives the final conlcusions of the thesis, as well as discusses future work.

CHAPTER VI. FINAL CONCLUSIONS AND FUTURE WORK

6.1. Overview

Networking still remains vertically integrated, where hardware comes with its proprietary software and is not open to innovation. Data networks have become increasingly complex nowadays. Even though technologies like Ethernet, IP protocol and packet forwarding are rather simple, control mechanisms like middleboxes, Access Control Lists (ACLs), firewalls, traffic engineering, VLANs, etc. have largely contributed to increasing their complexity. Primarily this is due to the lack of basic principles in networking. Denial of Service has been another concern for the last decades, and in the current Internet it has not been fully addressed. In inter-domain routing, BGP (Border Gateway Protocol) lacks sufficient flexibility for performing various traffic engineering and traffic management operations. It's generally possibly to route only on destination IP address blocks, and it is only possible to exert indirect control over how switches and routers forward traffic, through local preference, etc. But generally it is difficult to introduce new network services, such as ability to arbitrarily route traffic flows through middleboxes.

Software-Defined Networking (SDN) instead decouple the data plane (which is and should remain the job of the physical routers) and control plane. The control plane in SDN is removed from the routers and switches, and instead is done in the edge of the network, thus allowing for third party software, open interface to devices regardless of hardware type and vendor, and easier management of networks.

However for the time being, only larger campuses and data centers are utilizing the advantages of SDN, primarily taking advantage of easier network management, better resource utilization, better load and traffic balance on network devices and routes. But majority of networks, mainly small and campus networks, those that constitute the largest number of corporate networks while smaller campuses and enterprises are not yet widely deploying SDN. The future of SDN is therefore uncertain and it tightly depends on apps that are to be developed for various types of networks.

The research questions that are discussed in the thesis, based on the abovementioned issues on one hand, and the literature review on the other, are:

a) Is SDN a feasible alternative to traditional campus and data-center networks?

b) Can network performance be improved with SDN in data centers with unpredictable traffic patterns?

c) Can SDN be successfully implemented for traffic between various Autonomous Systems (AS)?

d) Does OpenFlow allow for better network security and traffic engineering?

During the first decade of this centure, a numerous reserarch articles and projects were carried out in order to propose ways to mitigate and reduce network complexity in a systematic manner, but separating data plane from control plane in the network. Furthermore, this decoupling would also allow for remote administrator and writing of easier third party applications for network and traffic management for network administrators. The real problem was how to access the router in order to install these policies in it, knowing that there was no such interface to interact with the router.

A paper which stresses the need and gives a specific idea for a network operating system is NOX. This “operating system” provides a uniform and centralized programmatic interface to the entire network. Analogous to the read and write access to various resources provided by computer operating systems, a network operating system provides the ability to observe and control a network. A network operating system does not manage the network itself; it merely provides a programmatic interface. Applications implemented on top of the network operating system perform the actual management tasks. The programmatic interface should be general enough to support a broad spectrum of network management applications.

A very nice OpenFlow controller is proposed with Beacon. Beacon is a Java-based open source OpenFlow controller created in 2010. It has been widely used for teaching, research, and as the basis of Floodlight. This paper describes the architectural decisions and implementation that achieves three of Beacon’s goals: to improve developer productivity, to provide the runtime ability to start and stop existing and new applications, and to be high performance.

Frenetic is a language for Software-Defined Networks proposed in 2013. In the Frenetic project, authors are designing simple and intuitive abstractions for programming the three main stages of network management: (i) monitoring network traffic, (ii) specifying and composing packet-forwarding policies, and (iii) updating policies in a consistent way. Overall, these abstractions make it dramatically easier for programmers to write and reason about SDN applications.

NetCore is a compiler and run-time system for SDN, where authors define a high-level, declarative language, called NetCore, for expressing packet-forwarding policies on SDNs. NetCore is expressive, compositional, and has a formal semantics.

Procera ia a language for High-Level Reactive Network Control, where a control architecture is proposed for software-defined networking (SDN) that includes a declarative policy language based on the notion of functional reactive programming;

Mininet is a system for prototyping large networks on the constrained resources. The lightweight approach of using OS-level virtualization features, including processes and network namespaces, allows it to scale to hundreds of nodes.

Based on the literature review, one can conclude that the research questions posed in this thesis are not fully answered, and the proposed models are not comprehensive in terms of both intra-domain and inter domain routing. SDN itself is a new phenomem in data networking, and therefore it is not widely deployed in networks, although almost all latest routers do implement OpenFlow as an interface that allows for a new way of interaction not only with a specific router, but rather with the whole network (the logical topology of the network). Most issues in traditional networks continue to persist, whereas the future of SDN directly depends on introduced models and applications.

6.2. Original contribution

The contribution of the thesis is as following:

- Review of the existing approaches for Software-Defined Networks implementations for campus networks and datacenters.
- Review of the current methods and techniques for traffic engineering and defense for inter-domain routing.
- Building a model for SDN implementation for campus and enterprise networks.
- Setting up case studies for various policy implementations, traffic forwarding dissemination and network performance investigation.
- Validating the model through practical implementation and results comparison with other traditional networks implementation.
- Building a model for SDN implementation for inter-domain routing between Autonomous Systems.
- Implementation of techniques to fight against network attacks, such as DDoS attacks in SDN.
- Implementation of techniques and policies at Internet exchanges for custom traffic engineering for inter-domain routing.
- Validating the developed models using existing SDX platform. The models were validated with examples of monitoring different parameters, such as IP packet header, network protocol or port number.

