University POLITEHNICA of Bucharest
Faculty of Automatic Control and Computer Science

TEZĂ DE DOCTORAT

Optimizarea Rețelelor Definite Software În Centre de Date Moderne

Optimization of Software Defined Networks In Modern Data Centers

---

Autor: Ing. Ovidiu Mihai Poncea
Coordonator științific: Prof. Dr. Ing. Florica Moldoveanu

COMISIE DOCTORAT

Președinte	**Prof. Dr. Ing. Theodor Borangiu**	**Universitatea POLITEHNICA din București**
Coordonator științific	**Prof. Dr. Ing. Florica Moldoveanu**	**Universitatea POLITEHNICA din București**
Referent	**Prof. Dr. Ing. Victor Valeriu Patriciu**	**Academia Tehnică Militară**
Referent	**Prof. PhD. Eng. Alexandru Soceanu**	**Munich University of Applied Sciences**
Referent	**Prof. Dr. Ing. Nicolae Ţăpuş**	**Universitatea POLITEHNICA din București**

București, 2016

To my family,
for their patience and constant,
unwavering, support.

ABSTRACT

When we speak about Networking in Data Centers we usually think of complex topologies of many devices and tend to ignore the unseeing part: *software* and its critical role played in moving data through Data Centers to our homes.

Today, more and more, the industry is starting to acknowledge the importance of software inside a Data Center with its capability of abstracting *everything* away from hardware by using multiple layers of *virtualization*. This is what we know as Software Defined Everything – SDx or SDE - which tries to bring into the same umbrella networking through Software Defined Networking (SDN), storage through Software Defined Storage (SDS) and the Data Center as a whole through Software Defined Data Center (SDDC). SDE promise to free software from hardware dependency.

In this research we focus on improving this *software* that moves data in a SDDC. We start with a theoretical part explaining what SDN is, why we need it in the first place and its impact on Data Centers. We also study Data Centers Architecture and design process, to determine its impact on networking. This allows us to make better decisions that provides a smooth transition to SDN.

We then present a powerful tool developed by the author to help researchers optimize SDDCs, the *Contron* SDN Controller. We later use this tool for validating a new algorithm that improves *congestion managem*ent. Finally, we present e novel routing approach that uses *source routing* for guiding traffic in a network.

***Contron, congestion management* and *source routing* represent the building blocks for an end-to-end dynamic routing solution applicable in real world deployments.**

I would like to dedicate my thesis to my wonderful family for unconditional support and patience.

I would like also to thank to my advisor, Prof. Florica Moldoveanu for her guidance and patience during the past several years and for the opportunity to find out what research is. And I gratefully acknowledge PhD. Andrei Piștirică, who convinced me to finish this journey and with whom I share parts of this research.

I also thank Victor Asavei and Mihai Claudiu Caraman who provided me with important advice and ideas.

At the end I would like to express appreciation to my beloved wife Anca who spent long days waiting, supporting and encouraging me.

CONTENTS

Abstract 3

Acknowledgments 4

List of Figures 8

List of Tables 10

1 Introduction 11

2 Software Defined Networking 13

2.1 Structure of a typical Switch in traditional networking 13

2.2 SDN Definition 15

2.3 Separation of Data from control Plane 16

2.4 Integration: Vertical vs. Horizontal 17

2.5 Approaches to SDN 18

2.6 Historical References: From Centralized Control to Distributed and back 19

2.6.1 Important innovations before Software Defined Networking birth 19

2.6.2 Important events since SDN birth 27

2.7 Requirements that led to SDN 28

2.8 Open Source Controller implementations 30

2.9 The OpenFlow protocol 33

3 Software Defined Data Centers and Cloud computing 36

3.1 Introduction 36

3.2 Data Centers classification and characteristics 37

3.3 Data center architecture 41

3.3.1 Design aspects 41

3.3.2 Modular data centers building blocks 52

3.3.3 Basic networking design: the hierarchical model 53

3.4 Storage 54

3.4.1 Evolution of storage architectures 54

3.4.2 Short introduction to Distributed Storage and its theoretical impact on networking 56

3.4.3 Ceph: Hands on performance of distributed storage and its real impact on networking .. 58

3.5 Networking 61

3.5.1 Classification of traffic patterns in Data Centers ... 61
3.5.2 Typical query and response of client server application exemplified with Openstack CLI . 64
3.5.3 Network topologies ... 69
4 Design and implementation of the Contron NS-3 OpenFlow controller ... 72
4.1 Introduction ... 72
4.2 Requirements ... 73
4.3 Architecture ... 73
4.4 Design details ... 75
4.4.1 Topology model ... 75
4.4.2 Simple Topology Discovery ... 78
4.4.3 Flow abstraction and management ... 79
4.4.4 Statistics ... 80
4.4.5 Packet Processing ... 80
4.4.6 Proxying the ARP packets ... 82
4.4.7 Connection Tracker ... 84
4.4.8 Shortest Path Routing ... 85
4.5 Case study: observing queue size and traffic profile at the receiver on congested links when routes are dynamically updated ... 86
4.6 Description of the simulation ... 87
4.7 Result analysis and conclusions ... 89
4.8 Implementation state and future work ... 90
4.9 Conclusions ... 91
5 QCN-WFQR & SDN: Maximize network throughput and reduce packet drop ... 92
5.1 QCN: Quantized Congestion Notification ... 93
5.2 Congestion Indicatives in QCN-WFQR ... 95
5.3 Quantized Congestion Notifications and SDN ... 96
5.3.1 Minimal integration model ... 96
5.3.2 Partial integration model ... 98
5.3.3 Full integration model ... 99
5.3.4 Conclusions ... 101
5.4 Case study: Load Balancing based on congestion ... 101
5.5 QCN-WFQR simulation of SDN flow migration ... 103

5.6 Conclusions 105
6 VLAN-PSSR: Port-Switching based Source Routing using VLAN tags in SDN Data Centers 107
6.1 Introduction 107
6.2 Approaches to Source Routing using OpenFlow 108
6.3 Limitations of destination/labels based routing and advantages of source routing 109
6.4 Description of the VLAN-PSSR solution 110
6.5 Functional validation in Mininet 111
6.5.1 Unicast solution 111
6.5.2 Multicast solution 113
6.6 Performance evaluation of VLAN-PSSR with Open vSwitch on Xeon servers 117
6.7 Conclusions 121
7 Conclusions and Future Work 122
7.1 Original contributions of this thesis 122
7.2 Future work 125
8 Acronyms 126
List of Publications 128
References 129

List of Figures

Figure 1: Structure of a traditional Switch; *left*: Hardware, *right*: Software ... 13
Figure 2: Separation of Data, Control & Application ... 16
Figure 3: Vertical vs. Horizontal integration – adapted from [14] ... 18
Figure 4: Before SDN ... 19
Figure 5: Stored Program Controlled Network [17] ... 20
Figure 6: TEMPEST Architecture [19] ... 22
Figure 7: ForCES – Left: Architecture, Right: Connection between a single CE and FE [20] ... 23
Figure 8: RCP Master Backup Application model ... 23
Figure 9: The VINI architecture as presented in [19]. ... 24
Figure 10: Network virtualization – Red and blue virtual nets have different topologies compared to the black physical layout [28] ... 25
Figure 11: An ethane network (printed with permission from [29]) ... 26
Figure 12: SDN Timeline ... 27
Figure 13: OpenDaylight and ONOS architectures ... 31
Figure 14: Structure of a switch that only provides OpenFlow ... 34
Figure 15: Data center classification ontology ... 40
Figure 16: Facility, Cooling and Electrical decisions ... 43
Figure 17: Design choices for servers inside a DC ... 44
Figure 18: Data Center Storage design decisions ... 46
Figure 19: Networking design decisions ... 49
Figure 20: SDDC Management Applications and their relations ... 50
Figure 21: ANSI/TIA-492 Data Center Architecture ... 52
Figure 22: The hierarchical architectural model with redundant connectivity ... 54
Figure 23: Data Center storage evolution ... 55
Figure 24: Ceph simplified architecture ... 56
Figure 25: Ceph benchmark setup ... 59
Figure 26: *Left*: Read and write benchmark (with SSD journals). *Right*: Impact of SSD journals on write throughput ... 60
Figure 27: Neutron net-list sequence chart ... 66
Figure 28: Openstack *neutron net-list* message sequence chart ... 68
Figure 29: Most used *Clos* topologies ... 69
Figure 30: Torus and Hypercube network topologies ... 70
Figure 31: Network diameter of *3D torus* and *hypercube* [75] ... 71
Figure 32: Contron's Architecture ... 74
Figure 33: A leaf-spine in a Data Center with 9 racks and 5 servers per rack ... 76
Figure 34: Topology model ... 77
Figure 35: Topology Discovery ... 78
Figure 36: Flow Abstraction ... 79
Figure 37: Flow Service ... 79
Figure 38: Statistics ... 80

Figure 39: Packet Processing 81
Figure 40: Arp Proxy 82
Figure 41: Connection Tracker 84
Figure 42: Routing 85
Figure 43 Case Study: Topology and Flow Routes 86
Figure 44 Left: Raw Data Received by the Destinations; Right: Queue Size Variation 88
Figure 45: Standard QCN mechanism 94
Figure 46: QCN and SDN – Minimal integration model 97
Figure 47: QCN and SDN – Partial integration model 99
Figure 48: QCN and SDN – Full integration model 100
Figure 49: QCN-WFQR SDN traffic balancing 102
Figure 50: QCN-WFQR Route Migration 104
Figure 51: *Left* - CP3, CP4 queues weight. *Right* - flow rates variations 104
Figure 52: VLAN-PSSR Packet Format 110
Figure 53: VLAN-PSSR unicast validation setup 112
Figure 54: Multicast between *racks* 114
Figure 55: VLAN-PSSR Multicast setup 115
Figure 56: Multicast flow entries of core switches 116
Figure 57: Benchmarking setup 118
Figure 58: Performance when pushing tags at edge on a single CPU core 120
Figure 59: Multicast performance 120

LIST OF TABLES

Table 1: State of SDN open source controller implementations (as of 02.2016) 32
Table 2: Data Center multitier model of ANSI/TIA-942..... 40
Table 3: Open Source Cloud Management Platforms..... 51
Table 4: Ceph setup drives 59
Table 5: Openstack neutron net-list message count and sizes 68
Table 6: Topology Model Notifications 77
Table 7: ARP Proxy registered notifications 83
Table 8: Connection Tracker Notifications 84
Table 9: Key simulation moments 88
Table 10: Unicast flow table of switch S_1 112
Table 11: Unicast flow table of switch C_{11} & C_{12} 113
Table 12: Flow Table of Edge switch S_1 in multicast case 115
Table 13: Flow Table of Edge switch C_{12} in multicast case 116
Table 14: Flow Table of Edge switch S_2 in multicast case 117

1 INTRODUCTION

"Keep your eyes on the stars, and your feet on the ground."

- Theodore Roosevelt

In the last 25 years we saw the rise of the Internet from simple interconnecting, packet exchange network used only by government, academics and enthusiasts to *"just another utility"*, present everywhere and used by everyone. During this time, the speed at which users connected to internet increased from a few kilobits/s to gigabits/s and the number of users reached 3.5 billion (estimate - July 1 2016 [1]) and it is estimated that internet traffic will double in just 3 years (2016-2019) [2]. This rapid increase in demand puts a constant pressure on networking infrastructure causing it to expand very fast, therefore increasing cost and complexity. Also, the growing amount of data transferred through the network needs more processing and storage facilities which, in turn, needs constant upgrades. And there is no end in sight to this exponential need for growth so any breakthrough that can help either reduce complexity or cost is welcomed.

One improvement to this is what we know as Cloud Computing. A cloud provides enormous amount of processing and storage capacity at a lower cost than previous solutions (i.e. proprietary mainframes, standalone computers or small private datacenters). Clouds brings in scale and modular designs to reduce CAPEX, automation and industrial processes to reduce OPEX and multi-tenancy to level the usage of resources thus increase income. Multitenancy represent the ability of multiple entities to share the same networking, compute and storage resources and at the same time keep their data separate from each other. Multi-tenancy gives the Cloud operator the ability to *rent* resources to third parties – they own and use the infrastructure but, at the same time, rent the unused part of it. Unused resources represent a money loss to the operator as hardware becomes obsolete quite rapidly.

Another important improvement comes from the use of *agile* processes in contrast with the traditional rigorous project plans. The agility provided by the new processes give companies the ability to rapidly adapt to change and to gather essential feedback in the early stages of a product development this way risks are minimized, products have the functionality that is mostly needed by the market and time-to-market is shorter.

Agility, which proved beneficial for the overall industry, hit one of the main limitations of IT: high cost and slow cycle of hardware development. The features provided by hardware keeps up the pace with the industry, but slow cycles are reducing time-to-market considerably as companies are stuck to what they have for a long time (6 months can be a long time in a continuously changing market). Until a faster way to create and deliver hardware will be discovered the industry needs to limit the effects of hardware slowness as much as possible. One way to do it is by creating an abstraction layer between hardware and software, a layer that would allow software to move independently from hardware. This abstracting gave birth to a new concept: Software Defined Everything – SDx or SDE - which tries to. SDE

To most of us (except researchers and top execs) Software Defined Everything is currently just a marketing buzzword but it is an objective that we have to consider and, even if it is just a buzzword, it will create new innovations that should help us overcome the gap between the software agility and hardware slowness.

In this SDE landscape Software Defined Networking takes an important place as it provides the communication infrastructure of the entire data center and beyond. The main concerns of SDN are providing simplification, lower cost, better programmability and more agility compared to existing solutions.

Even though SDN promises to be a disruptive technology, it is still far from reaching that point. There is a lot more work needed to bring it to the same scalability, performance and security that traditional networking reached through many years of slow, but solid, development.

Second chapter presents SDN, its main concepts (i.e. centralized control, separation of forwarding decisions from control and applications and network programmability), a short history and finishes with the requirements that led to it.

Chapter three focuses on impact of Software Defined Data Center (SDDC) design on networking; it proposes a new set of data center classification criteria, continues with its architecture and design. It presents in more detail how storage impacts networking – with a case study of Ceph and its benchmarked throughput. The last section presents networking itself and proposes a new classification of the traffic in the data center by observing the traffic pattern of a modern distributed application that uses *microservices*. This analysis is used for optimizing SDN networking.

Chapter four proposes a new design for a controller suitable for experimentation in the scientific community. Its architecture and functionality is similar to that of full-fledge controllers yet is simple, extensible and easy to modify. We call it *Contron* and it works with NS-3 discrete-event network simulator. A case study of a real simulation that uses Contron concludes the chapter.

In chapter five we propose QCN-WFQR, a novel approach to congestion control that fits well with SDN. We then propose one method of using QCN-WFQR in distributed and parallel file systems and one method of flows migrations achieving better load balanced networks. Contron is used to prove that our proposed algorithm provides good improvements in SDN networks.

Finally, chapter six, proposes VLAN-PSSR, an optimization of a SDDC traffic using classification at the edge and Port-Switched based Source Routing (PSSR) with VLAN stacking for specifying *unicast* and *multicast* routes. The method provides three major optimizations for data centers:

- flow table usage of core switches is dramatically reduced,
- throughput is increased by easily enabling multipath routing and,
- when compared to other source routing solutions, this method provides both a smaller overhead and support for multicast.

This new method was proved functionally in an emulated setup using Mininet and then benchmarked on Intel Xeon class servers. This showed that the method is usable without any special modifications in software nor hardware.

2 Software Defined Networking

In computer networking, there are usually two types of nodes: those that create and consume packets and those that forwards them. We will refer to first type of nodes as *hosts* or *servers* and to the 2nd as *switches*. *Hosts* exist at the edge of the network and switches at the core. Interconnection of different networks is done by edge *switches* or *routers*[1]. The network can also be seen as a graph with *hosts* as leafs, most of the switches as inner vertexes and just a few *switches*, for interconnecting with other networks, as leafs.

The first section of this chapter introduces the inner working of a switch, its logical blocks and functionality. This provides the necessary background for understanding what is the current networking norms and the difference that SDN brings to the table. The next two sections (section 2.2 and 2.3) provide the definition of SDN, and details its most important concept, the separation of data from control. In section 2.4, to better understand the steps and solutions that led to it, we take a look at its history. Then, after understanding what it means and where it comes, we present its requirements and advantages (section 2.7). The chapter is concluded by comparing the most important open source controller implementation (section 2.8) and, with a short introduction to OpenFlow, SDNs most important protocol (section 2.9).

2.1 Structure of a typical Switch in traditional networking

In Figure 1 a minimal component view of a switch, both from a hardware and software perspective is depicted. The intention is to provide a short introduction and only components relevant to our topic are presented.

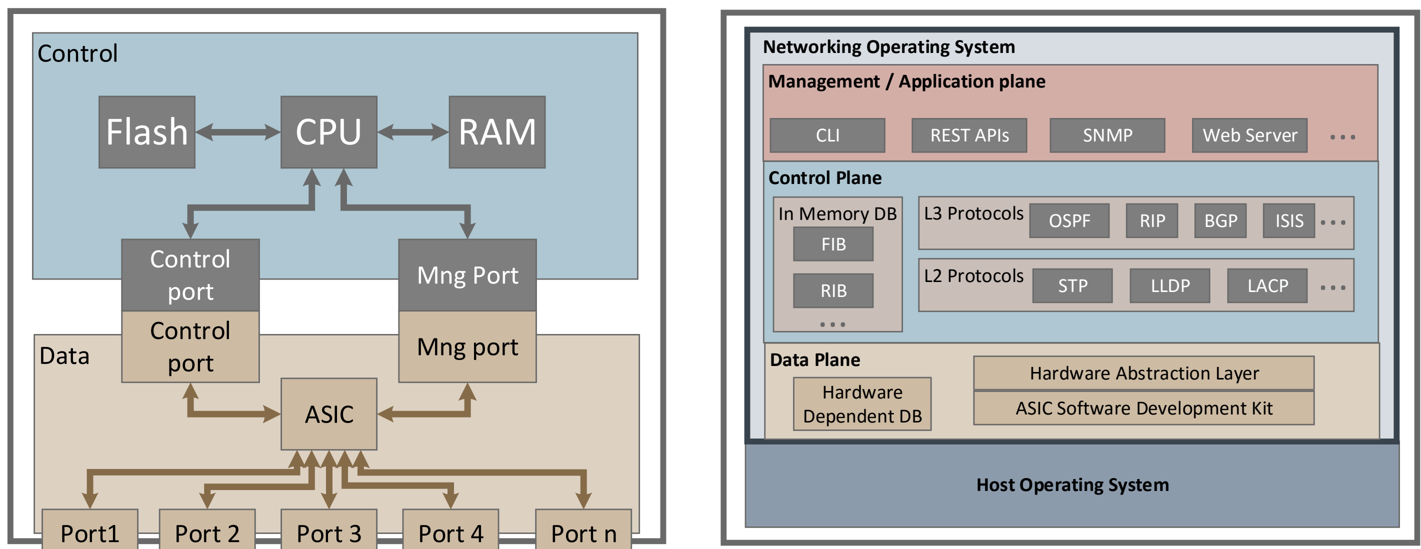


Figure 1: Structure of a traditional Switch; *left*: Hardware, *right*: Software

[1] *Routers* are also *switches* – in today's network switches perform Layer 3 operations and routers perform Layer 2 operations and from a hardware and software architectural perspective the differences are minimal.

Looking from the exterior of a switch one can see many ports, usually between 10 to 64 Ethernet ports of different speeds and types. Most of these ports are fiber optic ports but some can be UTP ports or even high speed serial connections. Some special ports are also present, a serial management console and sometimes ports that connect to the storage infrastructure (e.g. Fiber Channel) or very high speed connectors (e.g. InfiniBand).