In this thesis we propose models for enterprise Software-Defined networks using Python and Mininet. We have also built a conventional network using ns-3 with the same number of nodes, topology, etc. Our tests show that our proposed network model outperforms the conventional/existing model in parameters like minimum and mean delay for ICMP and HTTP packets, as well as minimum and maximum convergence time, and therefore are a best solution to transition corporate networks with hundreds or more nodes, mainly due to the fast propagation of routing policies from the central controller. As we have seen in our models, we can conclude that Software-Defined Networks can perform well in enterprise networks, as far as both data packets and control packets are concerned. This, having in mind the fact that we have used two different platforms for the test, namely the well known ns-3 simulator for the conventional networks, and Mininet to emulate the virtual OpenFlow network. This is due to the fact that Software-Defined Networks have specific hardware and

software requirements and practically it is impossible to carry out tests and compare SDN and non-SDN networks in a same virtual environment.

However, both tests have been carried out on the same platform, with the same hardware configuration. Also, the tests have been carried out in a specific node arrangement, which represents one specific topology, and similar tests can be carried out for different types of topologies. Based on the tests for ping, HTTP, and network convergence, our conclusion is that Software-Defined networks perform faster in all cases, except for the maximum delay of ICMP and HTTP packets. However, this is only valid for the very first packet that goes through the OVS switch, until the switch is trained.

The thesis also analyzes results gained from testing new policies for traffic engineering for inter-domain routing using Princeton's SDX platform, which is available. It can be seen that using SDN for interdomain routing proves not only useful, but much more practical for applications such as attack prevention and inbound traffic engineering, with more specific rules applying to specific traffic. We tested the suggested robustness of the proposed SDX, as a SDN Internet Exchange Point, by installing drop rules at one exchange in based on source and destination IP address. This proves to be easier, and more straightforward compared to traditional IXPs utilizing BGP.

6.3. Future research

SDN is a new set of abstraction with many unanswered questions related to practical implementations. The issues of mobility, security and privacy will have to be addressed in the future, as new control programs emerge on top of existing physical infrastructures. The third party apps developed by different network programmers will have to be fully validated and checked for security holes before they are implemented in actual networks. But SDN also allows for rapid prototyping at software speeds, not having to wait for vendors to come up with new features in networks. This is where we will focus our research work in the future. We will try to implement new routing policies in existing network environments using SDN and Frenetic in a real network, thus adding extra features to our network that the actual routers do not have. We will

also focus on mobility, as most network devices nowadays are mobile.

6.4. Overall conclusions

In today's networks we can easily says that topology is policy, meaning the actual physical location of routers, firewalls, etc. dictates how effective the network is, how well the ACLs work, what our broadcast domains are, etc. When networks are moved to the cloud, we usually want to keep same policies, but very few networks operators have an abstract expression of network policy, rather they have a network topology. SDN allows specifying a logical topology to the cloud. The cloud then ignores the physical topology and follows the logical topology based on the policy read based on the topology initially.

SDN allows for implementation of control function in the edge of the network, instead of in the routers, where it actually is, and the core only deals with delivering packets end-to-end. The core may easily remain legacy hardware and the network operators need not to know at all the one is implementing SDN from the own edge of the network. So it simplifies network troubleshooting and also there are no disruption periods during network convergence, due to the fact the one policy is implemented per specific flow, and each packet will be carried either by an old state, or by a new state. In traditional routing packets may well be lost during network convergence, or loops may appear. In SDN the expressions are high level and easily checked and corrected.

The SDN gives an extra benefit to the network operators to more easily customize their networks based on their needs. The policies, traffic engineering, monitoring and security is easier to implement after getting a network state quickly from the Network Operating Systems, and moving the network to a virtual environment without much effort is perhaps the greatest benefit in this specific scenario. SDN not only gives the possibility to do this, but more importantly, because the control plane now resides on the edge, it can be written by anyone and is vendor independent, it can be done in a very short period of time and tested in real time.

With all of the debate about OpenFlow and SDN progress, it's real-world deployments that will eventually tell the story. Many market participants have expressed frustration that commercial deployment is taking longer to develop than initially thought, but there are actual deployments taking place at both service providers and enterprises.

As per our hypothesis laid out in this thesis, we can conclude the folowing:

Hypothesis H1: 'Large datacenters and campus networks can be managed more easily using Software Defined Networking applications by creating logical overlays of network topologies' - is confirmed based on practical implementation results;

Subhypothesis H1a: 'Routers hardware and software decoupling and development of customized third party applications enables fast transition to Software Defined Networking' - is confirmed based on simulations and emulations presented in the thesis.

Hypothesis H2: 'Software Defined Networking applications enables an improved traffic control for inter-domain and intra-domain routing over traditional (non-SDN) networks' - is confirmed based on practical model measurements and data analysis.

Hypothesis H3: 'Software Defined Networking allows for better security and traffic engineering in inter-domain routing over traditional (non-SDN) networks.' – is partially confirmed for a number of test case scenarios for DDoS attacks. This does not exhaust all possible scenarios and types of attack.

CHAPTER VII. BIBLIOGRAPHY AND APPENDICES

7.1. List of references

[1] http://blogs.gartner.com/andrew-lerner/2015/12/15/predicting-sd-wan-adoption/

[2] http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html

[3] http://sunsite.uakom.sk/sunworldonline/swol-02-1997/swol-02-sunspots.html

[4] https://www.scribd.com/document/237513189/Software-Defined-Network-SDN. Retrieved Oct 26, 2014.

[5] https://www.opennetworking.org/sdn-resources/sdn-definition

[6] http://archive.openflow.org/documents/openflow-wp-latest.pdf

[7] http://yuba.stanford.edu/~casado/nox-ccr-final.pdf

[8] http://yuba.stanford.edu/~derickso/docs/hotsdn15-erickson.pdf

[9] http://delivery.acm.org/10.1145/2630000/2620756/p145-agarwal.pdf?ip=31.11.87.42&id=2620756&acc=OPENTOC&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2EE994ED6114094BD1&CFID=905142621&CFTOKEN=22331935&__acm__=1488128779_90591309bbef31583778a79f3fa31d96. Retrieved Feb 02, 2017.

[10] http://frenetic-lang.org/publications/overview-ieeecoms13.pdf

[11] http://www.cs.princeton.edu/~jrex/papers/icfp11.pdf

[12] http://www.cs.princeton.edu/~dpw/papers/ncore-popl12.pdf

[13] http://conferences.sigcomm.org/sigcomm/2012/paper/hotsdn/p43.pdf

[14] http://klamath.stanford.edu/~nickm/papers/a19-lantz.pdf

[15] https://www.nsnam.org/

[16] "Data Communications and Networking" 2nd edition, Behrouz Forouzan.

[17] https://en.wikiquote.org/wiki/Ken_Olsen

[18] Marcelo Yannuzzi, 'Open issues in interdomain routing: a survey', 2005.

[19] http://ns3-code.com/ns3-sdn-projects/

[20] Nick McKeown; et al. (April 2008). "OpenFlow: Enabling innovation in campus networks". ACM Communications Review. Retrieved 2009-11-02.

[21]https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

[22] https://en.wikipedia.org/wiki/OpenFlow

[23] Members of the Open Networking Foundation.
https://www.opennetworking.org/membership/members

[24] M. Casado, M. Freedman, J. Pettit, J. Luo, N. McKeown, S. Shenker. Ethane: Taking Control of the Enterprise. In Sigcomm 2007.

[25] M. Casado. What OpenFlow is (and more importantly, what it's not).
http://networkheresy.com/2011/06/05/what-openflow-is-and-more-importantly-what-its-not/

[26] M. Casado, T. Koponen, S. Shenker, A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In HotSDN 2012

[27] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker. Composing Software-Defined Networks. In NSDI 2013.

[28] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker. Abstractions for Network Update. In Sigcomm 2012.

[29] M. Canini, D. Venzano, P. Perešini, D. Kostić, J. Rexford. A NICE Way to Test OpenFlow Applications. In NSDI 2012.

[30] M. Casado. The Scaling Implications of SDN.
http://networkheresy.com/2011/06/08/the-scaling-implications-of-sdn/

[31] A. Nayak, A. Reimers, N. Feamster, R. Clark. Resonance: Dynamic Access Control in Enterprise Networks. In WREN 2009.

[32] A. Tavakoli, M. Casado, T. Koponen, S. Shenker. Applying NOX to the Datacenter. In HotNets 2009.

[33] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, R. Johari. Plug-n-Serve: Load-balancing Web Traffic using OpenFlow. In Sigcomm 2009 Demo.

[34] List of OpenFlow Software Projects http://yuba.stanford.edu/~casado/of-sw.html

[35] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker. NOX: Towards an Operating System for Networks. In CCR 2008

[36] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, D. Walker. Frenetic: A Network Programming Language. In ICFP 2011.

[37] H. Kim, N. Feamster. Improving Network Management with Software Defined Networking. In IEEE Communications Magazine Feb 2013.

[38] M. Bansal, J. Mehlman, S. Katti, P. Levis. OpenRadio: A Programmable Wireless Dataplane. In HotSDN 2012.

[39] L. E. Li, Z. Morley Mao, J. Rexford. Towards Software-Defined Cellular Networks. In EWSDN 2012.

[40] D. D. Clark, “The Design Philosophy of the DARPA Internet Protocols”, SIGCOMM „88, Computer Communication Review Vol. 18, No. 4, August 1988, pp. 106–114

[41] R. Braden, Ted FaberMark Handley, “From Protocol Stack to Protocol Heap – RoleBased Architecture”, ACM SIGCOMM Computer Communications Review, HotNets I, Princeton, NJ, USA, October 2002

[42] T. Koponen, M. Chawla, B. G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker and I. Stoica, “ A Data-Oriented (and Beyond) Network Architecture, SIGCOMM‟07, August 27–31, 2007, Kyoto, Japan

[43] R. Jain, “Internet 3.0: Ten Problems with Current Internet Architecture and Solutions for the Next Generation”, Proceedings IEEE Military Communications Conference (Milcom 2006), Washington DC, October 23-25, 2006

[44] S. Ratnasamy, S. Schenker and S. McCanne, “Towards an Evolvable Internet Architecture”, 2005, SIGCOMM‟05 August 21–26,2005, Philadelphia, Pennsylvania, USA, p.2

[45] D. D. Clark, K. Sollins, J. Wroclawski and T. Faber, “Addressing Reality: An Architectural Response to Real-World Demands on the Evolving Internet”, ACM SIGCOMM 2003 Workshops, August 25&27, 2003, Karlsruhe, Germany, p.3

[46] “GENI”, http://geni.net

[47] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger and S. Shenker, “DDoS Defense by Offense”, SIGCOMM '06, September 11.15, 2006, Pisa, Italy

[48] D. X. Song and A. Perrig, “Advanced and Authenticated Marking Schemes for IP Traceback,” in Proceedings IEEE INFOCOM, 2001.

[49] D. Dean, M. Franklin, and A. Stubblefield, “An Algebraic Approach to IP Traceback,” in Proc. 8th Network and Distributed System Security Symposium, 2001.

[50] A. Yaar, A. Perrig, and D. Song, "FIT: Fast Internet Traceback", in Proceedings IEEE INFOCOM, 2005

[51] V. Paruchuri, A. Durresi and R. Jain, “On the (in)effectiveness of Probabilistic Marking for IP Traceback under DDoS Attacks”

[52] D. R. Kuhn, “Sources of Failure in the Public Switched Telephone Network”, IEEE Computer, Vol. 30, No. 4 (April, 1997).

[53] Establishing End to End Trust, Scott Charney, Microsoft Corp.2008.