On the inside, these switches usually have two blocks on the same or separate Printed Circuit Boards (PCBs):

1. **The Data processing block**: a set of specialized components with the role of forwarding packets at, or close to, wire rate. The most important components of this bloc is the ASIC – application-specific integrated circuit – a specialized chip that has the ability to receive, store, modify, block and forward packets.
2. **The Control block:** Has an architecture similar to that of an embedded system. It has an embedded CPU that provide the compute necessary to execute the protocol stack. This CPU uses a PowerPC, RISC, ARM or even x86 architecture (e.g. an Atom or low voltage – LV, Low Voltage, or ULV, Ultra Low Voltage – Pentium processor). It also has memory and usually a low capacity flash chip or removable storage card (e.g. SD card) for the firmware and configuration files.

The two blocks are usually connected by a management port which is actually an Ethernet port but wired through an internal bus to the control block and a control port used to program & control the ASIC.

The main reason for this block separation (i.e. Data vs. Control) is the limited processing capabilities of the CPU. With current technologies is impossible to process packets in software at speeds of 640 Gbps (64ports x 10Gbps/port). Processing packets in software is still an option but only for specific applications where packet forwarding speeds limitations are acceptable [3] [4].

Note that there are variations to this design: in some cases, for very cheap switches, there is no ASIC at all and data is processed in software, in other cases there are multiple ASIC chips in the same switch. Each chip manages a set of ports or provide some special functionality that the main ASIC is unable to provide (e.g. provide Fiber Channel support). On multiple ASIC switches the chips are interconnected, usually in a mesh network, through high speed internal busses. Other times, the data processing ASIC is replaced by NPUs – Network Programmable Units – that provide the near wire speed through internal programmable parallel pipelines – similar to those in today's GPUs [5] [6] [7].

The most important aspect to note here is that data processing is offloaded from the CPU through a fast data processing path provided by the ASIC and only protocol packets and packets that the data block is unable to process go to the CPU.

From a software perspective, a switch has a firmware composed of a Base Operating System and a protocol stack – the so called Networking Operating System. The protocol stack is implemented using programming languages that provide high performance code (e.g. C, C++ or a combination of the two) and it is, arguably, the most important intellectual property that a networking equipment vendor possesses.

Nowadays, new protocol stacks are usually build on top of a stripped down version of a Linux based operating system that initializes the hardware and provides the basic libraries. Both Juniper Junos and the new Cisco NXOS are implemented this way [8] [9].

In the past, network operating systems were implemented from scratch (e.g. Cisco IOS) or sometimes on commercial embedded OSes. For example, VxWorks has support for some Broadcom ASICs [10] and it is used in some commercial products [11], also QNX is used by Cisco in IOS XR [12]. The OS is not the critical part as it only provides the basic functionality to bootstrap the CPU and peripherals, most effort go into the development of the Networking Operating System itself.

Going into details, the Networking Operating System of a switch has three logical layers:

1. **The Data Plane** – usually a slim layer that uses the ASIC SDK provided by the producer of the chip. Is used to translate requests from/to higher layers into ASIC commands and to do some of the packet forwarding in software, if the ASIC is not available or unable to do some complex processing. The performance is very important at this layer[2].
2. **The Control Plane** – the logical layer that contains the protocol stack. This is the place where decisions are taken. Protocol packets, events and answers to requests coming from the data layer are processed by the control layer. Decisions are sent back to the data layer either as protocol packets or requests to the hardware abstraction layer. Usually no data packet processing is done at this layer.
3. **The management and application plane** – contains higher level abstractions and applications that help the network operator configure and monitor the switch by using, for example, a Networking Management System (NMC). The most important and the most used ones are: the Command Line Interface (CLI) then Simple Networking Management Protocol (SNMP) and NetConf. Some switches have a Web Interface and a custom, proprietary, REST APIs[3].

Conclusion: a typical switch is a complex device both from a hardware and software perspective and has a control, data and management/application logical planes all in the same device. This architecture reached its limits, as we will see later, and a new improved architecture is needed. Before going into the advantages of SDN over classic networks, let's see what SDN stands for.

2.2 SDN Definition

Over time, SDN has been defined in multiple ways and by multiple entities, depending on the understanding of the concept and interest of the industry. The concept started in the academic arena and,

[2] Forwarding of packets is expensive in software therefore a great effort is spent to optimize these algorithms.

[3] REpresentational State Transfer (REST) is a simple communication pattern used in distributed systems. Its roots are in web development as it uses HTTP to encapsulate its payload. Data transmitted is platform independent, easily analyzed by machines and most of the time human readable. REST purpose is to simplify communication between distributed services as all interactions are stateless. State information is kept externally, usually in a database shared between services. Therefore, without state, they may be started, restarted or upgraded on any node without impacting functionality of the system. Also, REST is highly scalable as any service from a pool of identical services may be chosen at any time to solve a task without needing complex inter-process synchronization.

originally, it defined an architecture based on OpenFlow, the concept of *flows* and the separation between Data Plane & Control Plane. At that time SDN and OpenFlow were almost synonyms.

Companies, trying to position themselves as SDN providers, extended the concept even further. They even renamed their existing solutions as SDN enabled.

The most comprehensive definition and historically correct is the one given by Open Networking Foundation (ONF):

> *"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 network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services."* [13]

To conclude, the three main characteristics of SDN are:

1. Centralized control,
2. Separation of data plane (forwarding decisions) from control and applications,
3. Programming network behavior through a set of APIs.

2.3 Separation of Data from control Plane

The most important differentiator of SDN when compared to the traditional networking model is the separation between Data plane and Control plane. We saw in section 2.1 that traditional switches have logical data, control and application/management planes but each device has its own software implementation and the planes are tightly coupled together. The difference that the new architecture brings to the table is the *physical separation* of these planes. Data plane is still present in each switch but the control plane resides somewhere else and communication is done through special protocols as opposed to the direct function calls of a traditional stack.

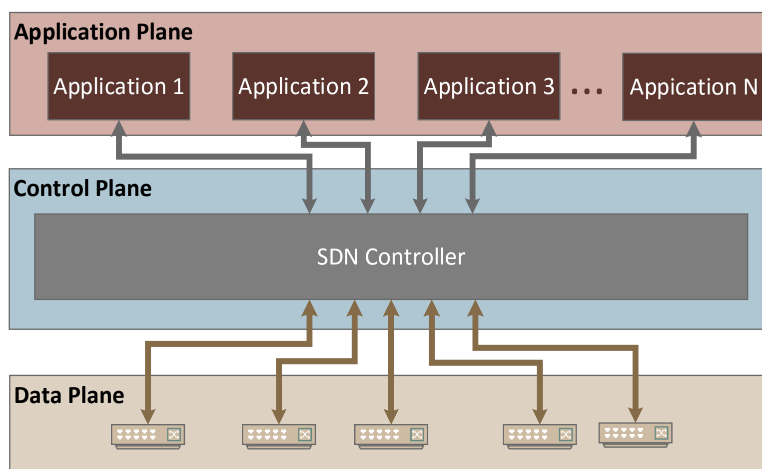


Figure 2: Separation of Data, Control & Application

The control plane is represented by the SDN Controller which can be centralized – a single entity – or distributed – multiple entities cooperating together to provide a centralized, unified and transparent view of the network. The control plane is considered to be *centralized* because the view is unique even if the controller is distributed. A distributed Controller provides advantages such as high availability and better performance at the expense of increased complexity.

The main advantages of a centralized control plane are the global view provided to the applications and the simplification of the programming model. To make changes to a network, applications no longer have to create their own views of the network by accessing each individual element, they can simply access the view of the controller through clearly defined interfaces (i.e. APIs).

The concept of a centralized control plane is not unique but the main differentiator is the open interfaces provided to the applications and the programmability these interfaces provide to the network. In classic networking, control and application are tightly connected together without any APIs, therefore in order to solve a specific problem a network operator had to either come with a, usually complex, workaround[4] or wait for standardization to catch up[5]. For example, one approach for forcing centralized changes into a classic network was to inject custom routes in existing IGP or EGP protocols, therefore many implementation of BGP route injection exists.

2.4 Integration: Vertical vs. Horizontal

A traditional switch is a complex device build as a monolith by a single vendor (see Ch. 2.1). The new architecture proposes a different approach: provide an ecosystem of interchangeable components that are simpler and cheaper. This would give a user the power of choice and reduce the chances of locking him to a vendor specific ecosystem.

The layers (planes) of a traditional switch are considered to be vertically integrated as everything is specialized and cannot be interchanged between vendors. In case of SDN, we can use any switch with any controller with any application as long as the interfaces respect the specification of the open interfaces between the layers (see Figure 3).

Regarding monetization – how one makes money – traditionally vendors mostly sell hardware, the cost of software development is usually included in the price of the hardware. The new architecture proposes a different model of monetization: at each layer. At the data layer cheaper hardware with simple software, at Control Plane the Controller is sold as a software application and the applications themselves are also independently sold – much like what is currently seen in the PC industry: the hardware (i.e. the pc, laptop) the operating system (e.g. Windows) and the applications are sold independently.

A horizontally integrated industry should also foster innovation. Again, the analogy with the PC industry can be used – IBM closed mainframes vs. Personal Computer based data centers.

---

[4] Or pay the switch manufacturer for a solution

[5] Protocols that solves almost every conceivable problem are defined and many of them are standardized, but each new protocol added increases the code size and complexity of software and sometimes even of hardware.

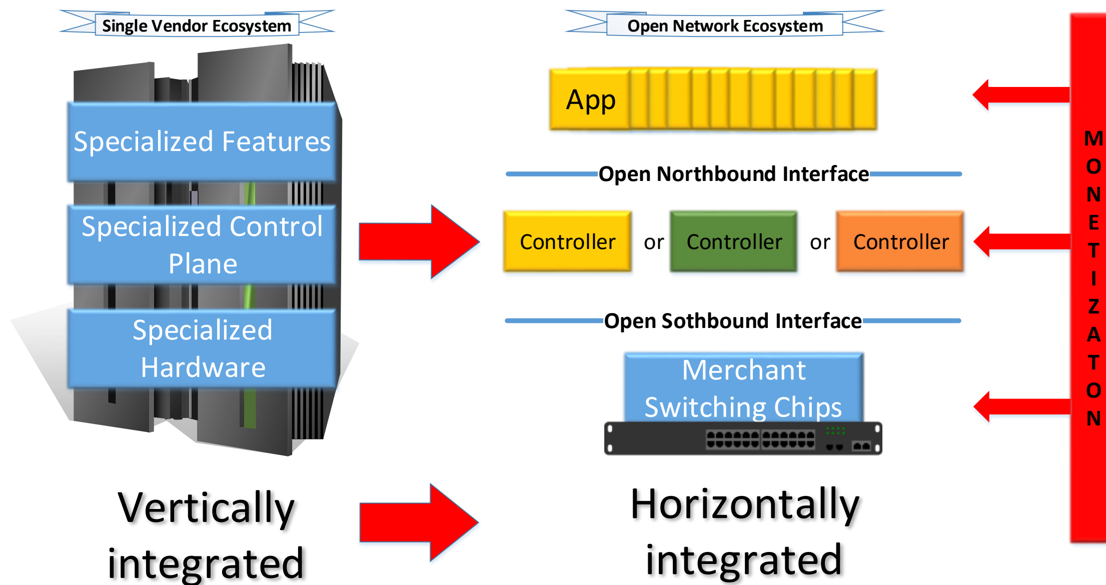


Figure 3: Vertical vs. Horizontal integration – adapted from [14]

2.5 APPROACHES TO SDN

The separation of data, control & application planes (see Ch. 2.3) can be subtle in current SDN implementations as many solution providers focus on solving the problems (see Ch. 2.7) rather than following the standard architecture. These solutions are still considered SDNs as they embrace some, but not all, of the key aspects of the standard definition and they differ from standard networking.

There are four approaches to SDN (based on [15]):

1. **Open SDN** – Emerged from educational institutions, this is the classical, ideal, SDN that we are focusing on this paper. Is based on Open APIs for *northbound interfaces* – the interface between control and data plane – and *southbound interfaces* – the interface between control and application. Controllers are interchangeable, applications are also interchangeable (or at least easy to port to different controllers) and network devices from different vendors can coexist in the same network as long as they provide the *southbound interfaces* that the controller understands;
2. **API-based SDN's** – These solutions provide a public northbound API for network programmability but the control and data planes are proprietary. These are mostly complete solutions and the users treats them as black boxes. Usually the northbound APIs are not even accessible by the user.
3. **Overlay-based SDN** – This is a type of SDN that can be installed on top of an existing IP infrastructure and is done in software, at the edge, in the virtual switch that is running on a server. There is usually no new hardware required for this. This is actually the most deployed solution today as it is leveraging existing data centers (i.e. you don't need a brand new data center to implement SDN). It overcomes the limitations of the network (e.g. VLAN or MAC table exhaustion) by using tunneling technologies (e.g. VXLAN). Network programmability is still present through the Open APIs for the controller northbound interface. In this case the southbound interface is not flow based (i.e. OpenFlow is usually not present).

4. **Closed SDN** – These solutions provide no APIs for programmability but they are, arguably, considered SDN solutions because they solve the same problems and because they, more or less, separate control from data plane (e.g. VMware NSX solution).

2.6 Historical References: From Centralized Control to Distributed and back

The starting point of SDN is considered to be 2008, when the creators Nick McKeown and Martin Casado published [16], a paper presenting OpenFlow, a protocol that is the catalyst of the new architecture.

The history of the main concepts in SDN, control and data plane separation, and similar attempts stretches far back, to 1981. This timeline is important as we can see a trend: networks started centralized, then decentralized, and now the tendency is to go back to a centralized model, therefore some concepts and innovations that were abandoned in the past may come back, in a new form, in SDN.

Bellow we present two timelines, one with the history before SDN (Figure 4) and another one with important events after the new architecture was acknowledged by industry (Figure 12).

2.6.1 Important innovations before Software Defined Networking birth

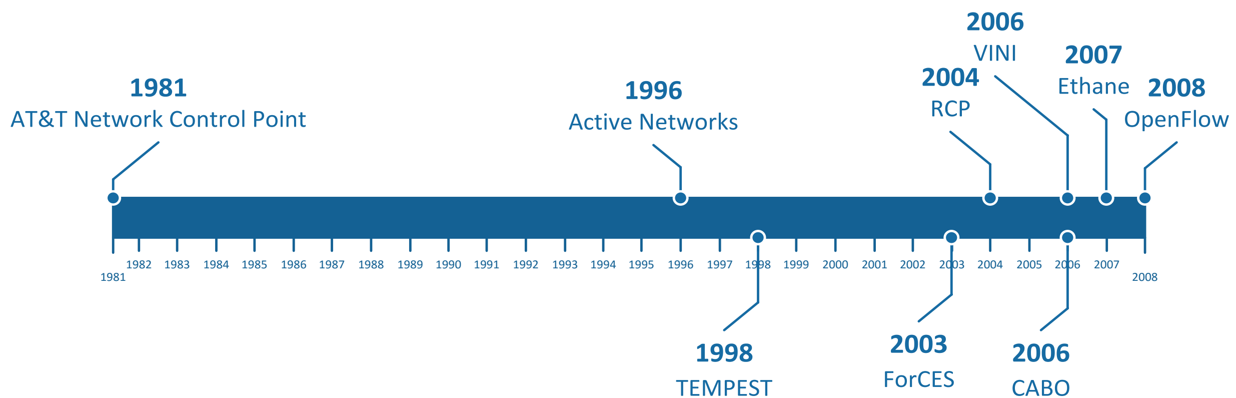


Figure 4: Before SDN

2.6.1.1 1981: AT&T Network Control Point

The idea of a centralized control dates back to 1981 when AT&T was building its Control Center for the analog telephone industry called Network Control Point [17]. This central point was programming each element of the telephone network through an off-band channel and getting feedback from those elements (e.g. node status, call duration or user position in the network). One of the most interesting features of the system was user mobility – a user terminal could connect anywhere in the network, register itself with the control point, and could be called using a unique phone number. A very interesting feature of that time. Another aspect that is similar to current SDN are the different applications built on top of it (e.g. mobility).

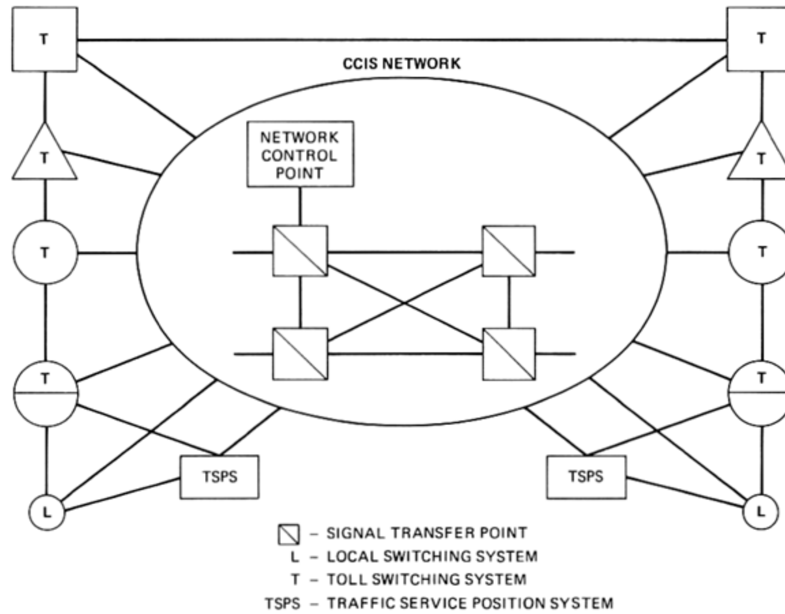


Figure 5: Stored Program Controlled Network [17]

After the AT&T network and the down of the mainframe era, in time, the current distributed networking model came into existence and started expanding due to its simplicity (at that time), resilience to errors, reduced price and relative openness as routers and switches were available from multiple vendors and were able to interoperate due to the standard protocols implemented. The main issue was that, in order to satisfy more complex requirements, the technology also became complex and expensive.

2.6.1.2 1996: Active Networks

In order to satisfy some of the requirements of the networks and to better use the network resources, in 1996 a new concept was born: Active Networks. The research community is still interested in this area but to a lesser extent than in the past.

The main idea of Active Networks is that packets themselves can contain code that is executed on the network nodes traversed by the packet. In traditional network packets are just forwarded, content is used for making decisions yet the payload itself is not executed. Basically, this solution allows users to inject customized programs into the nodes of the networks.

There are two types of active networks:

1. **Programmable switches** – applications are sent to switches through a separate channel (e.g. binaries, simple scripts);
2. **As capsules –** incorporate the logic in the packet. Some of the packets contain simple scripts with executable code intended for switches to run.

There are multiple advantages [18]:

1. Capsules provide a mean of implementing fine grained application-specific functions at strategic points within the network;

2. Protocols can be adaptive – they can adapt themselves based on output of the custom program;
3. The infrastructure can be *customized* by the user through the abstractions provided.

Some applications include: multicast caching (e.g. YouTube cache), routing decisions, firewall at the edge, web proxies.

Why did Active Networking not work? There are multiple reasons for this, the main reason is that the end user ***is the programmer, not the network operator*** – the network operator operates (owns) the infrastructure and will not (yet) open his network to this type of access (see the CABO proposal below) – security being just one of the concerns of the net op.

Other reasons:

- Ahead of its time
- Security concerns: flow isolation, hacking, DoS – just imagine what kind of damage can be done by allowing third party code running on your devices if a security flow is exploited;
- Performance issues – node resources are limited;
- Multi-tenant? Yes, but risky, one entity capsules/applications can impact other entities traffic/process, solving this results in increased complexity;
- Complex – see above problems that need complex solutions.

2.6.1.3 1998: TEMPEST separates Control from Data in ATM networks

The TEMPEST framework provides multi-tenancy which, in this case, represents the possibility of multiple entities to independently manage the – or part of the – same infrastructure by adding a new element called *The Divider*. This element practically separates the data from the control and provide independent *Partitions* to the users. A partition is a *slice* of the network resources that is visible to the user. The slices are managed by the *Partition Control*. This is very similar to SDN: The Divider is equivalent to the Controller, Switchlets are applications, ATM switches are the data plane and there are open interfaces between each component similar to the Northbound and Southbound interfaces in SDN.