[54] http://www.potaroo.net/tools/ipv4/index.html, Retrieved Feb 13, 2017.
[55] J. Saltzer, “On the Naming and Binding of Network Destinations In Local Computer Networks”, North-Holland Publishing Company, Amsterdam, 1982, pp. 311-317. Reprinted as RFC 1398, August 1993.
[56] Y. Chu, S. G. Rao, and H. A Zhang, “Case for end system multicast. In Proc. of ACM SIGMETRICS’00 (Santa Clara, CA, June 2000), pp. 1–12.
[57] H. Holbrook, and D. Cheriton, “IP multicast channels: Express support for large-scale single-source applications”, in Proc. of ACM SIGCOMM’99 (Cambridge, Massachusetts, Aug. 1999), pp. 65–78.
[58] I. Stoica, T. Ng, and H. Zhang, “A recursive unicast approach to multicast”, in Proc. of INFOCOM’00 (Tel-Aviv, Israel, Mar. 2000), pp. 1644–1653
[59] G. Ballintijn, M. v. Steen, and A. S.Tanenbaum, “Scalable Human-Friendly Resource Names” Oct. 2001
[60] “RFC 1737”, http://www.apps.ietf.org/rfc/rfc1737.html
[61]”Apple Inc.”, http://www.apple.com/pr/library/2008/03/06iphone.html
[62] A. C. Snoeren, and H. Balakrishnan, “An end-to-end approach to host mobility”, in proc. of ACM/IEEE MOBICOM’99 (Cambridge, MA, Aug. 1999).
[63] I. Stoica, D. Adkins, S. Zhuang, S. Shenker and S. Surana “Internet Indirection Infrastructure”, SIGCOMM ‟02 Pittsburgh, Pennsylvania USA
[64] M. S. Blumenthal and D. D. Clark, “Rethinking the Design of the Internet: The End-to-End Arguments vs. the Brave New World”, ACM Transactions on Internet Technology, Vol. 1, No. 1, August 2001, Pages 70–109.
[65] Foundation, “OpenFlow Switch Specification (Version 1.3.0),” June 2012.
[66] http://yuba.stanford.edu/~derickso/docs/hotsdn15-erickson.pdf
[67]https://www.opennetworking.org/images/stories/downloads/sdn-resources/special-reports/Special-Report-OpenFlow-and-SDN-State-of-the-Union-B.pdf
[68] https://www.opennetworking.org/our-members
[69] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner. OpenFlow: Enabling Innovation in Campus Networks. In CCR 2008.
[70] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. “NOX: towards an operating system for networks”, ACM SIGCOMM

Computer Communication Review, 38(3):105–110, 2008.

[71] B. Lantz, B. Heller, N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In HotNets 2010.

[72] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, D. Walker. Languages for Software-Defined Networks. In IEEE Communication Magazine Feb 2013.

[73] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. "A NICE way to test OpenFlow applications", in Proceedings of USENIX NSDI '12, April 2012.

[74] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, "Languages for software-defined networks". Communications Magazine, IEEE, 51(2):128–134, 2013.

[75] M. Gupta, J. Sommers, P. Barford, "Fast, Accurate Simulation for SDN Prototyping", HotSDN '13, August 16, 2013, Hong Kong, China

[76] POX. http://www.noxrepo.org/pox/about-pox/.

[77] N. Gude, "NOX: towards an operating system for networks", SIGCOMM CCR, 38(3), 2008.

[78] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation" in Proceedings of ACM CoNeXT '12, pages 253–264, 2012.

[79] V. Yazıcı, "Controlling a Software-Defined Network via Distributed Controllers", in NEM Summit, 2012.

[80] T. Koponen, "Onix: A distributed control platform for large-scale production networks", OSDI, 2010.

[81] S. H. Yaganeh, A. Tootoonchian, Y. Ganjali. On the Scalability of Software-Defined Networking. In IEEE Communications Magazine Feb 2013.

[82] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network", in Proceedings of ACM SIGCOMM '09, August 2009.

[83] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: scalable ethernet for data centers", in Proceedings of ACM CoNeXT '12, pages 49–60, 2012.

[84] A. Tootoonchian, "On controller performance in software-defined networks", in

HotICE, 2012.

[85] Mininet, http://mininet.org/

[86] Open Networking Foundation, https://www.opennetworking.org/

[87] M. Durresi, L. Barolli, V. Paruchuri, A. Durresi, “Scalable traceback against distributed denial of service”, IJWGS, 2006.

[88] Sh. Latifi, A. Durresi, B. Cico, “Emulating enterprise network environments for fast
transition to Software-Defined Networking”, MECO, 2014.

7.2. Appendix - Codes

Network simulation - basic code for creating network topology in Ns-3.

```
NS_LOG_INFO (“Create nodes.”);
NodeContainer nodes;
nodes.Create (2);
NS_LOG_INFO (“Create channels.”);
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute (“DataRate”, StringValue (“500Kbps”));
pointToPoint.SetChannelAttribute (“Delay”, StringValue (“5ms”));
NetDeviceContainer devices;
devices = pointToPoint.Install (nodes);
InternetStackHelper internet;
internet.Install (nodes);
NS_LOG_INFO (“Assign IP Addresses.”);
Ipv4AddressHelper ipv4;
ipv4.SetBase (“10.10.10.0”, “255.255.255.0”);
Ipv4InterfaceContainer i = ipv4.Assign (devices);
NS_LOG_INFO (“Create Applications.”);
uint16_t port = 9;
BulkSendHelper source (“ns3::TcpSocketFactory”,
InetSocketAddress (i.GetAddress (1), port));
source.SetAttribute (“MaxBytes”, UintegerValue (maxBytes));
ApplicationContainer sourceApps = source.Install (nodes.Get (0));
sourceApps.Start (Seconds (0.0));
sourceApps.Stop (Seconds (10.0));
```

Network topology implementation in python

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

```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf

def myNetwork():

   net = Mininet( topo=None,
                        autoStaticArp=True,
              build=False,
             ipBase='10.0.0.0/8')
   info( '*** Adding controller\n' )
   c1=net.addController(name='c1',
                  controller=RemoteController,
                  ip='192.168.0.11',
                  port=6633)
   c0=net.addController(name='c0',
                  controller=RemoteController,
                  ip='192.168.0.21',
                  port=6633)

   info( '*** Add switches\n')
   s6 = net.addSwitch('s6', cls=OVSKernelSwitch, dpid='0000000000000206')
   s5 = net.addSwitch('s5', cls=OVSKernelSwitch, dpid='0000000000000205')
   s4 = net.addSwitch('s4', cls=OVSKernelSwitch, dpid='0000000000000204')
   s3 = net.addSwitch('s3', cls=OVSKernelSwitch, dpid='0000000000000203')
   s2 = net.addSwitch('s2', cls=OVSKernelSwitch, dpid='0000000000000202')
   s1 = net.addSwitch('s1', cls=OVSKernelSwitch, dpid='0000000000000201')
```

```
info( '*** Add hosts\n')
h6 = net.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)
h5 = net.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
h4 = net.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
info( '*** Add links\n')
net.addLink(h6, s6)
net.addLink(h5, s5)
net.addLink(h4, s4)
net.addLink(h3, s3)
net.addLink(h2, s2)
net.addLink(h1, s1)
info(' *** switch to switch \n')
net.addLink(s6, s3)
net.addLink(s5, s2)
net.addLink(s4, s1)

info( '*** Starting network\n')
net.build()
info( '*** Starting controllers\n')
for controller in net.controllers:
   controller.start()
info( '*** Starting switches\n')
net.get('s6').start([c0])
net.get('s5').start([c0])
net.get('s4').start([c0])
net.get('s3').start([c0])
net.get('s2').start([c0])
net.get('s1').start([c0])

info( '*** Configuring switches\n')
```

```
    CLI(net)
    net.stop()
if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```

Implementation of remote controller of data plane traffic in python

```
#!/usr/bin/python
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, UserSwitch
from mininet.link import TCLink
from mininet.topo import Topo
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from time import sleep
import os
import signal
DATA_CONTROL_IP='192.168.123.254'
DATA_CONTROL_PORT=6634
OFPP_CONTROLLER=65533
class DataController( RemoteController ):
    def checkListening( self ):
        "Ignore spurious error"
        pass
class MininetFacade( object ):
  """Mininet object facade that allows a single CLI to
       talk to one or more networks"""
    def __init__( self, net, *args, **kwargs ):
        """Create MininetFacade object.
           net: Primary Mininet object
           args: unnamed networks passed as arguments
```

```
        kwargs: named networks passed as arguments"""
    self.net=net
    self.nets=[ net ] + list( args ) + kwargs.values()
    self.nameToNet=kwargs
    self.nameToNet['net']=net
def __getattr__( self, name ):
    "returns attribute from Primary Mininet object"
    return getattr( self.net, name )
def __getitem__( self, key ):
    "returns primary/named networks or node from any net"
    #search kwargs for net named key
    if key in self.nameToNet:
        return self.nameToNet[ key ]
    #search each net for node named key
    for net in self.nets:
        if key in net:
            return net[ key ]
def __iter__( self ):
    "Iterate through all nodes in all Mininet objects"
    for net in self.nets:
        for node in net:
            yield node
def __len__( self ):
    "returns aggregate number of nodes in all nets"
    count=0
    for net in self.nets:
        count += len(net)
    return count
def __contains__( self, key ):
    "returns True if node is a member of any net"
    return key in self.keys()
def keys( self ):
    "returns a list of all node names in all networks"
    return list( self )
```

```
    def values( self ):
        "returns a list of all nodes in all networks"
        return [ self[ key ] for key in self ]
    def items( self ):
        "returns (key,value) tuple list for every node in all networks"
        return zip( self.keys(), self.values() )
class TestTopology( Topo ):
    def __init__( self, control_network=False, dpid_base=0, **kwargs ):
        Topo.__init__( self, **kwargs )
        # Generate test topology (star of 5 switches)
        switches=[]
        for switch_id in range(0, 5):
            dpid="%0.12X" % (dpid_base + switch_id)
            if control_network:
                switches.append(self.addSwitch('cs%s' % switch_id, inNamespace=True,
dpid=dpid))
            else:
                switches.append(self.addSwitch('s%s' % switch_id, inband=True,
inNamespace=True, dpid=dpid))
        self.addLink(switches[0], switches[1], bw=10, delay='1ms')
        self.addLink(switches[0], switches[2], bw=10, delay='1ms')
        self.addLink(switches[0], switches[3], bw=10, delay='1ms')
        self.addLink(switches[0], switches[4], bw=10, delay='1ms')
        if control_network:
            # Add a host which will run the controller for the data network
            pox=self.addHost('pox', cls=DataController, ip=DATA_CONTROL_IP,
port=DATA_CONTROL_PORT, inNamespace=True)
            self.addLink(pox, switches[0], bw=10, delay='1ms')
def run():
    "Create control and data networks, and launch POX"
    # Launch the control network
    info('* Creating Control Network\n')
    ctopo=TestTopology(control_network=True, dpid_base=100)
```