The intent of TEMPEST is to ease the innovation in this area: "*As anyone who can obtain a virtual network will effectively be a network operator we hope to see an increase in the creativity that can be brought to bear upon the problem of network control."* [19]

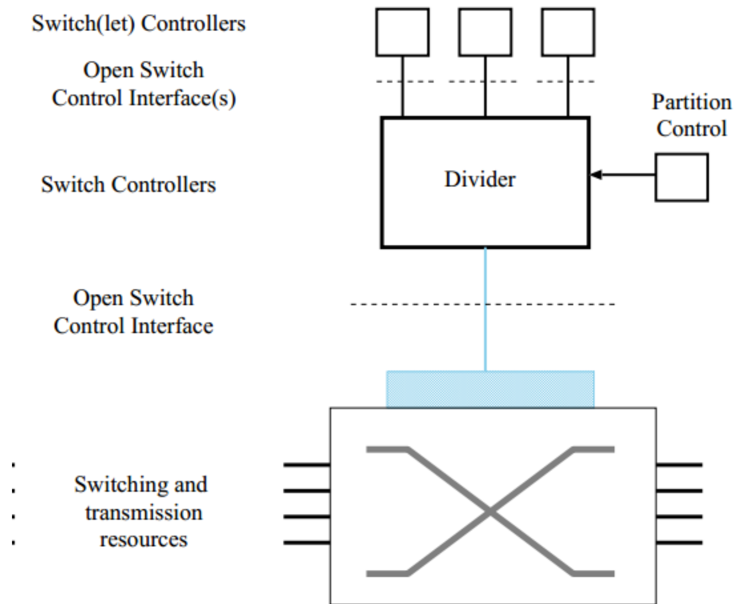


Figure 6: TEMPEST Architecture [19]

2.6.1.4 2004: ForCES - Forwarding and Control Element Separation

Another approach that focused on the separation between Data & Control plane dates back to 2004 and was standardized by RFC3746 [20]. It defines an architectural framework and associated protocols to standardize information exchange between the two planes. It is similar to OpenFlow.

At that time it was envisioned that standard control protocols such as OSPF, RIP, and BGP are on top of an abstraction layer – the ForCES Interface (Figure 7). The Network Element (NE) is composed of a Control Element (CE) and a Forwarding Element (FE). The CE implements the control plane and is usually an application running on a COTS server independent of the forwarding element.

The Network element is an atomic block that provides high availability – two CEs are managing two FEs so that if any of the four devices would fail the system will continue to function. The FEs data interfaces are usually connected through link aggregated ports to the servers bellow (i.e. a server has a LAG with two ports, one to each FE).

For its time, the architecture was complex, the solved problems were few and, the most important aspect, it required hardware modifications to accommodate it.

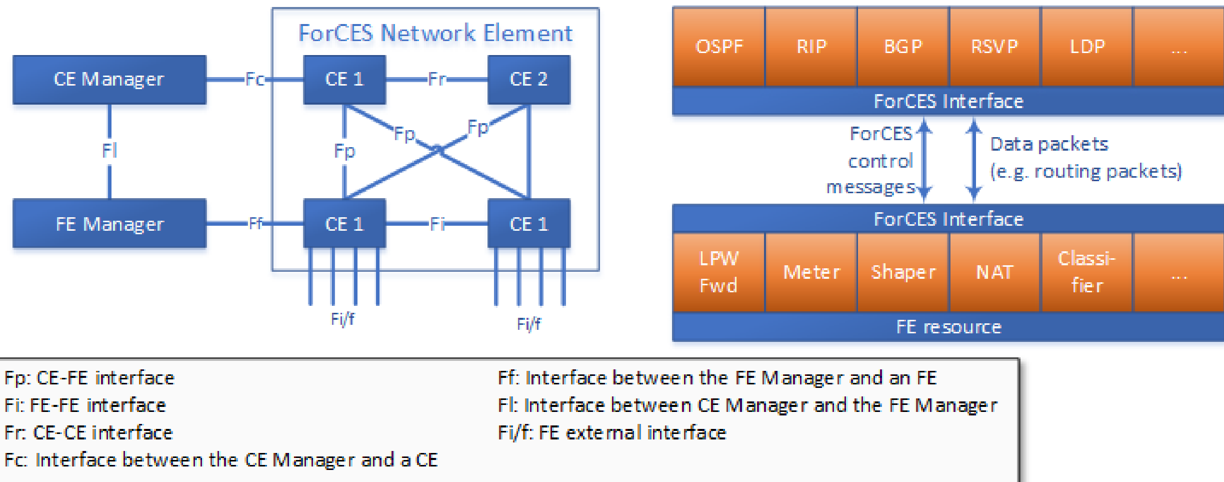


Figure 7: ForCES – Left: Architecture, Right: Connection between a single CE and FE [20]

2.6.1.5 2004: Routing Control Platform

The Routing Control Platform (RCP) uses BGP as the Control Plane and provides an application that controls traffic by injecting routed from a central application into BGP.

The use of BGP is limiting the options that the Control Plane could offer but it is useful for automatic traffic redirection and providing DDoS protection.

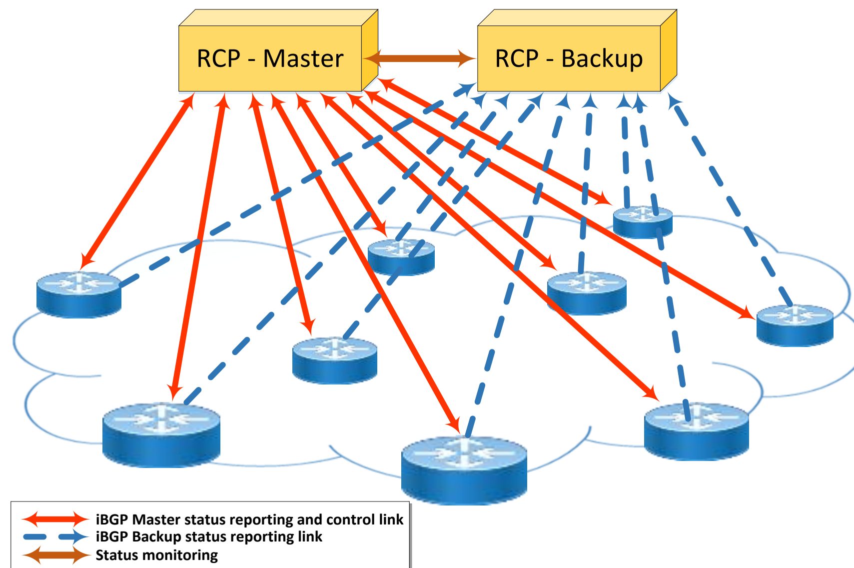


Figure 8: RCP Master Backup Application model

The Routing Control Platform provides high availability by implementing the Master/Backup model as seen in Figure 8. Both master and backup applications connect to the routers through iBGP but only the master is sending route table modifications. The backup keeps only an updated status of the network by reading the state from the routers without performing modifications. When the master fails or a switch from

master to backup is manually initiated the backup application already has the updated state and can easily take over as the new master. The master and backup applications monitor each other to detect failures.

The RCP provides a narrow solution for a limited set of issues that a datacenter operator had to solve yet the solution proved efficient and demonstrated that a centralized Master/Backup model – or Active/Standby as it is better known as – can successfully control a computer network. This is a strong case in supporting the centralized model of control and a good POC - Proof of Concept.

2.6.1.6 2006: Virtual Network Infrastructure (VINI)

Emulates multiple network devices and virtual infrastructures on the same physical infrastructure thus allowing easy development and testing of new protocols [21]. The environment provided by VINI is close to a real life deployment. The architectures provides network virtualization and separation of control from data plane.

The VINI architecture uses XORP [22], Linux container based virtualization (i.e. the VServers implementation [23]) and *uml_switch* application [24]. The architecture is executed on top of the PlanetLab network research infrastructure [25]. The PlanetLab is composed of networked multiuser Linux servers distributed all over the globe that use container based virtualization to separate users from each other.

See below a short description of most important components in Figure 9:

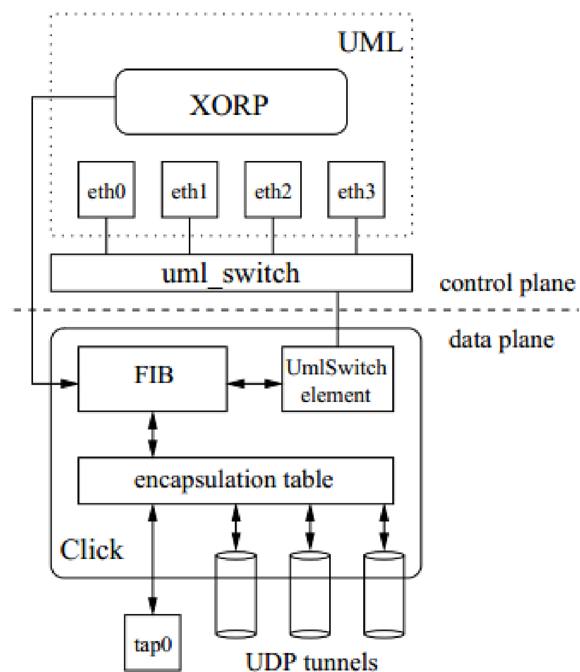


Figure 9: The VINI architecture as presented in [19].

- **XORP** is an application that runs on Linux and manages the routing tables of the OS through the layer 3 protocols it provides. The result is that packets are forwarded by the Linux kernel based on the rules added by XORP, this is similar to the forwarding behavior of a physical switch or router yet is done in software. The advantage is that, being an open source project, new protocols are easy to implement and test but, given the software nature of it, packet processing is slow.
- **Linux containers** provide operating system level virtualization between applications running on the same server [26]. It is used to run multiple XORP instances on the same machine thus simulating many nodes on the same server. Each container has its own routing table and ports.
- **Uml_switch** is a user space application that forwards packets between user mode applications with no connection to the host's network. It acts either as a switch or hub and uses UNIX domain sockets as ports. It is simple when compared to Open vSwitch. In VINI the uml_switch is used to connect Linux containers that run XORP instances to each other into a virtual network.

- **Click** calls itself a *modular router* [27] and it is a toolkit for creating configurable routers by using combinations of packet switching units called *elements*. By chaining toghether multiple *elements* a spcific functionality can be obtained. This flexibility is useful for experimentation and testing.

The open source components are connected through custom components that are abstracting the PlanetLab infrastructure for e.g. the UDP channels in Figure 9 correspond to virtual interfaces, when a real interface goes down, clicks simulates this by adding a filter on the tunnel that blocks all traffic.

As a conclusion, the virtualized network configuration is hard codded in configuration files, is missing the abstractions of SDN and performance is limited to software processing but it provided a good insight into the possibilities of network virtualization. Another important aspect to note here is the separation of data plane from the control plane.

2.6.1.7 2006: CABO, Concurrent Architectures are Better than One

CABO proposes to separate infrastructure from services providers. Currently the same entity provides both the infrastructure and the services to end users. Separation of the two roles is possible if a new architecture is deployed. In this architecture each service provider has access to a virtual infrastructure overlaid on top of the physical infrastructure. CABO important from an SDN perspective because, as we will see later, virtualization is one of its key applications.

In the current architecture the infrastructure providers maintain routers, links, data centers and other physical infrastructure and service providers offer services such as layer 3 VPNs, performance SLAs etc. to end users. The CABO proposes that the infrastructure providers should maintain physical infrastructure needed to build networks and service providers should lease “slices” of the physical infrastructure from one or more providers.

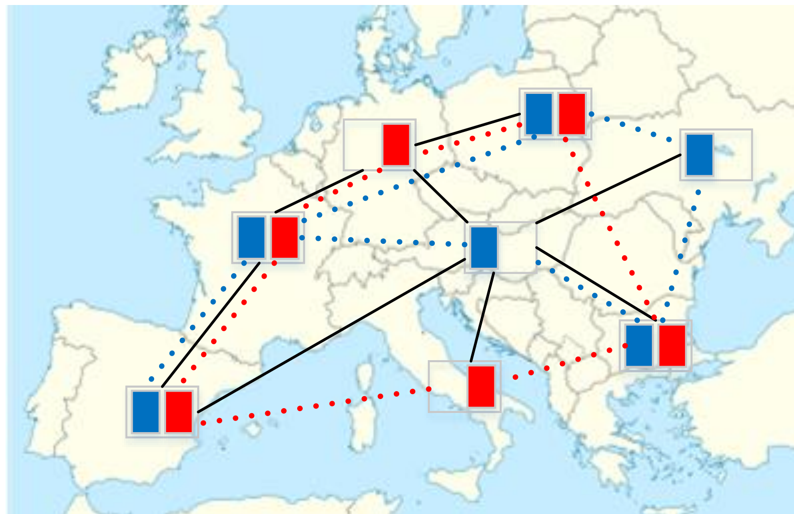


Figure 10: Network virtualization – Red and blue virtual nets have different topologies compared to the black physical layout [28]

2.6.1.8 2007: Ethane, the hardware abstraction

This project was implemented at Stanford University by the same group of researchers that later invented OpenFlow and provide valuable insights into the separation of data from control plane and the use of *policies* for managing the network instead of using classic configurations based on IPs, MACs, VLANs etc.

The Ethane architecture consists of a centralized controller (i.e. the “Domain Controller”) plus very simple hardware and software switches. The centralized controller holds the policy rules, global view of the network and authentication data and, based on this info sets, programs the low level flows into the switches.

A policy represents the *intention* of the network operator, what he really wants the network to behave like. For examples, if the same network has VoIP phones and PCs connected we may not want the two groups to communicate with each other, therefore adding a policy that specifies this is enough; the controller takes care of the low level details. In classical networks the intention is converted by the admin through his experience into configuration options for the devices in the network – for our example the admin may choose to separate the two groups by VLANs but this is an end result, and not the intention.

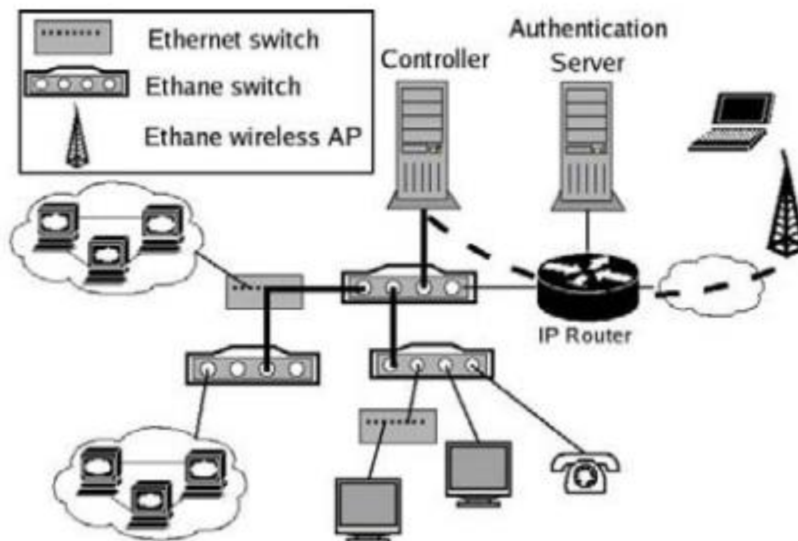


Figure 11: An ethane network (printed with permission from [29])

Ethane which was built around 3 fundamental concepts [29]:

1. *“The network should be governed by policies declared over high-level names”* – as network management is easier through polices rather than low level flows,
2. *“Policy should determine the path that packets follow”* – users should define the policy and flows should result from the policy,
3. *“The network should enforce a strong binding between a packet and its origin”* – origin of a packet should be determined by *user* and/or *host* rather than a source address which can easily change during transport.

The main disadvantages of this approach, and ones that hindered adoption, are:

- The architecture required hardware modifications - and no one wants to implement a new switch from zero. The Stanford team used NetFpga modules to implement a minimal switch in Verilog, therefore the industry would have to take a similar path and implement their own ASICs.
- No clear and open specification of the APIs between the controller and the switches. Ethane is a closed ecosystem, it does not provide the openness that the classic solutions provide – you can't just replace a switch with another switch.

Nevertheless the lessons learned were invaluable and constituted the base for SDN.

2.6.1.9 2008: OpenFlow – Enabling Innovation in Campus Networks

In a 2008 paper [16] Nick McKeown et al. described the OpenFlow protocol. This is considered the birth of SDN.

2.6.2 Important events since SDN birth

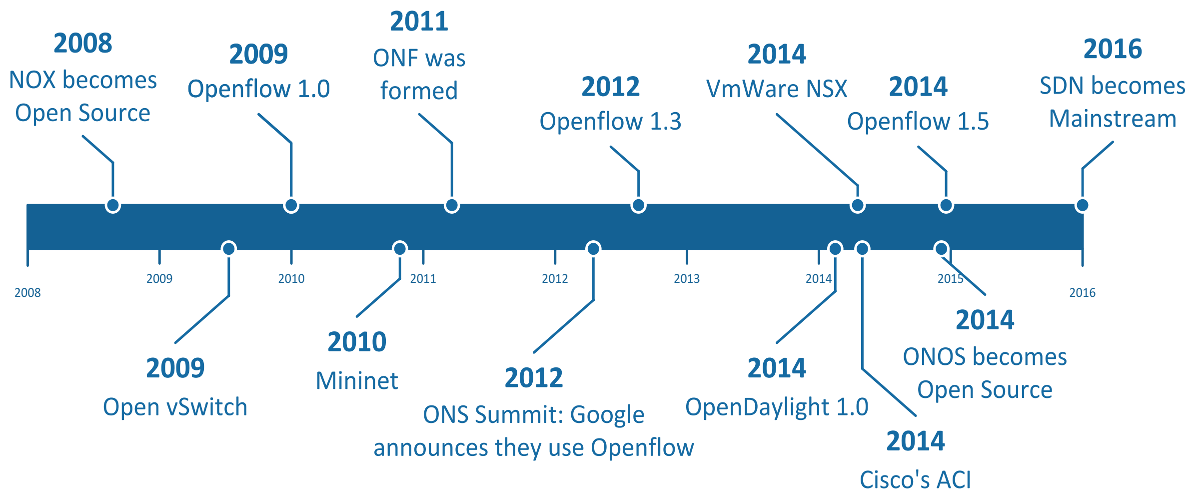


Figure 12: SDN Timeline

OpenFlow marked the beginning of SDN, yet the specification was in draft for over two years and evolved rapidly with major modification from release to release[6] . The 1.0 final version was released at the end of 2009 and it represents an important baseline as most controllers implemented it and it is well support them to this day.

Two more important events happened in this time:

1. the NOX controller [30], which is the first controller developed for SDN was Open Sourced, this is important as it provided a functional architecture, algorithms and feature set that could be used by other controller implementations, therefore this pushed forward the development of many other controllers.
2. Open vSwitch (OVS) was released [31]. OVS is software switch that runs on Linux and provides an almost complete implementation of OpenFlow (from 1.0 to 1.5). The main advantage is that, with its release, users could easily create emulated networks with many virtual switches by executing

[6] OpenFlow 0.98 is not compatible with 1.0 for example.

multiple instances of OVS on the same machine and connecting them together in a virtual topology. This provided easy deployments of new topologies and, being software, provided a rapid prototyping platform.

In 2010 Mininet exploited the advantages provided by OVS (i.e. OpenFlow support) and operating-system-level-virtualization (i.e. Linux kernel process and networking namespaces) and provided users with the ability to create realistic virtual networks on a single machine [32].

Mininet is written in Python and provides a CLI that can be used to easily create complex topologies of virtual switches and instantiate virtual hosts connected to these topologies. The user would then just login into each virtual hosts and be able to transmit or receive data just as it would happen from a real host.

In 2011 Open Network Foundation (ONF) was formed[7] as a nonprofit organization aiming at improving networking through SDN and on leading the ongoing development of OpenFlow [33].