```
   cnet=Mininet(topo=ctopo, ipBase='192.168.123.0/24', switch=UserSwitch,
controller=None,
link=TCLink, build=False, autoSetMacs=False)
   info('* Adding Control Network Controller\n')
   cnet.addController('cc0', controller=Controller)
   info('* Starting Control Network\n')
   cnet.start()
   # Launch the network controller for the data network
   pox_process=None
   pox_arguments=['pox.py', 'log', '--file=pox.log,w', 'openflow.of_01', '--address=%s'
%
DATA_CONTROL_IP, '--port=%s' % DATA_CONTROL_PORT,
'openflow.discovery', 'log.level', '--DEBUG']
   info('* Launching POX: ' + ' '.join(pox_arguments) + '\n')
   # Direct POX stdout and stderr to /dev/null
   with open(os.devnull, "w") as fnull:
      pox_process=cnet.get('pox').popen(pox_arguments, stdout=fnull, stderr=fnull,
shell=False, close_fds=True)
   sleep(1)    # Arbitrary delay to allow controller to initialize
   # Launch the data network
   info('* Creating Data Network\n' )
   topo=TestTopology(control_network=False, dpid_base=0)
   net=Mininet(topo=topo, switch=UserSwitch, controller=None, link=TCLink,
build=False, autoSetMacs=False)
   info('* Adding Data Controllers to Data Network\n')
   for host in cnet.hosts:
      if isinstance(host, DataController):
         net.addController(host)
         info('* Added DataController Running on Host "%s" to Data Network\n' %
str(host))
   info('* Starting Data Network\n')
   net.start()
   info('* Connect Data Network Switch Management Interfaces to Control Network
Switches\n')
```

```
    for switch1 in net.switches:
      if isinstance(switch1, UserSwitch):
          switch_id=int(str(switch1)[1:])
          switch2=cnet.get('cs%s' % switch_id)
          cnet.addLink(node1=switch1, node2=switch2, port1=OFPP_CONTROLLER)
          info('* Connected Management Interface of Switch "%s" to Switch "%s"\n' %
(str(switch1), str(switch2)))
    info('* Network Intialization Complete\n')
    mn=MininetFacade(net, cnet=cnet)
    CLI(mn) # POX should be receiving control traffic at this point
    info('* Stopping POX\n')
    pox_process.send_signal(signal.SIGINT)
    pox_process.wait()
    info('* Stopping Data Network\n')
    net.stop()
    info('* Stopping Control Network\n')
    cnet.stop()
if __name__ == '__main__':
    setLogLevel( 'info' )
    run()
```

Mininet test after installation:

```
sudo mn --test pingall
```

Mininet custom topology example:

```
 Two directly connected switches plus a host for each switch:
    host --- switch --- switch --- host
  Adding the 'topos' dict with a key/value pair to generate our newly defined
 topology enables one to pass in '--topo=mytopo' from the command line.
 """
  from mininet.topo import Topo
  class MyTopo( Topo ):
```

```
    "Simple topology example."
     def __init__( self ):
        "Create custom topo."
         # Initialize topology
        Topo.__init__( self )
         # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )
         # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )
  topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Mininet custom topology example:

host --- switch --- switch --- host

```
from mininet.topo import Topo
class MyTopo( Topo ):
   "Simple topology example."
   def __init__( self ):
      "Create custom topo."
      # Initialize topology
      Topo.__init__( self )
      # Add hosts and switches
      Host1 = self.addHost( 'h1' )
      Host2 = self.addHost( 'h2' )
      Host3 = self.addHost( 'h3' )
      Host4 = self.addHost( 'h4' )
      Host5 = self.addHost( 'h5' )
      Host6 = self.addHost( 'h6' )
```

```
        Host7 = self.addHost( 'h7' )
        Host8 = self.addHost( 'h8' )
        Switch1 = self.addSwitch( 's1' )
        Switch2 = self.addSwitch( 's2' )
        Switch3 = self.addSwitch( 's3' )
        Switch4 = self.addSwitch( 's4' )

        # Add links
        self.addLink( Host1, Switch1 )
        self.addLink( Host2, Switch1 )
        self.addLink( Host3, Switch2 )
        self.addLink( Host4, Switch2 )
        self.addLink( Host5, Switch3 )
        self.addLink( Host6, Switch3 )
        self.addLink( Host7, Switch4 )
        self.addLink( Host8, Switch4 )
        self.addLink( Switch1, Switch2 )
        self.addLink( Switch2, Switch3 )
        self.addLink( Switch3, Switch4 )
topos = { 'mytopo': ( lambda: MyTopo() ) }



#!/usr/bin/python

 """
 Create a 1024-host network, and run the CLI on it.
 """

  from mininet.cli import CLI
 from mininet.log import setLogLevel
 from mininet.node import OVSSwitch
 from mininet.topolib import TreeNet

 if __name__ == '__main__':
```

```
    setLogLevel( 'info' )
    network = TreeNet( depth=2, fanout=32, switch=OVSSwitch )
    network.run( CLI, network )
```

```
#!/usr/bin/python

"""
Example to create a Mininet topology and connect it to the internet via NAT
"""

from mininet.cli import CLI
from mininet.log import lg, info
from mininet.topolib import TreeNet

if __name__ == '__main__':
    lg.setLogLevel( 'info')
    net = TreeNet( depth=1, fanout=4 )
    # Add NAT connectivity
    net.addNAT().configDefault()
    net.start()
    info( "*** Hosts are running and should have internet connectivity\n" )
    info( "*** Type 'exit' or control-D to shut down network\n" )
    CLI( net )
    # Shut down NAT
    net.stop()
```

```
#!/usr/bin/python

"""
This example creates a multi-controller network
using the net.add*() API and manually starting the switches and controllers.
```

```
"""

from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info

def multiControllerNet():
    "Create a network from semi-scratch with multiple controllers."

    net = Mininet( controller=Controller, switch=OVSSwitch )

    info( "*** Creating (reference) controllers\n" )
    c1 = net.addController( 'c1', port=6633 )
    c2 = net.addController( 'c2', port=6634 )

    info( "*** Creating switches\n" )
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )

    info( "*** Creating hosts\n" )
    hosts1 = [ net.addHost( 'h%d' % n ) for n in ( 3, 4 ) ]
    hosts2 = [ net.addHost( 'h%d' % n ) for n in ( 5, 6 ) ]

    info( "*** Creating links\n" )
    for h in hosts1:
        net.addLink( s1, h )
    for h in hosts2:
        net.addLink( s2, h )
    net.addLink( s1, s2 )

    info( "*** Starting network\n" )
    net.build()
```

```
    c1.start()
    c2.start()
    s1.start( [ c1 ] )
    s2.start( [ c2 ] )

    info( "*** Testing network\n" )
    net.pingAll()

    info( "*** Running CLI\n" )
    CLI( net )

    info( "*** Stopping network\n" )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )  # for CLI output
    multiControllerNet()
```

```
#!/usr/bin/env python
"""
```

Network topology creation. This package includes code to represent network topologies.

```
"""

from mininet.util import irange, natural, naturalSeq

class MultiGraph( object ):
    "Utility class to track nodes and edges - replaces networkx.MultiGraph"

    def __init__( self ):
```

```
    self.node = {}
    self.edge = {}

def add_node( self, node, attr_dict=None, **attrs):
    """Add node to graph
      attr_dict: attribute dict (optional)
      attrs: more attributes (optional)
      warning: updates attr_dict with attrs"""
    attr_dict = {} if attr_dict is None else attr_dict
    attr_dict.update( attrs )
    self.node[ node ] = attr_dict

def add_edge( self, src, dst, key=None, attr_dict=None, **attrs ):
    """Add edge to graph
      key: optional key
      attr_dict: optional attribute dict
      attrs: more attributes
      warning: udpates attr_dict with attrs"""
    attr_dict = {} if attr_dict is None else attr_dict
    attr_dict.update( attrs )
    self.node.setdefault( src, {} )
    self.node.setdefault( dst, {} )
    self.edge.setdefault( src, {} )
    self.edge.setdefault( dst, {} )
    self.edge[ src ].setdefault( dst, {} )
    entry = self.edge[ dst ][ src ] = self.edge[ src ][ dst ]
    # If no key, pick next ordinal number
    if key is None:
        keys = [ k for k in entry.keys() if isinstance( k, int ) ]
        key = max( [ 0 ] + keys ) + 1
    entry[ key ] = attr_dict
    return key

def nodes( self, data=False):
```

```
    """Return list of graph nodes
       data: return list of ( node, attrs)"""
    return self.node.items() if data else self.node.keys()

def edges_iter( self, data=False, keys=False ):
    "Iterator: return graph edges"
    for src, entry in self.edge.iteritems():
        for dst, keys in entry.iteritems():
            if src > dst:
                # Skip duplicate edges
                continue
            for k, attrs in keys.iteritems():
                if data:
                    if keys:
                        yield( src, dst, k, attrs )
                    else:
                        yield( src, dst, attrs )
                else:
                    if keys:
                        yield( src, dst, k )
                    else:
                        yield( src, dst )

def edges( self, data=False, keys=False ):
    "Return list of graph edges"
    return list( self.edges_iter( data=data, keys=keys ) )

def __getitem__( self, node ):
    "Return link dict for given src node"
    return self.edge[ node ]

def __len__( self ):
    "Return the number of nodes"
    return len( self.node )
```

```
    def convertTo( self, cls, data=False, keys=False ):
        """Convert to a new object of networkx.MultiGraph-like class cls
           data: include node and edge data
           keys: include edge keys as well as edge data"""
        g = cls()
        g.add_nodes_from( self.nodes( data=data ) )
        g.add_edges_from( self.edges( data=( data or keys ), keys=keys ) )
        return g


class Topo( object ):
    "Data center network representation for structured multi-trees."

    def __init__( self, *args, **params ):
        """Topo object.
           Optional named parameters:
           hinfo: default host options
           sopts: default switch options
           lopts: default link options
           calls build()"""
        self.g = MultiGraph()
        self.hopts = params.pop( 'hopts', {} )
        self.sopts = params.pop( 'sopts', {} )
        self.lopts = params.pop( 'lopts', {} )
        # ports[src][dst][sport] is port on dst that connects to src
        self.ports = {}
        self.build( *args, **params )

    def build( self, *args, **params ):
        "Override this method to build your topology."
        pass

    def addNode( self, name, **opts ):
```

```
    """Add Node to graph.
       name: name
       opts: node options
       returns: node name"""
    self.g.add_node( name, **opts )
    return name

def addHost( self, name, **opts ):
    """Convenience method: Add host to graph.
       name: host name
       opts: host options
       returns: host name"""
    if not opts and self.hopts:
        opts = self.hopts
    return self.addNode( name, **opts )

def addSwitch( self, name, **opts ):
    """Convenience method: Add switch to graph.
       name: switch name
       opts: switch options
       returns: switch name"""
    if not opts and self.sopts:
        opts = self.sopts
    result = self.addNode( name, isSwitch=True, **opts )
    return result

def addLink( self, node1, node2, port1=None, port2=None,
             key=None, **opts ):
    """node1, node2: nodes to link together
       port1, port2: ports (optional)
       opts: link options (optional)
       returns: link info key"""
    if not opts and self.lopts:
        opts = self.lopts
```

```
    port1, port2 = self.addPort( node1, node2, port1, port2 )
    opts = dict( opts )
    opts.update( node1=node1, node2=node2, port1=port1, port2=port2 )
    self.g.add_edge(node1, node2, key, opts )
    return key

def nodes( self, sort=True ):
    "Return nodes in graph"
    if sort:
        return self.sorted( self.g.nodes() )
    else:
        return self.g.nodes()

def isSwitch( self, n ):
    "Returns true if node is a switch."
    return self.g.node[ n ].get( 'isSwitch', False )

def switches( self, sort=True ):
    """Return switches.
       sort: sort switches alphabetically
       returns: dpids list of dpids"""
    return [ n for n in self.nodes( sort ) if self.isSwitch( n ) ]

def hosts( self, sort=True ):
    """Return hosts.
       sort: sort hosts alphabetically
       returns: list of hosts"""