For almost two years the domain continued to evolve but without major events as SDN was considered mostly a research topic. In 2012 Google announced that their entire network is SDN based and uses OpenFlow [34] [35]. This announcement triggered a massive interest from the industry, big names announce their commitment in bringing products to the market. Two years later the first mature products entered the market: VMware NSX and Cisco's ACI platforms.

Important events are also the release of two complex open source controllers: OpenDaylight [36] and ONOS [37]. OpenDaylight is a community effort managed by *The Linux Foundation* [38] backed by many prominent companies in the industry[8]. The development evolved rapidly, from nothing to 2.5 Million in 2.5 years (as of 02 2016) [39], and ONOS which is an alternative project created by ON.Lab, a non-profit organization founded by the initiators of SDN from Stanford university [40]. ONOS focuses on providing a controller for telecom operators.

2016 is expected to be the year when SDN is adopted worldwide and when we will see standardizations of the northbound interfaces[9] [41].

2.7 Requirements that led to SDN

Many of current data centers are complex sometimes with more than 1.000 switches, 50.000 servers and over a million VMs [42]. Configuration of this many networking devices is an issue as, in standard networking, it is usually distributed – each device has its own configuration file leading to thousands of files. These files can be stored in a central repository (on a file system or in versioned repository[10]) and custom applications that take these configurations and apply them to devices can be built but this is still a hard and error prone process. Just think of configuration management complexity and what damage could be done from configuration errors in any of these files!

[7] ONF founding members: Deutsche Telekom, Facebook, Google, Microsoft, Verizon, and Yahoo!
[8] OpenDaylight founding members are: Arista Networks, Big Switch Networks, Brocade, Cisco, Citrix, Ericsson, HP, IBM, Juniper Networks, Microsoft, NEC, Nuage Networks, PLUMgrid, Red Hat and VMware
[9] The southbound interfaces are standardized and well adopted by the market (e.g. OpenFlow)
[10] Code Versioning Systems such as Git [61], Subversion [62] or Mercurial [63] are used.

Distributed configurations are hard to maintain with many files and a complex configuration management process, they are error prone due to:

- **Complex validation** – configuration files need a running setup (at least a virtual setup or a minimal hardware setup) and a set of good tests.
- **Unpredictable** – there is no direct traceability between the intention of the admin (e.g. I want to block external access to the VoIP phones of the company) and the configuration option (e.g. "create VLAN 20")
- **Error prone** – the admin has to rely on his experience, but human errors can and do occur.

The conclusion: it takes a long time to deploy and validate a configuration change. Anecdotally, in 2014 it took Cisco 5 days to do a deployment, IBM was aiming for a 1 day and they were looking at SDN as it could do the same deployment in minutes.

Some of the requirements of the current data center:

- **Network automation –** Manual workflows should be automated. In classic networking the *intention* is expressed a set of requirements coming from different actors (e.g. engineers, management, clients, security audits) using different communication channels (e.g. verbal, emails, presentations, spreadsheets) and are converted into network configurations by network administrators who then test and deploy the new changes. This is slow and involves a lot of manual steps. We could say that the *intention* is *compiled* by humans into *low level* configuration options. In an automated workflow the *intention* will be written as a *policy* by the net admin or directly by the client or security engineer and the system will take care of compiling it into low level options. The policies are defined in an easy and clean way so that, when an admin looks at them, he can see what where the initial intentions.
- **Network virtualization –** Multiple tenants (e.g. departments of the same company, external clients) may need to use the same infrastructure isolated from each other or different project phases may share the same infrastructure (e.g. production, development and testing may have different slices of the same network or different versions of the same application may coexist);
- **Better security –** Traffic should be isolated between tenants and, for a domain, security should be enforced at the edge to reduce network load (e.g. DoS attacks should be blocked at the edge).
- **Reduce complexity –** Data centers are complex, any reduction is welcomed.
- **Reduce costs.**
- **Reduce deployment time.**

Advantages of SDN over conventional networking:

- Easier to:
    - **Evolve** – deployments are easy, interfaces are open, custom apps and protocols can easy be implemented. Network virtualization can provide rapid set-up and tear down of test beds;
    - **Maintain** – operations are automated, no need for complex scripting. Replacing equipment is also easier as long as it implements the standard interfaces;

    - **Can easily be integrated with Virtual Machines** – VM migrations are much easier due to the global view of the controller and the open interfaces (i.e. the Cloud Management Platform can specify the desired operation to the SDN Controller). Also, load balancing is much easier as there is no need for a dedicated load balancer – a good controller can do it correctly;
    - **Provide network virtualization** – overlays can easily be configured network wide in a controller. Overlays provide the necessary traffic isolation and security requirements;
    - **Multiple networks can independently coexists** – production, testing, development, research use the same physical setup;
    - **Do on the fly upgrades** – upgrades are faster and safer;
    - **Reduce data congestion** – this is a complex issue in both conventional and SDN networks but the new architecture provides better tools to manage it.
- **Management plane can:**
    - Coordinate behavior among different types of services (e.g. router + firewall + IDS coordinated by the same application)
    - Integrated configuration and management for different types of devices – not only switches and routers but servers, IDS, firewalls and many virtual functions.
- **Apply conventional Compute Science approaches to (old) networking problems:**
    - Programming: orchestration done using custom scripting or high level languages (e.g. Python)
    - Software engineering applied to networking: agility (i.e. iterations), phases, disciplines (e.g. analysis, implementation, testing, deployment)
- **Having a centralized control plane**:
    - Easy to reason about specific logic and algorithms
    - Easy to infer behavior
- **Reduce cost** – switches are simpler and easier to manufacture. Also, if they implement the standard interfaces, switches can be bought from different vendors and interchanged easily. A short calculation: if we have 1000 switches in a Data Center traditionally 1000 switches * 5000$ = 5M$ for an SDN solution for the same number of units 1000 switches * 1000$ = 1M$ **this results in an economy of 4M$/Data center!**

2.8 OPEN SOURCE CONTROLLER IMPLEMENTATIONS

In this section we are briefly going to present two most important open source controllers: OpenDaylight [36] & ONOS [37]. OpenDaylight is targeted at developers and has a vast feature set and ONOS, targeted at operators, which is more deployment ready. These two are currently under active development and have become a reference to the industry. Both of them are implemented in Java. A third one, Ryu, is written in Python and used for research and for validating custom OpenFlow implementation as it has an extensive test suite for it. Floodlight & Pyretic are still under development but interest is low. Many other controller implementation exists but their development has halted. A detailed status, as of February 2016, of most important Open Source is presented bellow in Table 1.

OpenDaylight was initiated in 2013 and as an industry supported project coordinated by *The Linux Foundation* [38] to provide an open source SDN framework. In 2014 the first version was released. It is a complex Java project implemented using the OSGi Framework (Apache Karaf [43]).

OpenDaylight provides not only the basic set of services that usually come with a controller but a multitude of *microservices* which the user may select to install as needed. Services are bundled together and dependency checks are made when starting them up (a component will load and initialize all dependent components on startup). It has two interfaces, one external (the *northbound* interface) using REST APIs and another one internally for java OSGi *microservices* using internal fast communication mechanisms.

From an architectural perspective (Figure 13-1), it has a service abstraction layer (SAL) on top of different *southbound* protocol drivers (e.g. SAL connects to an OpenFlow driver), on top of SAL lies a set of base networking services and many other services that the user can activate. On top of them is a REST API that applications can access. The base architecture is straight forward but, because of its massive code size (2.5 mil lines of code) and many teams working independently on different components understanding the details is a complex task.

ONOS**,** the other important controller, is implemented in Java by Open Networking Lab (ON.Lab) as collaborative project supervised by The Linux Foundation. ON.Lab was created as a non-profit organization founded by SDN inventors and leaders from Stanford University and UC Barkley as opposed to OpenDaylight industry supported project.

ONOS is Carrier grade – which means that high availability and scalability are fundamental aspect (OpenDaylight works better as a standalone controller). It is designed as a distributed controller with all peers running the same identical code.

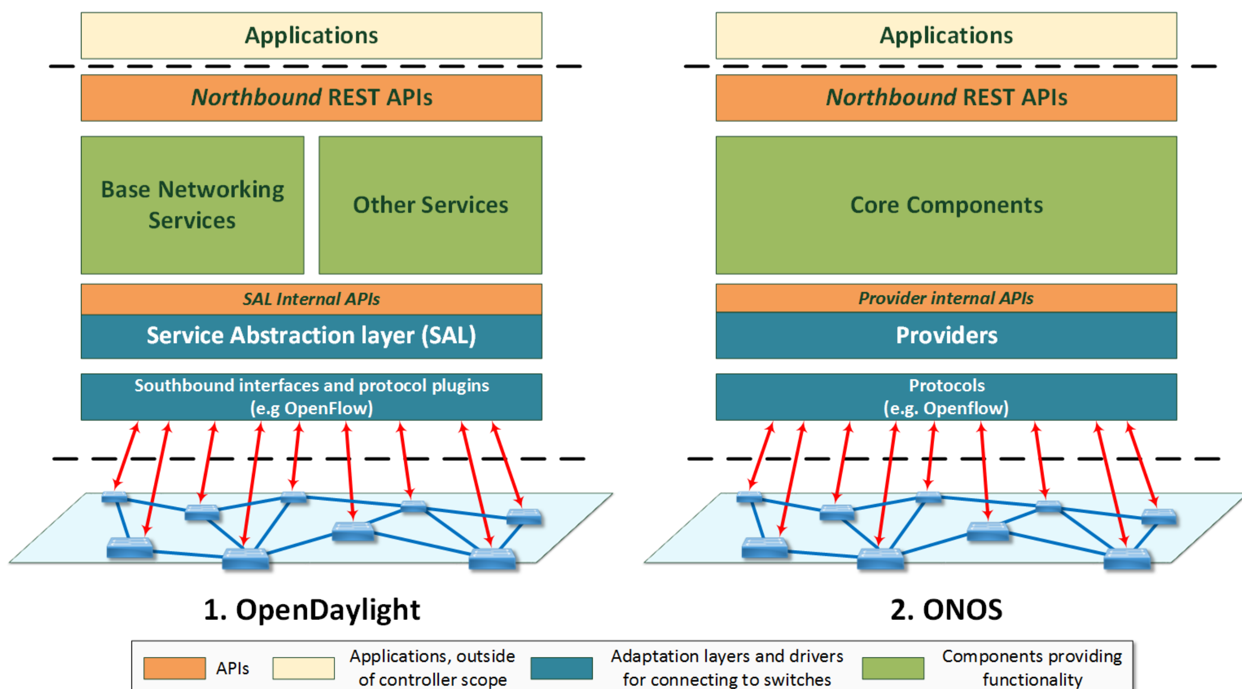


Figure 13: OpenDaylight and ONOS architectures

From an architectural perspective (Figure 13-1), ONOS has horizontal multilevel stacked architecture, protocols have providers and are abstracted by *southbound* provider API on top of which lies the *Core* which exports *northbound* APIs. Vertically it provides services (also known as subsystems). ONOS defines a service as a vertical slice through the tiers of the software stack that provides a unit of functionality.

ONOS uses the same OSGi Framework as OpenDaylight yet their vision is different both from their approach to SDN (see 2.5) and from targeted use cases. ONOS is following the Open SDN concepts (it is after all created by the inventors of SDN) while OpenDaylight is a mix of Open SDN with Overlay-based SDN with a focus on creating a set of interoperable APIs. Both solutions provide *frameworks* that can be used by companies to build their own proprietary controller implementations tailored for their unique use cases.

From our experience with them the main difference we observed is that OpenDaylight is more a set of APIs and internal Java interfaces submitted by different networking manufacturers trying to leverage their own custom offering (OpenDaylight works well on top of custom solutions) while ONOS is more practical as it target telecom operators use cases. Also, for developers who are familiar with it, OpenDaylight provides a broader set of services.

Table 1: State of SDN open source controller implementations (as of 02.2016)

Name	Source	Programming language(s)	LOC	Activity/month (commits & contributions)	Notes
NOX	ICSI	C++ 64%, Python 27%, Other 9%	81,980	Inactive	Last commit in September 2010
POX	ICSI	Python 96%, Other 4%	20,928	Inactive	Last commit in July 2012
Ryu	NTT Comm.	Python 87%, Erlang 9%, Other 4%	115,000	33 commits, 8 contributors	Long lived project with low but steady activity
Beacon	Stanford University	Java			Beacon became Floodlight so inactivity is expected
Floodlight	Big Switch Networking	Java 98%, Other 2%	98,000	20 commits, 7 contributors	Old controller with low activity, development has stopped
Pyretic	University (multiple)	Python 80%, Erlang 9%, Other 11%	188,000	40 commits, 2 cont	Very low activity

Name	Source	Programming language(s)	LOC	Activity/month (commits & contributions)	Notes
Trema	NEC	Ruby, C	60,000	Inactive	2 commits in last 6 months
OpenMUL	OpenMUL Foundation	C 86%, Shell script 7%, Other 7%	213,741	Inactive	Last commit September 2015
OpenDaylight	OpenDaylight foundation	Java 55%, C++ 16%, JavaScript 9%, Other 20%	2.6 Mil	1000 commits 120 cont	Very high activity, code base increases very fast!
ONOS	ON.Lab	Java 80%, JavaScript 14%, Other 6%	450,000	500 commits, 50 cont	Second fastest growing

2.9 THE OPENFLOW PROTOCOL

OpenFlow provides internal access to low level forwarding tables of switches for external applications. It is defined by Open Networking Foundation (ONF) in [44] and the latest version is 1.5.1 released in 26 March 2015.

It is a binary protocol that works over TCP or UDP [11] with security provided by the underlying communication stack (e.g. by TLS tunnels). A session consists of three phases:

1. **Initiation** – Connection is usually established by the switch as they know how to connect to controllers[12]. After connection is established (e.g. over TCP with TLS tunnel) the protocol performs a short version negotiation. Then, if negotiation is successful, a controller requests a list of features that the switch supports (some features are optional) and it also sends an initial configuration. Each switch uniquely identifies itself using a *data-path id* (64 bits unsigned integer).
2. **Operation** – flows are added, removed or updated, packets are sent from controller to switch and vice-versa.
3. **Monitoring** – the controller requests statistics from a switch or the switch sends statistics periodically to controller. Communication channel between switch and controller is permanent and monitored using *echo* messages.

A switch that only implements OpenFlow is much simpler than a full NOS implementation of a traditional switch (Figure 14 bellow versus Figure 1). OpenFlow protocol only uses the Hardware Abstraction Layer and ASIC's development kit. From the old management plane in Figure 1 usually only the CLI remains as it is needed for initial configuration (e.g. configuring a secure management channel to the Controller or setting the Datapath Id).

[11] In theory it is independent of transport layer but most implementations use TCP or UDP.
[12] IP address of controller is configured through an external channel – e.g. serial CLI console.

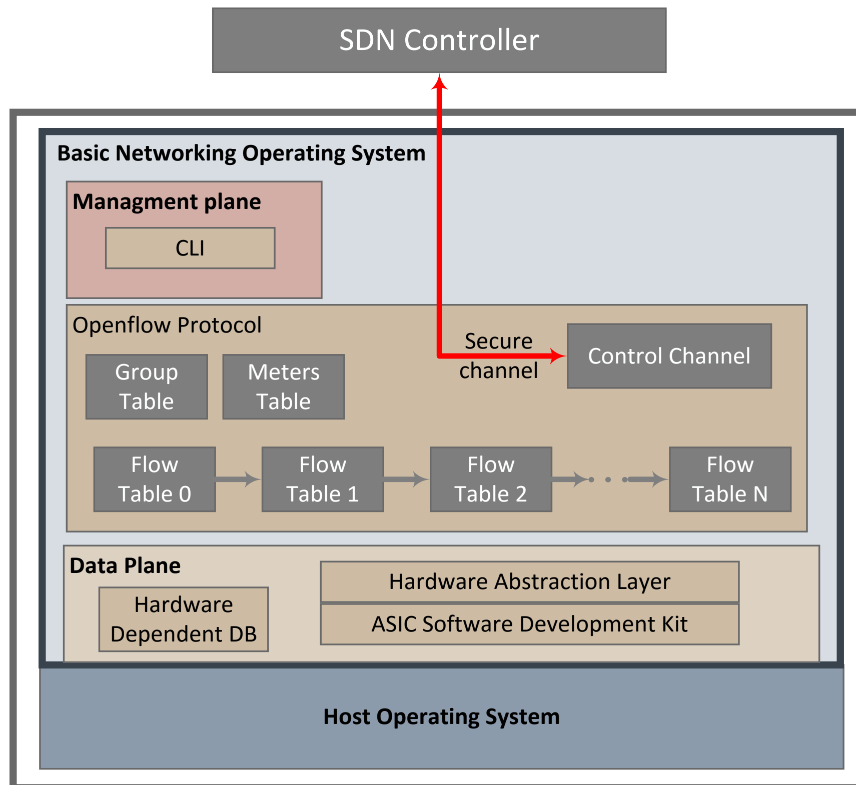


Figure 14: Structure of a switch that only provides OpenFlow

OpenFlow, as the name suggests, manages *datapath flows*, therefore, the most important concepts are the *flow entry* and *flow table.* According to standard [44]:

> "[a flow entry is] *an element in a flow table used to match and process packets. It contains a set of* ***match fields*** *for matching packets,* ***a priority*** *for matching precedence,* ***a set of counters*** *to track packets, and* ***a set of instructions*** [or actions] *to apply* [to a packet]"

Where a *flow table* is a stage of the pipeline that contains flow entries. Multiple flow tables are (usually) supported by switches. A packet entering the pipeline is processed in the first table, matched against the flows in that table in order of their priority until first match is hit. If that packet matched contains a *goto another table* instruction then processing continues on that table[13].

To better understand the concepts of flows, actions and matches we added bellow four examples of flows from our experiments with VLAN-PSSR in Ch. 6. The binary representation is converted in a human readable format:

```
1    table=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
2    table=0, priority=60000,dl_dst=ff:ff:ff:ff:ff:ff actions=FLOOD
3    table=0, priority=32768,vlan_tci=0x0800/0x0800 actions=goto_table:10
4    table=0, priority=0 actions=CONTROLLER:65535
```

[13] Else if no match is found and it is not the last table in the pipeline then processing continues with the next table otherwise packet is dropped.

All four flows are in table 0, the first one has the highest priority (`65535`) and the last one has the lowest priority (`0`):

1. Fist flow matches on MAC `01:80:c2:00:00:0e` and on `dl_type=0x88cc` representing the source address and EtherType of the packets. EtherType is a two-octet field in an Ethernet header. Is used to indicate which protocol is encapsulated in the payload of the packet. 0x88cc value is used by Link Layer Discovery Protocol (LLDP) a protocol useful in topology discovery. The action is to forward the packets matching this mac end EtherType to the controller (`actions=CONTROLLER:65535`). LLDP is a management protocol so it is sent to the controller for further processing.
2. The second flow matches on broadcast addresses (`ff:ff:ff:ff:ff:ff`) and its action is to forward these packets to all switch ports (`actions=FLOOD`) except the one it was received on.
3. The third flow sends any VLAN tagged packet to another pipeline table in the pipeline for further processing (`goto_table:10`).
4. Last flow has the lowest possible priority (0), therefore it is the last one to be checked and its action is to forward to controller, same as flow 1. The difference is that this flow actually has no match defined, this means that, given its low priority, all packets that do not match any other rule in the table are sent to the controller for further processing.

A flow *match* can be configured for one or more header fields in the packet, such as: all Ethernet and IP headers, VLAN ID, VLAN priority, MPLS tag, TCP/UDP source and destination ports etc. Some fields provide *wildcard mask* matching. Widcard mask is a set of bits that indicates which parts of a header field (e.g. IP & Ethernet addresses) are available for examination.

Multiple actions can also be added, such as sending (*outputting*) the packet to a list of ports, to another table, flood on all ports, send back to input port or send to controller. Other actions can be to decrease TTL, to *push* a new VLAN or MPLS tag or to modify an existing tag. Pushing and popping VLANS and sending packets to different tables is used extensively by our VLAN-PSSR proposal in Ch. 6.

First OpenFlow version (1.0) was limited in functionality yet provided the advantage that it did not need hardware modifications. Later versions increased in complexity and necessitate hardware modifications or costly software processing to work around hardware limitations. This led to slower adoption. The newest version to date is 1.5.1, yet reference is considered to be 1.3 as it is well supported by vendors.

3 SOFTWARE DEFINED DATA CENTERS AND CLOUD COMPUTING

3.1 INTRODUCTION

In a Software Defined Data Center (SDDC) compute, networking and storage resources are abstracted. Compute and networking are virtualized and storage is partitioned and distributed. In an SDDC workflows can be migrated from a machine to another one without interruptions[14]. Differences in hardware and configuration complexity is hidden from the users. The resources are managed by a Cloud Management Platform and presented to the user in an abstracted manner. Access to those resources is done through a CLI, GUI (usually Web based) and/or API (usually REST).

Cloud computing, in simplified terms, means storing and accessing data and applications over the Internet instead of storing it or running applications on your own computer. A more formal definition is given by NIST in [45]:

> "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*"

NIST also states that the cloud model is composed of three service models: IaaS, PaaS, SaaS (see Ch. 3.1 - *Service model*), four deployment models: Private, Community, Public, Hybrid (see Ch. 3.1 - *Deployment*) and it has five essential characteristics:

1. **On-demand self-service** – resources can be provisioned by a user (tenant) automatically without user interaction;
2. **Broad network access** – can be accessed by standardized mechanisms that do not impose a restriction on the device type (PC, laptop, phone, tablet etc.) or operating system (Linux, Windows etc.) as long as they adhere to standards;
3. **Resource pooling** – the access is multi-tenant, users can dynamically reserve resources from a pool without caring about the strict location of the reservations;
4. **Rapid elasticity** – Resources can be rapidly provisioned and released;
5. **Measured service** – resource usage is monitored and measured at all times (e.g. number of VMs, CPU usage, IOPS operations, size of storage used) and reported to both the user and Cloud administrator.

By looking at these characteristics we see that SDDCs can easily conform to the cloud NIST definition, service model, deployment models and characteristics. In case of small private clouds, broad network access and measured service may be basic as only admins need to keep track usage statistics. Also, private clouds do not need to provide broad device support if the private network is used in controlled, internal, environments. Multi-tenancy is supported but may only be used as a way to just isolate between admin and user role.

[14] Or with minor interruption if live migration of VMs is not supported.

In the following sections, after exploring characteristics and classification of Data Center, the thesis presents its architecture then storage and networking. The focus is on identifying attributes of different technologies impacting networking itself. Compute is treated as part of architecture as its impact on networking is less than that of storage.

3.2 Data Centers classification and characteristics

Data centers are facilities used by enterprises to house servers, storage and networking for the needs of a single or multiple entities (e.g. companies, departments). This typically involves storing, processing and serving large amounts of data to clients with applications implementing client server architectures.

The requirements of a data center are diverse and depends greatly on the classification of the DC. So, by just understanding what *kind* of DC we are analyzing we can estimate its requirements and its high level design and implementation restrictions.

This thesis proposes the following criteria for classifying data centers:

1. **Size & availability** – this is how ANSI/TIA-942 classifies data centers. This categorization represents a simplification of multiple characteristics: Power and cooling redundancy, component redundancy, fault tolerance, power outage sustainability interval (see Table 2 below);
2. **Purpose** – type of applications it is serving:
   - *Application oriented* - built specially for one or more applications. E.g. Facebook or Google search engine data centers are purposely build for a single application, artificial intelligence – IBM Watson etc. These data centers are well optimized for their purpose and do not easily allow the execution of different apps. They usually serve a single tenant, or may be shared by multiple tenants with shared concerns, such as needed by grid computing.
   - *Generic* – these data centers focus is on providing support for *any* application, they are not as optimized as the application oriented class yet are more versatile. Clouds are built on top of this type of DCs. Also, they are multi-tenant.
3. **Orchestration model** [15] – defined by resource management and sharing model. This model influences many other characteristics such as network topology, storage and deployment:
   - *Clustering Middleware Management* – a cluster is a set of similar or identical compute nodes (i.e. computers) that are connected together through a high speed network with the objective of finalizing specific tasks in a distributed manner. A *clustering middleware* is a software layer that manages cluster nodes and allows users to treat the cluster as one large cohesive computing unit. A cluster has 2 definitions, only the first one uses a *clustering middleware*:
     i. The nodes are managed by a Cluster Controller which schedules the workload between them. Multi-tenancy is possible by providing a task queuing & scheduling

[15] Even though both *supercomputers* and *mainframes* provide a type of computing used in datacenters they do not play any role in SDN. Also, a *grid* can't be considered as part of a datacenter.

mechanism in the controller[16]. A middleware layer manages the scheduling and task distributions, applications need to be written specially for this.

ii. The term cluster is used to describe parts of the datacenter that perform a specific function for example: the management cluster, the storage cluster, a web server cluster behind a load balancer etc. This does not describe the orchestration model

- *Cloud Management Platform* – A specially designed application that manages cloud resources and offers them to clients. The resources are provided through virtualization or partitioning (for more details see section 3.3.1.4),
- *Application specific* – data centers built for a specific application or a set of applications, for example Facebook has its own datacenters tailored for a single application, and same can be said for some of Google's datacenters.

4. **Service model** – Determine the level of service provided by the data center:
    1. *IaaS: Infrastructure as a service* – provides virtual machine, containers or bare metal servers to the cloud user. The user is able to install its own operating system and the responsibility for managing the virtual environments is completely his.
    2. *PaaS: Platform as a service* – provides different execution environments to the user. The user does not have access to the underlying operating system nor infrastructure. The user provides a *package* in the format required by the PaaS, usually by uploading it through a web interface and the PaaS takes care of executing it. Sometimes PaaS is running on top of an IaaS.
    3. *SaaS: Software as a service* – the user usually can login in a software application provided by the cloud SaaS operator. The software is licensed on a subscription basis and is typically accessed through a Web Interface. SaaS may run on top of a PaaS or IaaS.
5. **Deployment** – Who owns versus who uses a DC:
    - *Private* – cloud is owned by the same entity that uses it;
    - *Community* – multiple entities with shared interest use the same cloud;
    - *Public* – multi-tenant, external entities may rent resources from these clouds. It has a single owner (i.e. DC operator), the other entities are external;
    - *Hybrid* – private clouds that extends into the public cloud. Either on service based – some services are externalized (e.g. storage) or based on scalability premises (i.e. extends in public cloud when demand is high).
6. **Network topology** – For clouds the most used deployed topologies are those in the Clos category: Fat-tree and leaf-spine.
7. **Storage model** – Storage model evolved in time from local to SAN and distributed. Each of the models have their advantages and disadvantages and where preferred at different moments in time. Five models are more important (for details on the first four see 3.4.1):
    1. *Independent storage network* – Storage arrays are connected to servers on a dedicated storage network. In this model, networking and storage traffic is isolated and two independent networks are needed. Storage have their own cabling switching devices.

[16] Not to be confuses with SDN controller.

2. *Partially converged storage and data networks* – Storage arrays are connected to network switches and storage traffic is tunneled aver data network.
3. *Converged storage over data network* – Special storage servers provide data for compute nodes. All storage traffic happens over data network. Usually this is done using a distributed storage solution (see 3.4.2).
4. *Hyper-converged* – Servers that provide processing also provide storage but, instead of using it only locally and wasting the unused space, it is shared between all servers using a distributed storage solution.
5. *Local storage* – storage is accessible locally on each server, no networking is needed.

8. **Compute model** – categorizes DCs based on how the compute resources are managed. Used models are:
   1. *Bare metal* – servers do not use any type of virtualization, they are managed either manually (i.e. an operator installs the OS when needed by going into the DC, connecting to the server with a portable display, keyboard and mouse) or automatically through a remote management console that connects to the server through a KVM switch (Keyboard Video Mouse switch) or through a special interface that server manufacturers provide (iLO for Dell and iDRAC for HP). Modern data centers provide a GUI for managing servers and the possibility to remotely connect image disks (DVD virtual images in .iso format) and boot from them. Some data center providers even support automatic OS installation of bare metal servers.
   2. *Hardware virtualization (Virtual machines***)** – Servers use hardware virtualization to partition compute and storage resources. These servers also have a virtual switch that provides connection between the physical Ethernet interfaces and the VMs virtual interfaces. These switches use software processing and are slower than their hardware counterparts but easily upgradable. Most of these virtual switches support the latest version of OpenFlow therefore they can easily become part of SDN networks.
   3. *Operating-system-level virtualization (Containers or jails)* – The operating system allows the existence of multiple isolated user-space execution environments. The isolation is done though kernel mechanisms and usually provides 3 categories of isolation:
      1. Process isolation – processes belonging to the same environment see and communicate with each other yet are not aware of processes in other environments.
      2. Networking isolation – this is done through virtual ports usually connected to a virtual switch. Each environment only has access to a dedicated set of virtual ports.
      3. Storage isolation – the host file system is not visible nor the storage spaces of the other environments.

Table 2: Data Center multitier model of ANSI/TIA-942

Tier	Size	Availability	Downtime	Other
1	Typical small business	99,671%	28.8 hours	o Single path of power and cooling o No redundant components
2	Medium-size business	99,749%	22 hours	o Single path of power and cooling o Some redundancy in power and cooling systems
3	Large company	99,982%	1.6 hours	o Multiple power and cooling paths o Fault tolerant (N+1) o Able to sustain 72-hour power outage
4	Multi-million dollar company	99,995%	0.04 hours	o Two independent utility paths o Fully redundant (2N+1) o Able to sustain 96-hour power outage

From an SDN perspective, the classification of Data Centers has a great impact on the requirements imposed on a controller mainly from a performance, availability and security perspectives:

- **Performance** – the main impact is made by the topology and by its size. Big data centers need performant and scalable controllers.
- **Availability** – some controller don't provide the required level of HA. Also, a high HA may impact performance.
- **Security** – Multi-tenant data-centers (e.g. Clouds) have a stricter security requirements than single tenant ones.

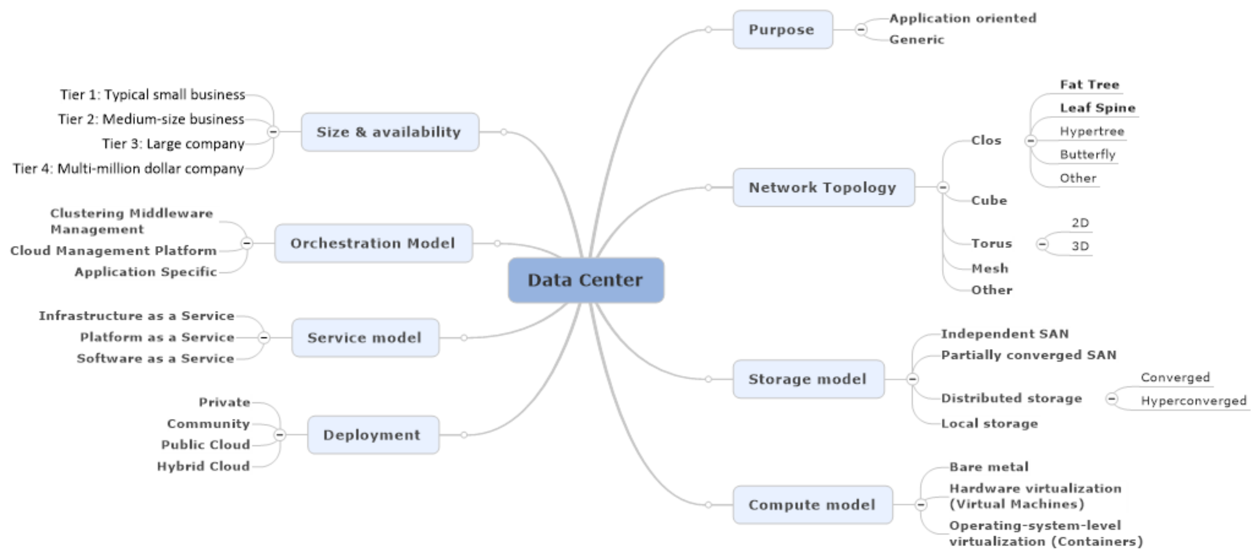


Figure 15: Data center classification ontology

3.3 DATA CENTER ARCHITECTURE

Data centers are diverse, both in size, performance and technological level. The size vary from very small ones with just a few cabinets stored in a computer room to warehouse size with hundreds of cabinets stored on multiple building floors and rooms that have special needs for power, cooling or security. The technologies used in data centers are also diverse as most of them are designed to operate for 10-15 years. During this time maintenance need to be supported for old equipment while also leaving room and preparing in advance for upgrades and capacity increase [46] [47].

During its lifetime, a data-center undergoes a few major upgrades, therefore the initial design should be able to accommodate these changes. To simplify this, standards have been created both for specifying the base architecture and connectivity between equipment. Most notably here is the TIA-942 [48] (with 2013 update TIA-942-A) and the Open Compute project [49].

Inside a data center replacing old hardware or purchasing new capacity is easily done if the architecture supports the new device interfaces, power, cooling and management needs. This can is possible by careful design and planning.

Arguably, the most important aspect of a data center is designing and building the communication network – both data, management and storage networks – as they interconnect all of the individual boxes into a cohesive fabric. This design is important as it affects performance, scalability, security and availability and its deployment is expensive and time consuming.

To optimize networking in SDN data centers it is important to understand the impact that different design aspects, data center layouts and hierarchical design have on the networking itself. Such constrains limit choices of topology, equipment, affects the cost, performance tradeoff and imposes certain limitations on the routing algorithms themselves.

In the following sections the thesis presents an overview of important aspects of datacenter design and the categories of choices that a designer has to consider. The next section presents the main building blocks of a DC as defined by ANSI/TIA-492 and continues with a short presentation of one of the hierarchical networking architecture.

3.3.1 Design aspects

Data Center design is a complex process that involves knowledge from multiple domains. A team of experts in each domain designs and then monitor its construction.

The design itself starts from a set of requirements and finalizes with the commissioning of the DC. The process itself is complex and varies from company to company and from deployment to deployment but usually is based on a waterfall process with a design and planning phase at the beginning, followed by construction (execution), commissioning and operation.

Requirements impose design constrains on the choices in each domain. Then these choices, because of interdependency limit the choices in other domains.

This thesis identified the following seven domains as the main drivers of DC design:

1. **Facility** – Represents the building where the equipment is installed. Decisions in this domain impact all of the other aspects, especially if it is already built;
2. **Cooling** – Type of cooling and its capacity. This has a smaller impact on the networking and compute, especially if standard, air cooled data centers are built but it can greatly impact them if liquid cooling is chosen;
3. **Electrical** – Servers and network equipment use tremendous amounts of power so a good design of this domain is important and may limit the available options – especially the compute choice and density;
4. **Networking** – Decisions are based on the requirements (mainly capacity, availability and latency), on the constrains imposed by the other domains and on available technologies;
5. **Compute** – model, architecture, equipment type;
6. **Storage** – is impacted mainly by its capacity and storage model;
7. **Data center management and operations** – applications that manage the data center and depends mostly on the level of automation desired.

The one that mostly interest us is *networking*, yet it is essential to have a view on all of the other domains as they all impact it.

Facility, cooling and *electrical* design impose little restrictions on networking. Probably the most important is the size of the facility. Bigger facilities need more complex design.

Cooling and electrical impact on networking is linked to the constant need for reducing power consumption. This is an important requirement, especially for large Data Centers. This provides another advantage for SDN as devices need less CPU processing in switches therefore need less powerful CPUs at switch level. Regarding cooling, custom solutions, such as liquid cooling, require customization of the networking equipment which is costly.

Key decision points for facility, cooling and electrical design, without further details, are listed in Figure 16.

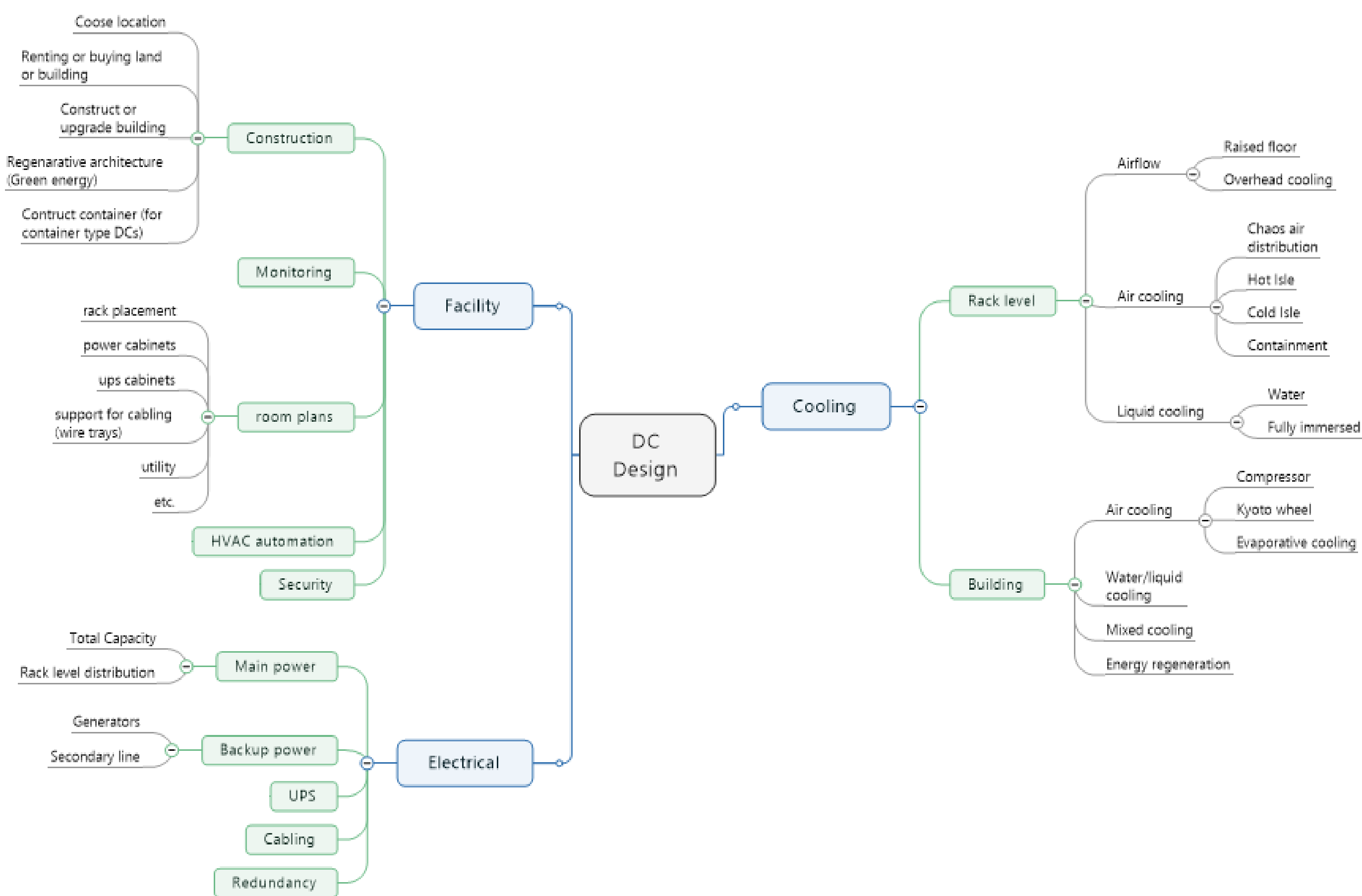


Figure 16: Facility, Cooling and Electrical decisions

3.3.1.1 Compute design decisions

Hardware choices mainly depend on the specific needs of the applications that run on them. The hardware specs include server size, density (blade, rack – 1U, 2U), *processor architecture* and *special devices* installed in servers.