    return [ n for n in self.nodes( sort ) if not self.isSwitch( n ) ]

def iterLinks( self, withKeys=False, withInfo=False ):
    """Return links (iterator)
       withKeys: return link keys
       withInfo: return link info
       returns: list of ( src, dst [,key, info ] )"""
```

```
    for _src, _dst, key, info in self.g.edges_iter( data=True, keys=True ):
        node1, node2 = info[ 'node1' ], info[ 'node2' ]
        if withKeys:
            if withInfo:
                yield( node1, node2, key, info )
            else:
                yield( node1, node2, key )
        else:
            if withInfo:
                yield( node1, node2, info )
            else:
                yield( node1, node2 )

def links( self, sort=False, withKeys=False, withInfo=False ):
    """Return links
       sort: sort links alphabetically, preserving (src, dst) order
       withKeys: return link keys
       withInfo: return link info
       returns: list of ( src, dst [,key, info ] )"""
    links = list( self.iterLinks( withKeys, withInfo ) )
    if not sort:
        return links
    tupleSize = 3 if withKeys else 2
    return sorted( links, key=( lambda l: naturalSeq( l[ :tupleSize ] ) ) )


def addPort( self, src, dst, sport=None, dport=None ):
    """Generate port mapping for new edge.
        src: source switch name
        dst: destination switch name"""
    # Initialize if necessary
    ports = self.ports
    ports.setdefault( src, {} )
    ports.setdefault( dst, {} )
```

```
    # New port: number of outlinks + base
    if sport is None:
        src_base = 1 if self.isSwitch( src ) else 0
        sport = len( ports[ src ] ) + src_base
    if dport is None:
        dst_base = 1 if self.isSwitch( dst ) else 0
        dport = len( ports[ dst ] ) + dst_base
    ports[ src ][ sport ] = ( dst, dport )
    ports[ dst ][ dport ] = ( src, sport )
    return sport, dport

def port( self, src, dst ):
    """Get port numbers.
        src: source switch name
        dst: destination switch name
        sport: optional source port (otherwise use lowest src port)
        returns: tuple (sport, dport), where
            sport = port on source switch leading to the destination switch
            dport = port on destination switch leading to the source switch
        Note that you can also look up ports using linkInfo()"""

    ports = [ ( sport, entry[ 1 ] )
              for sport, entry in self.ports[ src ].items()
              if entry[ 0 ] == dst ]
    return ports if len( ports ) != 1 else ports[ 0 ]

def _linkEntry( self, src, dst, key=None ):
    "Helper function: return link entry and key"
    entry = self.g[ src ][ dst ]
    if key is None:
        key = min( entry )
    return entry, key

def linkInfo( self, src, dst, key=None ):
```

```
        "Return link metadata dict"
        entry, key = self._linkEntry( src, dst, key )
        return entry[ key ]

    def setlinkInfo( self, src, dst, info, key=None ):
        "Set link metadata dict"
        entry, key = self._linkEntry( src, dst, key )
        entry[ key ] = info

    def nodeInfo( self, name ):
        "Return metadata (dict) for node"
        return self.g.node[ name ]

    def setNodeInfo( self, name, info ):
        "Set metadata (dict) for node"
        self.g.node[ name ] = info

    def convertTo( self, cls, data=True, keys=True ):
        """Convert to a new object of networkx.MultiGraph-like class cls
           data: include node and edge data (default True)
           keys: include edge keys as well as edge data (default True)"""
        return self.g.convertTo( cls, data=data, keys=keys )

    @staticmethod
    def sorted( items ):
        "Items sorted in natural (i.e. alphabetical) order"
        return sorted( items, key=natural )


class SingleSwitchTopo( Topo ):
    "Single switch connected to k hosts."

    def build( self, k=2, **_opts ):
        "k: number of hosts"
```

```
        self.k = k
        switch = self.addSwitch( 's1' )
        for h in irange( 1, k ):
            host = self.addHost( 'h%s' % h )
            self.addLink( host, switch )


class SingleSwitchReversedTopo( Topo ):
    """Single switch connected to k hosts, with reversed ports.
       The lowest-numbered host is connected to the highest-numbered port.
"""

    def build( self, k=2 ):
        "k: number of hosts"
        self.k = k
        switch = self.addSwitch( 's1' )
        for h in irange( 1, k ):
            host = self.addHost( 'h%s' % h )
            self.addLink( host, switch,
                          port1=0, port2=( k - h + 1 ) )


class MinimalTopo( SingleSwitchTopo ):
    "Minimal topology with two hosts and one switch"
    def build( self ):
        return SingleSwitchTopo.build( self, k=2 )

class LinearTopo( Topo ):
    "Linear topology of k switches, with n hosts per switch."

    def build( self, k=2, n=1, **_opts):
        """k: number of switches
           n: number of hosts per switch"""
        self.k = k
```

```
        self.n = n

        if n == 1:
            genHostName = lambda i, j: 'h%s' % i
        else:
            genHostName = lambda i, j: 'h%ss%d' % ( j, i )

        lastSwitch = None
        for i in irange( 1, k ):
            # Add switch
            switch = self.addSwitch( 's%s' % i )
            # Add hosts to switch
            for j in irange( 1, n ):
                host = self.addHost( genHostName( i, j ) )
                self.addLink( host, switch )
            # Connect switch to previous
            if lastSwitch:
                self.addLink( switch, lastSwitch )
            lastSwitch = switch

# pylint: enable=arguments-differ
```