The processor architecture used inside DC is usually x86 or Power PC. Older data centers are a mix of the two but new data centers today prefer x86. ARM servers are new to the market and, even though some vendors sell them the adoption rate is low [50] [51] [52] [53].

Other than the CPU, servers include *special devices* such as GPUs, for parallel processing, and Cryptographic accelerators.

To increase processing speed inside VMs, servers with *special devices* provide support for *PCI passthrough*, a technology that allows a Cloud Management Platform to allocate a PCI device directly to a VM by bypassing completely the virtualization layer of the *host* server. The operating system inside the *guest* VM will access this device directly (i.e. its PCI registers) without host translation of API calls. Therefore, these devices will operate at the same speed inside a VM as it would do in the host operating system. The disadvantage is security as the host operating system will not be able to intercept any call to this devices. For example, in multi-tenant systems, passing an Ethernet device will give the tenant VM

full access to it, sometimes even promiscuous mode, therefore special measures need to be taken. GPUs, NICs and Crypto accelerator modules are good candidates for *PCI passthrough*.

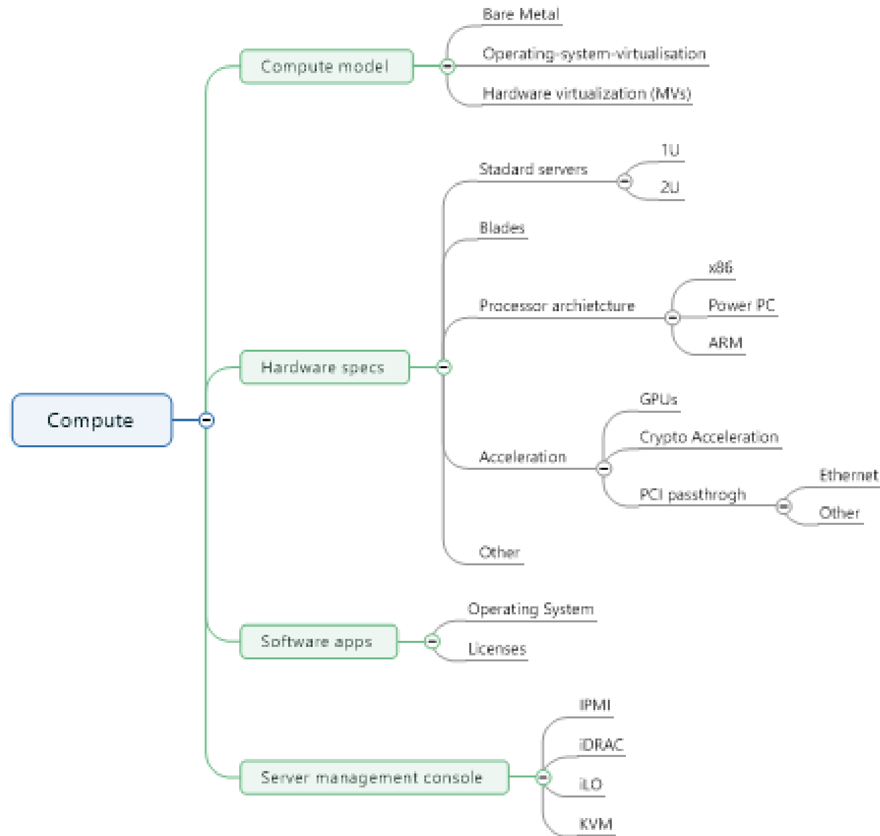


Figure 17: Design choices for servers inside a DC

Another important aspect is accessing the management console of the compute node[17]. Many technologies to control a server exists, some of them are standardized such as IPMI, Intel ME and serial (RS232) port some are vendor specific, such as iLO (from HP [54]) or iDRAC (from Dell [55]) or even custom boxes, known as KVM switch, that take the VGA and USB keyboard signals and transmit them over Ethernet to a centralized console management system.

From a software perspective, the operating system of the host needs to be chosen or, in case of hardware virtualization the hypervisor has to be decided. Also, libraries, applications and other tools that run on top of the compute to provide the desired level of service (for clouds, nodes connect to the Cloud Management Platform and provide at least IaaS if not PaaS or even up to SaaS). For more info about Cloud Management see section 3.3.1.4.

Compute node imposes the following constrains on the networking decisions:

[17] Remotely accessing keyboard, video and mouse same way we are accessing a local computer

1. Bare metal nodes[18] do not have a virtual switch – the edge switch is actually a HW switch, fast but limited in functionality and upgradability.
2. Virtualization solutions have a virtual switch – which is very slow compared to hardware solutions, but easily upgradable;
3. Compute nodes in application specific data centers may or may not have a vSwitch;
4. High performance computing impose certain restrictions on the topology and performance – Cube and hypercube topologies are sometimes used;
5. PCI passthrough of Ethernet devices (NICs) must be handled with care to avoid possible security issues;

3.3.1.2 Storage design decisions

New data centers are build using a converged design with distributed storage solutions. Older DCs with no or partial convergence still need to be supported, therefore FC, FCoE and iSCSI technologies are still going to be maintained and even deployed for years to come.

When designing the storage, two main aspects are taken into consideration: how much distribution is needed and what *storage model* will be used (see section 3.2).

Servers usually have a small amount of local storage for booting up (*bootstrapping*) the Operating System[19] and to keep local data such as management or local applications. After the OS is booted most data is access from a remote location either over a NAS, a SAN or from a Distributed Storage solution.

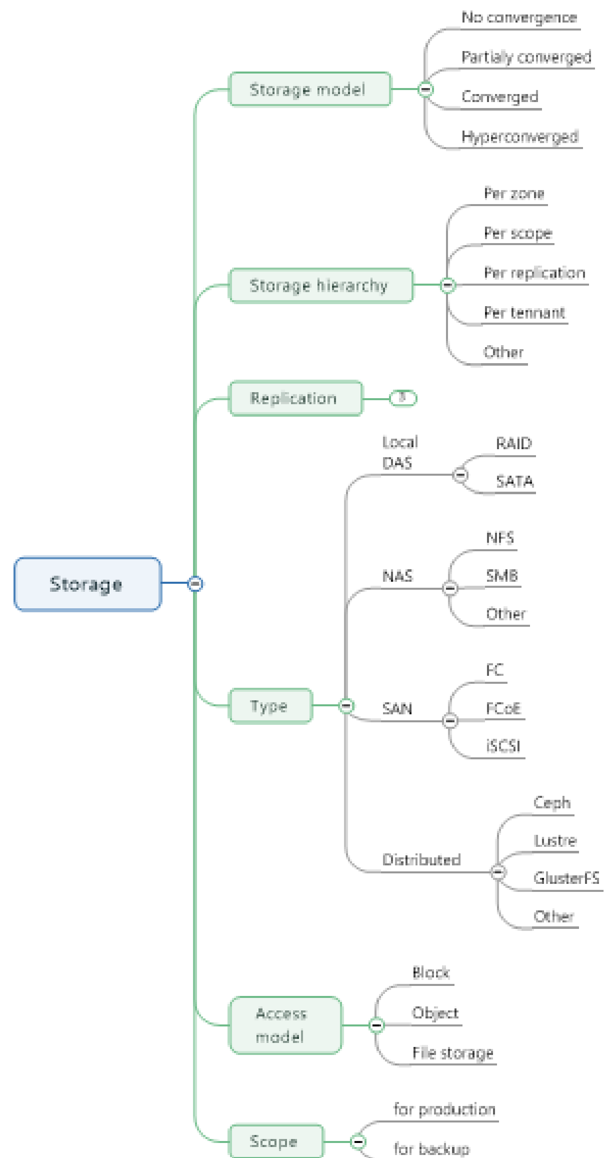
Three data access models are common:

1. **Block storage** – the entire storage is visible as a big, continuous, storage space to the operating system which then reads and writes data in fixed-sized blocks (e.g. 512 bytes or 4KB of data at time). This type of storage can be used to *bootstrap* an operating system and can be *mounted* and formatted using any *file system* by the host compute node.
2. **Object storage** – data is stored as objects. Each object typically include the data itself (of variable size, from a few bytes to gigabytes), a metadata (smaller size, usually stored as a set of key-value pairs) and a globally unique identifier. Useful for storing various types of, usually large, distributed pieces of data (e.g. pictures, videos) that don't make sense to be stored in a database.

[18] Nodes that are managed by the cloud platform but do not have a host operating system installed by default are called *bare metal*. No virtualization is involved. Keyboard, video and mouse of this nodes is captured and transmitted remotely (e.g. VGA video analog signals are digitized and converted to an IP video stream). Also the power and reset button are managed remotely. CD-ROMs and USB devices are emulated and remotely connected. Cloud Management Platforms install operating systems on these nodes just as a user would do if physically connected to the server. Sometimes users can take control of the installation process. This is just like installing an operating system on his own computer yet it is actually doing it somewhere in a data center.

[19] There are cases where the OS is booted from a remote location (e.g. via PXE or from a NAS). This approach is less often used than booting from local storage.

3. **File based storage** – keeps data in a hierarchical tree of files and directories. Can be local (i.e. the FS of an Operating System) or remotely accessible via a network protocol (e.g. NFS or SMB). Remote file based storage systems can be built on top of distributed block or object storages.



The types of storage are:

1. **Local, direct-attached storage (DAS)** – a HDD, SSD, NVRAM or a RAID array device that keep persistent data close to the server. This storage is directly attached to servers using a SATA (mSATA) or PCIe connection. The drives themselves are located inside the server. Performance is high, information is easily accessible with no communication needs but storage can easily be wasted as it is not remotely accessible.
2. **Storage Area Network (SAN)** – Devices are connected remotely through a storage network to servers. A SAN storage array comes with a management application that allow partitioning of the array space between multiple servers. A server BIOS then connect to the array and makes those partitions locally visible to the operating system. From an operating system perspective, the remote storage looks just like a local, directly attached storage. Performance is very good, capacity and reliability is high but cost can be prohibitive for some deployments.
3. **Network-attached storage (NAS)** – Provides remotely accessible file based storage over TCP/IP. A server runs a special applications that exports its local FS to other nodes through a NAS protocol (e.g. NFS, SMB). Servers that provide the NAS can have their own storage as either local (DAS) or connected remotely (i.e. a SAN or a distributed storage) in this case it acts like a gateway that creates the file based storage for other nodes to use (i.e. NAS using DAS or NAS using SAN). Performance usually is not the main target of these solutions but the shared file based access that is provided to the servers.

4. **Distributed storage** – provides, block, object or file based storage[20]. The difference to NAS is that information is distributed between multiple servers which behave as one storage system. The servers have their own DAS or, rarely, NAS connected storages. Distributed storage solutions may also have special servers that do not export storage space but manage the cluster or provide gateways to the data stored in the cluster. It has good performance (similar to that of a SAN), low cost when compared to SAN but uses TCP/IP and can be error prone[21]. Distribution storage solutions are not new to the market but they had many disadvantages that reduced adoption. In the last years many of these problems have been solved and as QoS and congestion control has improved.

Other aspects taken into consideration are storage hierarchy, replication level, scope (e.g. production, backup) and storage model.

Storage hierarchy (or storage *tiering*) represents how the storage is divided into parts. Having a very big pool of storage may not be easily manageable, some systems does not support arbitrarily big storage arrays and it may affect performance or security. To solve this, DC operators may choose to implement hybrid solutions, divide an existing cluster in partitions or implement multiple storage clusters that use same or different technologies for different user groups. The hierarchy depends on the scope of a storage, for example, a backup cluster may be implemented using low performance but high capacity disks, or even tape drives while a cluster holding an enterprise database may be implemented using SSDs for fast access.

Replication level is important, especially for distributed file systems[22], if data reliability or read performance is desired. A replication of 1 (one) means that a single copy of the data is kept inside the system and a replication of *n* means that *n* copies of the same data are kept and that data is recoverable if *n-1* replicas fail at the same time. Read performance also increases with the replication level but storage space requirements increases and write speed decreases. Also, a greater replication level means that network throughput is increased so, if a host writes with 100MB/s and the level of replication is 3 then the actual total network usage will be of approx. 3Gbps.

Storage model represents the degree of integration between storage networks and data networks. A storage network without convergence means that storage nodes have their own independent network separated from the data network. Partial convergence of storage is achieved when some of the storage traffic moves over data network, full-convergence is achieved when all of the storage traffic moves over data network and hyper-convergence is achieved when nodes combine storage and compute and the storage of nodes is also shared using a distributed file system. For more details on this topic *see 3.2 Data Centers classification and characteristics – Storage model* on page 38 and *3.4.1 Evolution of storage architectures* on page 54.

[20] Can be NAS or using a proprietary protocol.
[21] Data is distributed, therefore probability of failure of a device is directly proportional with the number of devices in the cluster, therefore data replication is a must
[22] RAID supports replication level of 2, known as RAID 1 mirroring, but for reliability it also provides error correction codes that guarantee data is recoverable even without a one-to-one replication.

Distributed storage, SANs and partial converged storage impacts networking usage directly, the impact is discussed in Ch. 5. The solution proposed there improves network throughput by reducing congestion.

3.3.1.3 Networking design decisions

Data Center network design decisions are influenced by network capacity, scalability, reliability and cost. Automation, in most cases, is just an enabling factor, not a direct requirement.

As data centers are constantly evolving, the network design is usually done for the entire network from the beginning but only parts of it are deployed in the first commissioning phase. Also, a lot of room need to be provisioned for future technologies that, given the 10-15 years lifespan of the DC do not exist yet.

When designing the DC network we have two perspectives:

1. External, backbone network access
2. Internal networking

The first part is usually implemented by the Network Carrier and it is his responsibility to bring the backbone into the datacenter. When the DC operator is also a Telecom Operator, it may become an internal design activity.

From a DC design perspective, this thesis only considers the internal networking. WAN or SD-WAN are complex topics with different characteristics and needs than those of a DC.

In DCs, external network access redundancy is important, therefore two connections to the external world are needed. Usually both of them are used in parallel to balance traffic or, if one of the connections is order of magnitude slower than the other, the second one may only be used for backup. The two connections are provided by same or different carriers.

Networking design aspects:

- Hardware – multiple vendors are available. Price is a big constrain as it increases fast with the number of ports and feature set. In SDN, as they are only required to provide flow passed forwarding, cost is reduced.
- Software - choice is mainly imposed by the technology used throughout the network:
  - Traditional networking – device management and monitoring software plus a Configuration Management Application are needed;
  - SDN - Controller and the application that are running on top of it.
- Topology – is chosen mainly based on performance requirements and hardware plus software constrains and limited by cost. Another two constrains are cabling and scalability as too much cabling will increase complexity and scalability may hinder feature upgrades. For details see section 3.5.3.
- Cabling – Management, data and storage network may need different cabling. In SDN both management and data are tunneled through the data network which reduce cabling complexity.

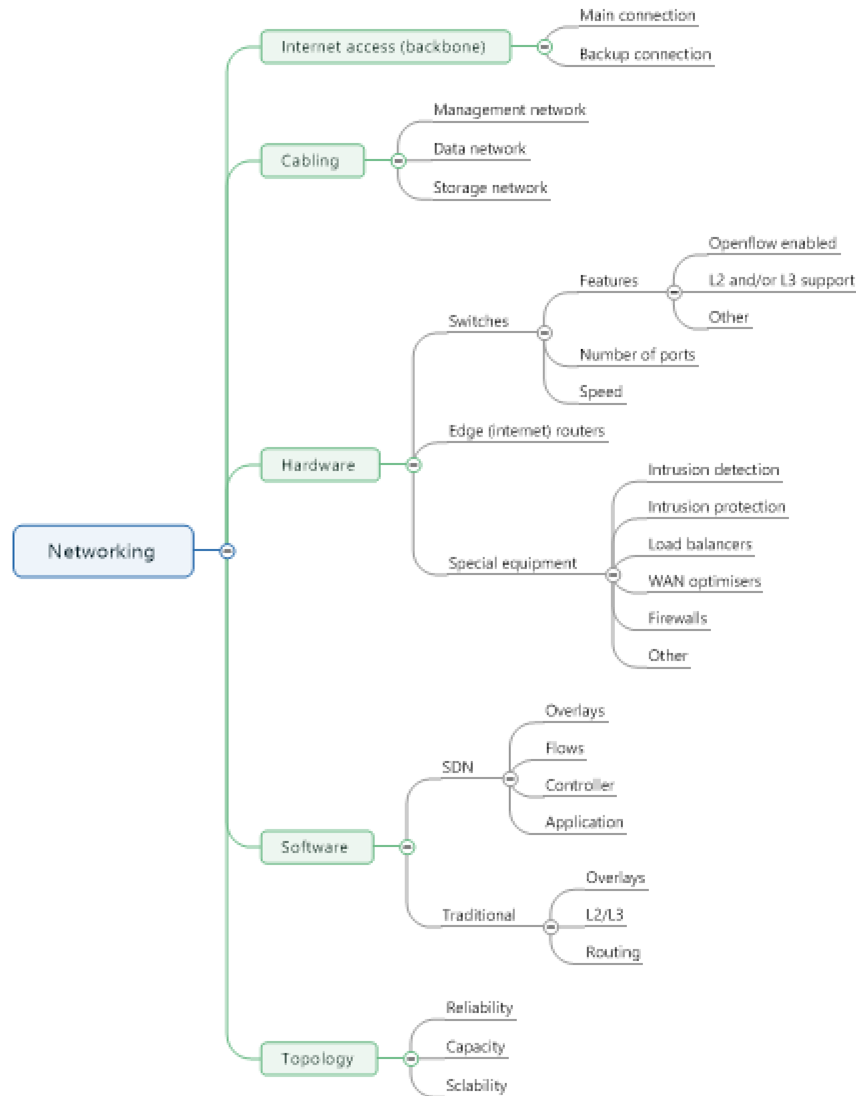


Figure 19: Networking design decisions

3.3.1.4 Data center management and operations

Data center management is done using a set of applications that integrate monitoring with planning, modeling, and asset management. These applications may also integrate with other systems such as service management and financial services.

Three categories of applications stand out (see :

1. **Computerized Maintenance Management System (CMMS)** – provides an integrated solution for the operational maintenance of data centers with the goal of supporting DC operators in increasing asset life, tracking maintenance details, predicting and preventing equipment failures, reducing downtime, and lowering the costs of maintenance.
2. **Data Center Infrastructure Management (DCIM)** - collects data from devices, sensors and meters, stores then processes this information into a more manageable form through its many modules and report it back to the operator. It also provide modules for capacity planning and analytics.

This application is important as it provide the network operator with a complete view of the Data Center through its real time reports and dashboards. It also provides the data for the CMMS and other applications that need it.

3. **Cloud Management Platform (CMP)** – manages cloud resources (compute, storage, and networking) and provides IaaS services. On top of it a PaaS and/or SaaS may run. In this area many opens source projects alternatives exists: CloudStack, OpenNebula, Eucalyptus and OpenStack for IaaS and OpenShift and AppScale for PaaS. The most well-known and successful is OpenStack. A short analysis is presented in Table 3. Based on it the best choices at this time seems to be OpenStack for IaaS and OpenShift for PaaS.

In SDDCs, DCIM communicate with CMP and with the SDN Controller to automatically optimize resource usage. For example, in case that congestions routinely occur in a part of the network then DCIM, based on the information from CMP and SDN Controller may move some VMs from nodes causing congestions through releasing valuable network resources. After congestion, CMMS may even decide to power off equipment that is unused or, instead of migrating instances, it may decide to power on backup equipment or inform the maintenance team to create temporary network connections till the congestion is resolved.

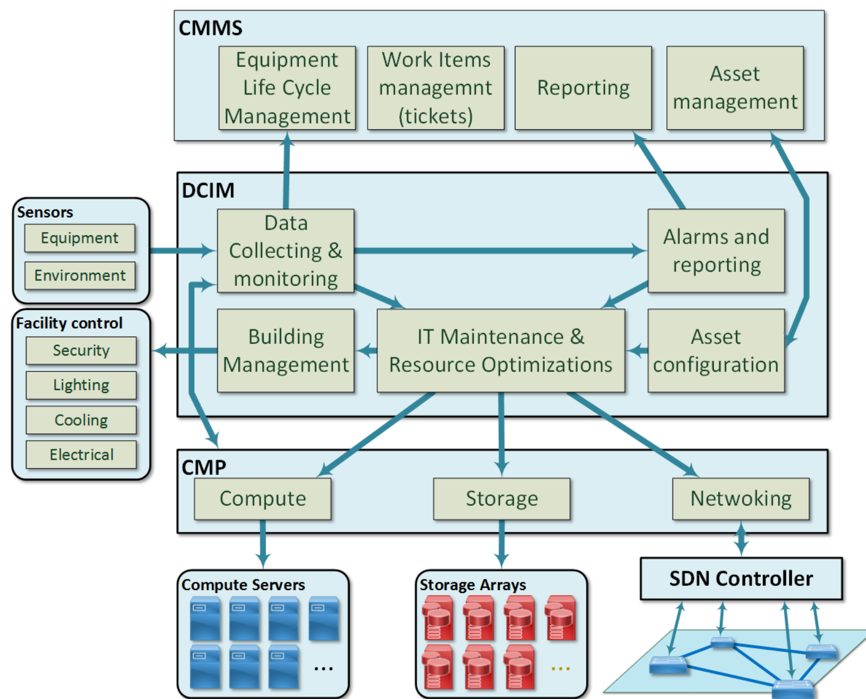


Figure 20: SDDC Management Applications and their relations

The SDN Controller is just one of many applications and works in close relation with CMP. These two applications are the most complex and critical to a correct DC operation. CMP has to able to manage multiple categories of resources and be able to abstract compute, storage and networking resources for multiple tenants. And the controller need to efficiently manage networking, all of the network devices and provide a broad range of services.

All three applications provide Dashboards (GUI) applications to ease management and most of them also provide a CLI and APIs to ease administration.

Table 3: Open Source Cloud Management Platforms

Application	Type	Programming language(s)	LOC	Activity (commits & contributions)	Notes
CloudStack	IaaS	Java 55%, Python 18%, JavaScript 10%	1.06 Mil	50/mo, 15 cont/mo	5% proprietary. Number of contributors and commits dropped dramatically in the last two years.
Eucalyptus	IaaS	Java 50%, C 14%, Python 11%, JavaScript 6% (Other 19%)	0.6 Mil	80/mo, 9 cont/mo	Number of contributors and commits is dropping.
OpenNebula	IaaS	C++ 48%, Ruby 36%, Shell script 10%, 26% (Other 6%)	0.14 Mil	260/mo, 10 cont/mo	Activity is small but constant.
OpenQRM	IaaS	PHP 67%, CSS 5%, Shell script 16% (Other 12%)	0.4 Mil	~=0/mo, -/year	Very low activity
OpenStack	IaaS	Python 82%, XML 7% (Other 11%)	1.7 Mil	3500/mo, 370 cont/mo	Code base increases very fast!
OVirt	IaaS	Java 59%, Python 16%, XML 10% (Other 15%)	1.5 Mil	650/mo, 70 cont/mo	provides a web interface for managing libvirt, rapidly growth (started in 2011)
OpenShift	PaaS	Go 43%, Ruby 36%, HTML 8% (Other 13%)	1.4 Mil	760/mo, 40 cont/mo	Steady increase
AppScale	PaaS	Python 53%, Go 24%, C 8% (Other 15%)	1.2mil	160/mo, 7 cont/mo	Low number of contributors, steady code base
Cloud Foundry	PaaS	Go 47%, Ruby 9%, PHP 11%, (Other 33%)	3.5 Mil	1750/mo, 150 cont/mo	Rapid code size increase, heavy development

3.3.1.5 Conclusions: Impact on Networking

When designing the networking, SDN or traditional, the data center should be seen holistically as many decisions have impact on networking decisions.

Two categories of factors impose the design of networking:

1. Requirements – a tradeoff between performance (e.g. capacity, latency), reliability, scalability and cost.
2. Constrains imposed by the other aspects of a DC design.

3.3.2 Modular data centers building blocks

A data center room is divided into multiple functional parts. ANSI/TIA-492 standardizes the architecture – and specially networking and cabling – by naming those zones according to their functionality, see Figure 21. The most important areas are Horizontal Distribution Area (HDA), Equipment Distribution Area (EDA) and Main Distribution Area (MDA) and they correspond to Access, Aggregation and Core layers of the network (more on network layers in next section).

- **Main Distribution Area (MDA)** – is a centrally located area where the backbone enters the data center (*main cross-connect*) and *core* routers, switches (for networking and storage) and *special equipment* (IPS, IDS) are located. Multiple MDA areas may be present in bigger (multi-room) data centers.
- **Horizontal Distribution Area (HDA)** – is used for aggregating connections from individual racks (cabinets in EDA), for housing active equipment needed for these connections (switches) and for connecting to the MDA.

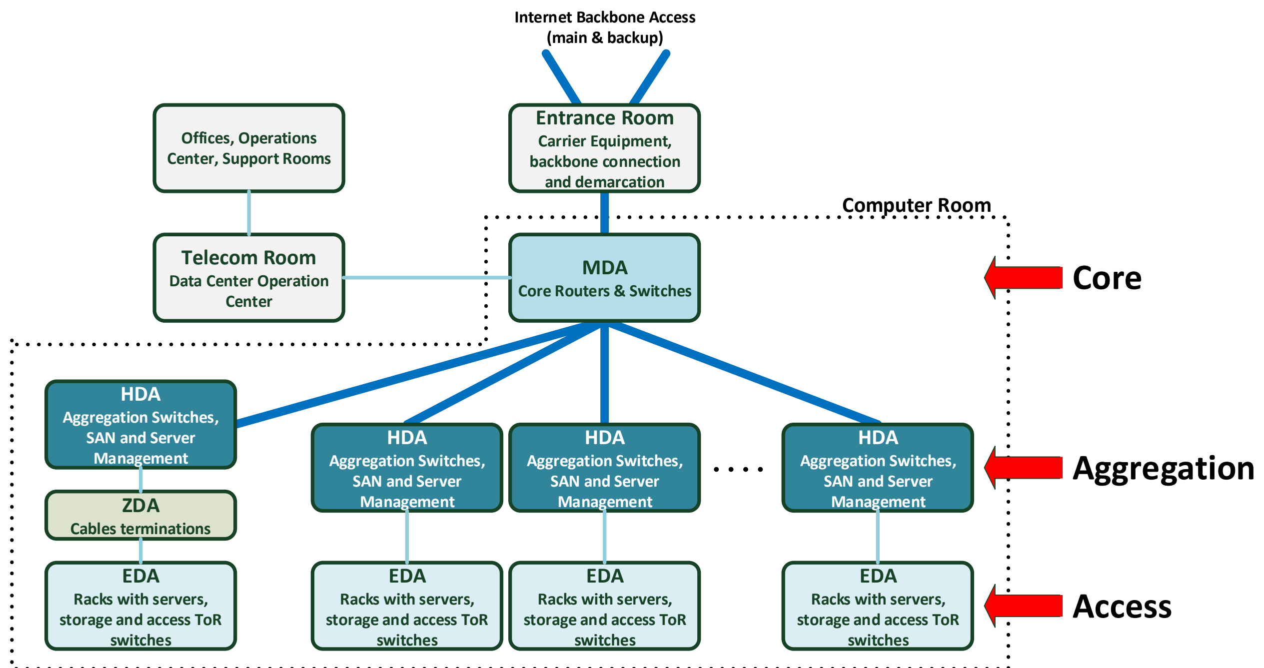


Figure 21: ANSI/TIA-492 Data Center Architecture

- **Equipment Distribution Area (EDA)** – a zone with compute and storage racks. Cables coming from HDA terminate in the EDA, either in a patch panel or ToR switch. Equipment connects either to the patch panel or ToR through short patch cables.
- **Zone Distribution Area (ZDA)** – is an optional interconnect between HDA and EDA, usually is composed of a set of patch panels. Is useful for terminating cables that are already installed but

for equipment that is not yet installed. When a DC is built, sometimes, is better and cheaper to have most of the cabling in place even if equipment will be installed in the future.

The *network carrier* installs its equipment in the *entrance room*. This room needs to be separated as carrier technicians usually have access to it for installing new equipment and for maintenance. *Offices operations* and *Telecom room* are for managing the DC operations. DCIM and CMMS are usually installed and accessed from here.

3.3.3 Basic networking design: the hierarchical model

Before describing the typical layout, let's first see how a datacenter networking basic architecture looks like. This architectural structuring of topology and equipment is the most common one used in datacenters as it fits very well with the ANSI/TIA-492 architecture, with datacenter layout and it provides good scalability.

The hierarchical model is part of Cisco's standard network design guides [56] and it has a big impact on network topology. Some of them are well suited others are not. This is one of the reasons why some topologies (as we will see in section 3.5.3) are preferred.

The three layers of the model are:

1. **Core** – this provides fast access between the distribution points in a network and backbone connections. No packet modifications is done at this layer as speed and reliable delivery of packets is the most important aspect. Operations are usually done at layer 3 only.
2. **Aggregation** (also known as **Distribution**) – Provides most of the packet filtering, policing, QoS and provides different protocol access gateways (L2 to L3 traffic) and special equipment (e.g. firewalls, hardware load-balancers, WAN Optimizers). This is the layer where virtual LAN segmentation happens (e.g. via VLAN) and where broadcast domains are created. The traffic is both Layer 2 and layer 3. Routing between segments also happens at this layer.
3. **Access** – where nodes (servers, special devices) access the network. It is usually dealing with layer 2 traffic only.

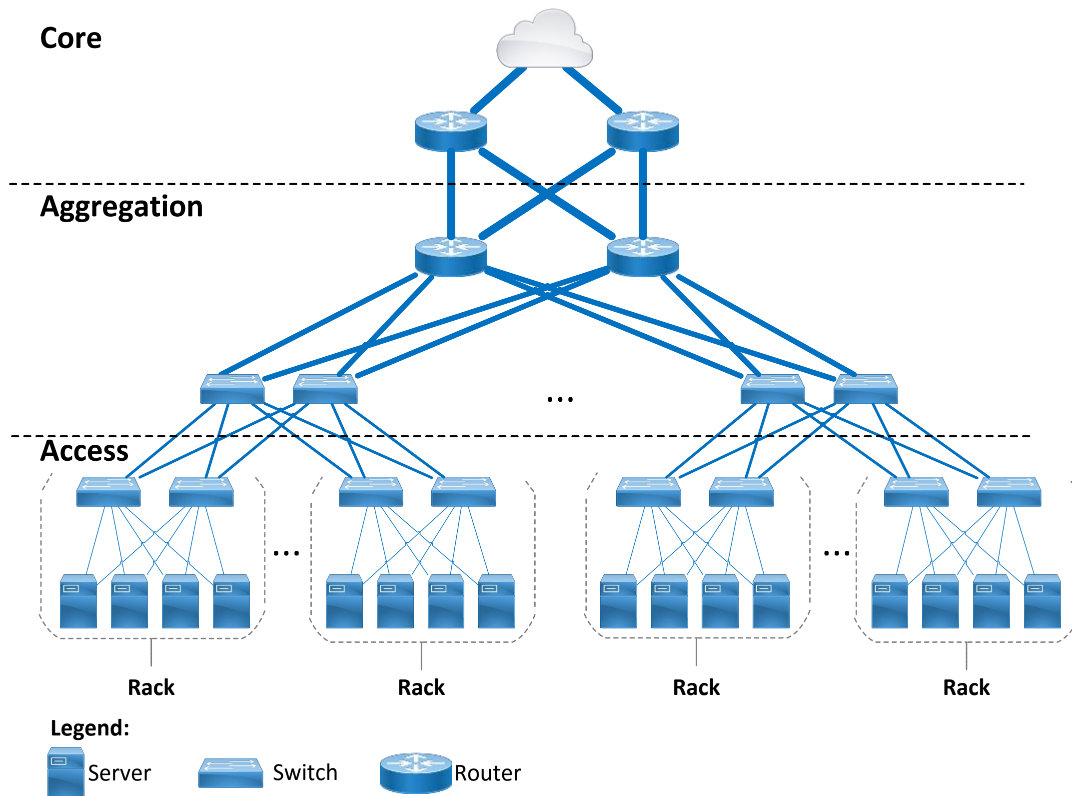


Figure 22: The hierarchical architectural model with redundant connectivity

Figure 22 shows the hierarchical architecture with access, aggregation and core. Four racks are displayed and each rack has only 4 servers. Each layer provides redundancy as networking devices are doubled, if any of the switches fail communication to the switches still works correctly, this a common approach in data centers.

3.4 Storage

Storage evolved from having its own network to sharing the same one with data. This is advantageous from a cost and deployment perspectives. Having a single network is much simple to install, uses less power and it has a single set of equipment yet it comes at a cost, supplementary network traffic generated by storage nodes. This traffic has specific characteristics and needs special policing to proper manage it.

3.4.1 Evolution of storage architectures

Over time storage solutions evolved from local to remote and finally to distributed storage. When looking at this evolution from a networking perspective four stages can be identified (Figure 23)[23]:

1. *Independent storage network* – where a SAN is connected to servers through a separate storage network independent from the data network. This approach is considered to be legacy with no new deployments. Yet many datacenters that still use this approach need maintenance and

[23] These phases are the same as storage models first presented in 3.2 on page 40.

hardware compatibility. It uses non-Ethernet protocols such as Fibre Channel (FC) and connection to the servers requires special cards named Host Bus Adapter (HBA). The main issue is high cost as data centers need special equipment to create the independent storage network (i.e. HBAs, FC switches and wiring costs are consistent).

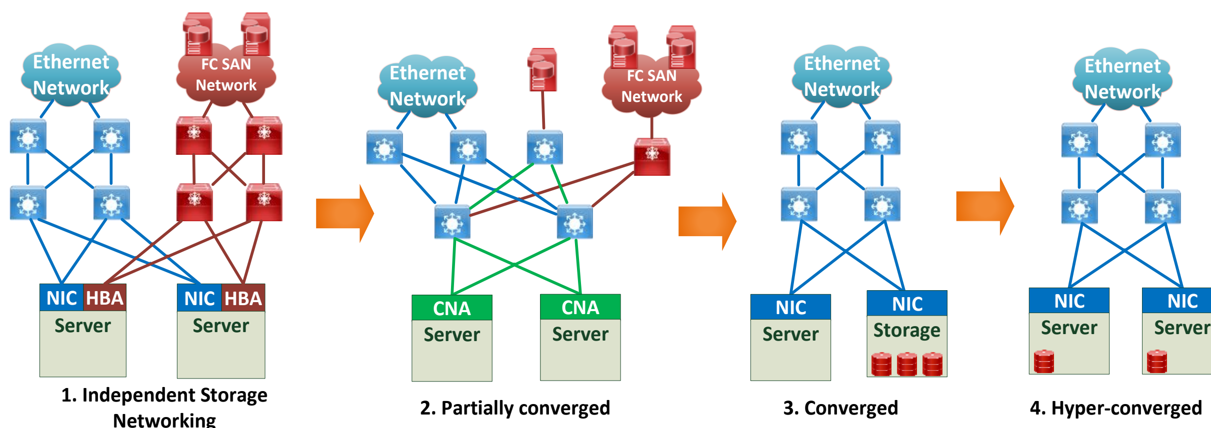


Figure 23: Data Center storage evolution

2. *Partially converged storage and data networks* - to reduce cost, the storage traffic was moved over Ethernet and the FC protocol was adapted to work over Ethernet – it became Fibre Channel over Ethernet (FCoE). Another protocol, similar to FCoE is iSCSI. The main issue with this approach is intolerance to packet loss as both protocols encapsulate SCSI commands. This implies that the data network is lossless, something hard to achieve with Ethernet but can be done as later shown in Ch. 5.
3. *Converged storage over data network* – uses distributed storage. Storage traffic moves over the Ethernet data network, protocols natively use TCP/IP and are more resilient to packet loss. Storage nodes are separate from compute nodes.
4. *Hyper-converged* – is also a type of convergence, the difference is that nodes combine storage and compute. In the converged model storage is separate from compute. From a software perspective the implications are minor as the node usually just have to run both storage and compute stacks at the same time on the same operating system (same kernel), the advantage is that instances running on the hyper-converged node can just go to the local storage for data and there is no need for separate storage hardware. Also, unused local storage that otherwise would be wasted is now part of a distributed storage pool and can be used by nodes that otherwise would not have enough space on their own local drives.

SDN networks are able to handle FCoE traffic as, from their perspective, it is just traffic from another Ethernet protocol but they need to have the necessary support for the QoS classes and, more important, to have mechanisms in place for avoiding network congestions. QoS mechanisms exists in many switches, but congestion control does not exists yet. Such a mechanism is proposed in Ch. 5.

The last two stages of evolution also started to be heavily deployed in modern SDDC datacenters because distributed storage solutions integrate very well with Cloud Management Platforms.

3.4.2 Short introduction to Distributed Storage and its theoretical impact on networking

In the last 3 – 4 years distributed storage solutions evolved dramatically both in performance and reliability. Many distributed file system solutions exist today, some are simple and have a reduced feature set and others are more complex and have more features and better performance. We will stop at one that is, arguably, considered as the most performant and sophisticate of all of them: Ceph. This File system became production ready a couple years ago and, since then, many companies started using it inside their datacenters. After efficient storage that works on standard TCP/IP became feasible the need for costly independent storage networks made no sense.

Ceph is a distributed storage solution that provides block, object and file system storage. Ceph is open source, massively scalable and software-defined. At its lowest level it stores *objects* (chunks of data) that are distributed between nodes in a network. On top of this different client applications provide block storage, external objects and file system hierarchies.

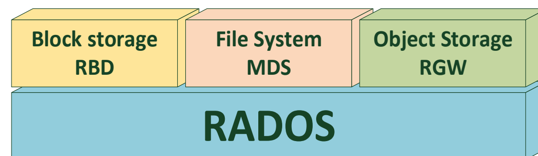


Figure 24: Ceph simplified architecture

Internal objects are managed by RADOS (Reliable, Autonomous and Distributed Object Store) which is composed of two node types:

1. **Monitors** – manages information about placement of objects and node status (e.g. *up* or *down*). Clients use this information to access objects. Monitors keep the same information but shared and synchronized through a Paxos consensus algorithm. At least three nodes have to be functional at all times for the metadata cluster to be considered healthy.
2. **Object storage itself** – called Object Storage Daemon (OSD), which is managed by a background service (i.e. a Linux daemon process) that converts a drive into an OSD (can be HDD, SSD, NVRAM etc.)[24]. Multiple OSD processes can be executed on the same server but each service manages a single drive. Resource usage is around 1GHz of Xeon class processing power with 1GB of RAM for each process [57]. OSDs keep objects, take care of replication (OSDs are able to communicate with each other) and monitor each other's status through heartbeats (i.e. if an OSD detects that one of its neighbors is down then it reports it to the cluster of *monitors*).

On top of RADOS three client types exist:

1. **Rados Block Device (RBD)** which provides block storage. Access to it is done either through a Linux kernel driver or through a driver in the Qemu KVM hypervisor[25] [58] [59].
2. **Ceph File System (Ceph FS)** - It is accessible either through a Linux kernel driver or a user space application[26], an unofficial Windows version is also available [60]. It needs at least one Metadata

[24] Or a partition on the drive
[25] Qemu is an open source paravirtualization technology and works with KVM to provide hypervisor based virtualization.
[26] Uses Filesystem in Userspace (FUSE)

Server (MDS) to keep the file system tree. Multiple MDS servers can work together either to provide HA – in this case one is active and the other one is stand-by – or in active-active by dynamically partitioning a tree between MDS nodes. In the latter case workload will also be distributed between nodes. Mixed configurations are possible (e.g. 3 active and one standby).

3. **Rados Gateway (RGW)** – provides object storage for external use through REST APIs. Two interfaces are provided, one compatible with Amazon S3 and another one with OpenStack Swift[27].

From a networking perspective most traffic comes from OSDs. Monitors, RGWs and Ceph FS represent only a small percentage of it. The reason is that metadata is smaller in size than data itself, yet it is critical to the speed of the overall storage cluster as many decisions are made based on it. So it needs to have QoS rules that provision for low latencies.

Inside a distributed storage system (not only Ceph) data is replicated on multiple nodes for redundancy. Replication happen in two modes:

1. **By keeping multiple copies** – multiple copies of an object is kept on different nodes. So, to provide redundancy to one failure, at least two copies need to be kept. Writing is slower and requires more network throughput. At least twice as slow as writing a single copy of it but reading speed is multiplied by the number of copies. The disadvantage is that storage space increases. If R is the replication level and B_{size} is the size of a block then space used by a block is:
$$B_{space} = R * B_{size}$$
2. **By using Erasure Codes** – data is stored using a Forward Error Correction algorithm which provides very good redundancy and increased capacity at the cost of reduced read speed and increased throughput. For this two parameters can be configured: K and M. The size of a block is first divided by K and each sub block copied on an OSD. M is the number of OSD failures we want our cluster to be resilient to. In this case used space is:
$$B_{space} = \frac{B_{size}}{K} * (K + M)$$
Much better than when using multiple copies (e.g. for $K = 2$ and $M = 1$ only 50% more space is needed). The disadvantage is that, when reading, at least K sub blocks from the total $K + M$ have to be read before data can be reconstructed.

When computing the throughput we want to avoid that networking is the bottleneck at *client*, in the *core network*, and at *storage server* levels. At *clients* (i.e. compute servers) data is requested in parallel by a number of *threads* and needs to come in with no bottleneck. In the core network congestion should be avoided and at *storage server* we need to make sure that the performance of the HDDs is not higher than what the network can provide.

Theoretical read throughput if all OSDs use disk drives with the same speed (S_{read}) is:

[27] Swift is the OpenStack component that provides a standardized interface for Object storage.

$$Read_{client} = \begin{cases} \sum_{i=0}^{Th} S_{read} * R, & with\ replication \\ \sum_{i=0}^{Th} S_{read}, & with\ erasure\ codes \end{cases}$$

$$Read_{storage} = \sum_{i=0}^{N_{OSDs}} S_{read}$$

Where N_{OSDs} is the number of OSDs installed on the storage node and Th is the number of parallel threads used to download data[28].

Theoretical maximum write throughput:

$$Write_{client} = \sum_{i=0}^{Th} \frac{S_{write}}{X}, where\ X = \begin{cases} 1, & R = 1\ and\ no\ erasure\ codes \\ 2, & R \geq 2\ \ \ \ or\ erasure\ codes \end{cases}$$

$$Write_{storage} = \sum_{i=0}^{N_{OSDs}} \frac{S_{write}}{X}$$

As an example of maximum theoretical throughput, on a 60 OSD storage server unit if 3TB commodity hardware is used and each HDD has a maximum of 200MB/s results in summed throughput of 1200GB/s which translates to approximately 120Gbps. A theoretical throughput that is much higher than the capacity of the fastest Ethernet links available today – 100Gbps. If the real performance expected from the array approaches the theoretical limit then networking will get in real trouble as it will not be able to easily achieve it. It's possible to use 2x 100Gbps aggregated links or 3x40Gbps but, since this links are expensive, it will also increase the costs of storage dramatically.

Such a high throughput affects the centralized SDN controller, at least its configuration if not the overall design – controllers need to rapidly respond to congestions by optimizing routes (migrating them to less congested links), otherwise they get overwhelmed with requests and start introducing delay in the entire network. See 5.3 and 5.5 for more details.

Note that in practice drive speeds vary greatly ($S_{read}$and S_{write}) with drive type, data density, inner and outer disk diameter and seek time. Therefore maximum is not reachable in most cases. Therefore benchmarking the storage cluster is very important yet knowing its theoretical limitations will help identifying bottlenecks early on.

3.4.3 Ceph: Hands on performance of distributed storage and its real impact on networking

To get an impression on the real performance of distributed storage a small Ceph storage cluster was benchmarked. The hardware used is equivalent to the one in converged data-centers: dedicated servers for storage monitors and object storage. Servers have Intel Xeon 10 core processors with 64GB RAM and

[28] A file is usually accessed by a single thread. And multiple processes and VMs access data in parallel from multiple servers

10Gbps links connections, as shown in Figure 25. Three of the servers where dedicated to Ceph Monitors to fulfill the minimum of 3 required by Ceph for a healthy cluster and two of them were used for storing object data, each server had 6 OSDs.

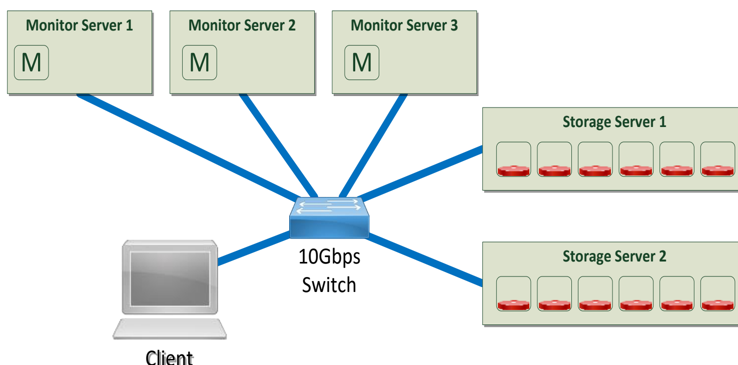


Figure 25: Ceph benchmark setup

The storage nodes are identical, with six HDDs and one SSD to store the journals for all of the OSDs on that node. Both HDDs and SSDs are enterprise class and currently used in Data Centers. One pool was created with a replication set to 2 (i.e. two copies of the same data were kept – this is a common settings for Data Centers as it provides a minimum level of redundancy). The data partition used XFS as underlying FS and Journals were using RAW partitions on the SSD drive.

Table 4: Ceph setup drives

	HDDs	SSDs
Capacity:	1 TB	800GB
RPM	7200	N/A
Sequential write:	**190MB/s**	**420MB/s**
Sequential read:	**170MB/s**	**390MB/s**

On this setup two tests were performed, one to validate the overall performance of the cluster using standard industry testing application *Ceph Bench* which is made by the same team that created Ceph and some tests with a VM running an the Client machine that uses a block device from the cluster. Performance inside the VM is similar to the overall performance observed when running *Ceph Bench*. Benchmarks were executed with different block sizes and number of parallel threads. Also, read and write buffers were flushed before executing each benchmark to avoid using RAM buffers created automatically by Linux.

In Table 4 sequential read & write was benchmarked at the outer diameter of a disk platter so performance was at its peak. For the chosen HDDs performance will drop by 2.5 times when reaching the inner diameter of the platter (to 76MB/s).

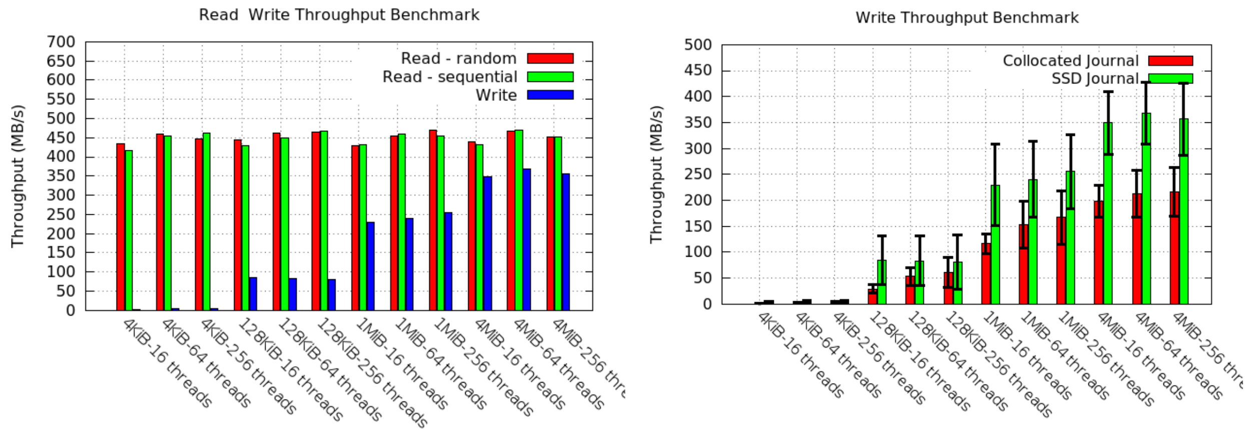


Figure 26: *Left*: Read and write benchmark (with SSD journals). *Right*: Impact of SSD journals on write throughput

Plots in Figure 26 show that read performance is stable at around 450MB/s and write performance depends greatly on block size. Also, moving journals to SSD has a great impact on write performance.

From a networking perspective, the Storage Cluster created a 4.5Gbps total throughput when reading from two servers (setup is configured with a replication of 2 – it creates 2 replicas for each block of data for redundancy) and half of it came from a single server, 2.25Gbps, extrapolating this to 60 disks, the maximum number of OSDs on a single storage node results in 22.5Gbps of traffic that can easily be handled by a 40Gbps link and it may also be handled by two 10Gbps aggregated as suggested in [61] where they had 2 servers with 64 HDDs and two SSDs for journals and with replication configured to one.

The total throughput is dependent on the type of workload used. Sequential operations with large blocks (i.e. 4MB blocks) yield much better performance than random operations with smaller blocks. In data centers performance is abstracted as IOPS – Input/Output operations per second. This eliminates the difference in performance caused by block size and platter diameter. IOPS is relatively constant for a HDD.

Current generation drives provide on average 100 to 200 IOPS/s [62] depending on model. This means that if all operations are on small blocks (4KB), e.g. common in database access or big data analytics, total throughput of a single disk is very small, between 0.39 to 0.78 MB/s (number of IOPS multiplied by minimum HDD block size of 4KB) therefore a storage node with 60 drives will be able to give only 23 to 46MB/s (60 * 0.39 and 60 * 0.78).

If large blocks are used, e.g. for multimedia, large pictures or large file downloads throughput can reach maximum level of HDD (given by its sustainable transfer rate) which is between 76, on inner plater diameter, and 190 MB/s, on outer plater diameter, for a 7200RPM HDD. This in theory could drive a storage cluster to 4.5-13GB/s (60 disks multiplied by 76 or 190 MB/s), an impressive number that can only be managed with a 100Gbps link.

In conclusion, theory shows that, in order to benefit from full throughput of the HDDs inside a Ceph cluster the data center operator has to provide high bandwidth 100Gbps links for Storage Servers with a high number of HDDs (i.e. 60/server) while reality shows that a 40Gbps or even two 10Gbps links should provide enough capacity for most cases but final choice depends on workload type used with the storage cluster.

To note that an SSD provides around 400-500MB/s throughput even for smaller blocks as it has no latency caused by repositioning heads on spinning drives, therefore a storage node with SSDs needs very high link bandwidth. In this case a 100Gbps is easily filled by 20 SSDs.

3.5 NETWORKING

As shown in chapter *3.3.1.3*, many technologies are available for building a network, both from hardware and software perspectives and that some of these technologies are not SDN ready. In chapter *3.3.3* we showed that usually networking in a DC is structured on three layers *Access*, *Aggregation* and *Core* this creates some limitations, especially in the choice of a topology. Also, in chapter *0* we showed that racks and cabinet placement is also impacting the topology, overall performance and scalability of the network. In this chapter we also showed that not all of the datacenter have the same design. More than that, a single data center may mix different technologies, forming clusters of nodes with different configurations.

This information gathered in the previous chapters will be complemented with a traffic profile, network topologies, specific SDDC approaches and some performance characteristics of virtual switches. All of this should give enough information for making better networking choices.

3.5.1 Classification of traffic patterns in Data Centers

Traffic profile of applications running in a DC is used by administrators to optimize the network for better application performance. Information for creating a traffic profile can be obtained by sampling packets in the network and analyzing the samples based on different criteria.

Sampling traffic is done using protocols such as Switched Port ANalyzer (SPAN) or Remote SPAN (RSPAN)[29] which mirrors parts of traffic from ports of a switch and transmit it to a central location for analysis[30]. When many ports are mirrored, to reduce total throughput to the central server, only some of packets on each port are transmitted (sampling rate for high bandwidth can be 1/50.000 packets). Statistically, over long intervals, sampling provide accurate information. In SDNs some of parameters of a flow can be monitored by reading its counters (e.g. number of packets transmitted in a flow)[31].

The statistics are then used to classify traffic and create network policies, either by updating configurations of switches in traditional networking, or updating the rules of SDN Controllers.

Analysis is concerned with determining correct classes of traffic and throughput of each class, either as relative (e.g. percentage of total, percentage over a period of time) or absolute values (e.g. MB/s, seconds or pps – packets per second).

[29] SPAN and RSPAN are also used for network debugging and security by routing traffic to an Intrusion Detection System (IDS).
[30] Sometimes this is done with a hardware tap – a device connected in the middle of a link that forwards all, or part of, traffic to another device for analysis.
[31] Most hardware switches have a limited number of counters for flows (e.g. 256 out of 16000), therefore getting an accurate view of the traffic can become a complex issue.

In today's data centers traffic is classified by application and then QoS rules are configured for each class. Analysis of traffic based on direction and bandwidth impact and flow lifetime have also been done, yet a complete classification based on multiple criteria is nowhere to be found.

The Controller is able to gather complex statistics much easily than in traditional networking and, based on them, can calculate metrics in each criteria automatically. This data provides a better view of a network workload. And it can then be presented to an administration (e.g. in a real-time dashboard or a report) and to the SDN applications though an API. Applications will use the data to make better decisions. Moreover, this classification based on multiple criteria provides more input for much better routing decisions.

This thesis proposes the following classification criteria:

1. **By direction** – represents the direction in which traffic flows through a network. From outside to inside or generated and consumed inside the data center. The quantity of traffic in each direction is usually measured in percent of total and is important in choosing the topology and link capacity.
    a. *East-west* – measures traffic between servers inside a DC. If this is high then links between servers need to be fast and to have a smaller delay. Backbone connection with outside world may have a smaller capacity.
    b. *North-south* – measures traffic from clients, outside of the DC, to servers inside and vice-versa. Backbone and connectivity between core and access needs to be faster as more traffic enters and exists the network. Topology has to also accommodate this pattern.
2. **By flow lifetime** – flow lifetime is important especially in an SDN context as best routing decisions need to be taken. For example, flows with short lifetime should not be rerouted in case of congestion, also expiration timers should be set to low values, and otherwise flows will remain allocated longer than needed as flow tables can accommodate a limited number of flows. In this case is important for the controller to automatically identify the lifetime of a flow and optimize for it. Measurements are done in milliseconds.
    a. *Short* – Usually a single request-response then connection closes. These are called *dragonflies* flows that last less than 2 seconds [63].
    b. *Normal* – Usually multiple request & responses packets are transmitted on the same connection, but connection closes after transmission finishes.
    c. *Long* – Almost permanently open. E.g. pooling between servers, status reporting, watchdogs and periodic packets. Called *tortoises* which are flows with a duration higher than 15 minutes [63].
3. **By bandwidth impact** – some flows have a bigger impact on the bandwidth than other flows
    a. Elephant flows – large, in bytes, continuous or bursty flow that uses a disproportionate share of the total capacity of a link. The protocol causing this is usually TCP with is slow start mechanism that created advantages for large flows while flows that have a short duration will be disadvantaged on congested links. Elephant flows have multiple definitions: (1) use 0.1% to 1% of the link bandwidth during a given measurement period [64], (2) not only contributes significantly to the overall load but also exhibit persistence

in time [65] and (3) if its rate exceeds the mean rate plus three standard deviations of the aggregate traffic for more than 500ms [66][32]

   b. Mice flows – anything else.

4. **By transaction type** – this thesis defines a network transaction as a sequence of information exchange that is needed to complete a specific task. A transaction can have a single source and a single destination (one-to-one), a single source and multiple destinations (one-to-many) or multiple sources and a single destination (many-to-one).
   a. *Unicast query & response* – client sends a small request and receives a single, usually small, response. Most east-west flows follows this pattern as they are small both in duration and bandwidth utilization. They also need small latencies as this is critical to RPC calls between processes on different machines.
   b. *Multicast query & incast response* – a single and small request sent to multiple servers will sometimes generate bigger answers from multiple servers coming at the same time. This may cause congestions on the receive link of a host as all packets arrive at the same time from links that usually have a higher capacity than the link to the requesting machine. This phenomenon is called *microburst* [67]. This is common for distributed storage [68] and should be avoided. Some methods include caching responses on the return path or delaying them at transmission. The incast problem is intensely studied [69] [70].
   c. *Stream* – usually voice & video, either unicast or multicast, but with constant bit rate[33]. Need to avoid packet reordering (reduce *jitter*) and route them though shortest path to reduce delay. The have a *medium* to *long* duration, have medium to large packet sizes and may be real-time. Rerouting them too fast should be avoided as that causes *jitter* (see case study results in section 4.5).
   d. *Best Effort data transfer* – files and *big* data structures. E.g. pictures, FTP transfer and distributed storage block transfers. Rerouting has a small effect, usually this type of traffic causes *elephant* flows.
5. **By packet size** – flows with big packets are processed almost at the same speed by switches and special devices as those with small packets. The reason it that most of the processing is usually spent with header manipulation. Therefore the share each packet size has of total throughput in a link affect network devices, especially in a Network Function Virtualization (NFV) context[34]. Many device manufacturers processing speeds are reported in packets per second instead of Mb/s. Packet sizes in each class may vary from data center to data center but previous studies [71] [72] have shown that small packet sizes of 40-100B usually count for 44% of packets and only 4% of bandwidth and 1400-1500B (or more) count for 37% of total packets and 47% of bandwidth. One such classification is proposed bellow:
   a. Very small – 40 – 100B
   b. Small – 100 – 552B
   c. Large – 553 – 1400B

[32] The authors call it *Alfa* traffic
[33] Variable bit rate streams are constant when averaged
[34] Special devices in a network are virtualized, e.g. firewalls, load-balancers, IDS systems are all software running in virtual machines.

    d. Very large – 1400-1500B
    e. Jumbo – up to max permitted packet size (usually 9000 bytes)
6. **By application** – this classification is important for monitoring and QoS. Some applications that may be part of this classification: Distributed Storage, HTTP, FTP, Voice (VoIP), Video, Torrent etc.
7. **By priority** – Strict in order prioritization of traffic may be enough for some deployments but usually a more fine grained approach is used with a classification similar to the one bellow:
    a. Hard Real time – both latency and throughput are guaranteed, this category is prioritized over all of the other. Used for critical traffic such as management packets between applications and sometimes even storage traffic, if access to storage is critical.
    b. Soft real time – a minimum is guaranteed but provides possibility for higher bursts if bandwidth is available. Common for variable bit rate video streaming where a minimum level of service is needed in order to have a successful transmission. If more bandwidth is available then it is granted to the services in this class. Storage traffic is also common here.
    c. Non real time – bandwidth is guaranteed but latency is not important. Good for large file transfers.
    d. Best effort – neither latency nor bandwidth is a requirement, data is transmitted on a best effort basis.

Based on these criteria the network operator creates a final set of classes that apply to his Data Center and program them in the SDN controller which automatically decides how to configure and use the resources of each switch and how to route and reroute traffic through the datacenter. Some types of routing may be more appropriate for a certain class – for example multipath routing may not be appropriate for voice traffic if the delay of each paths is not identical (or in an accepted range) as it creates too much jitter and packets will get dropped at reception for arriving too late and retransmission is costly.

Multiple issues can be observed here:

1. SDN controllers need to be smart enough to make good decisions based on the available information. Having good and fast algorithms is necessary.
2. Network characteristics, such as delay of each switch in the network, need to be well modeled in the controller otherwise decisions can be suboptimal or even damaging to network performance. This may be done automatically or manually by allowing the network operator to input the values for each device in the inventory.
3. Hardware needs to provide the necessary features (e.g. enough QoS queues and enough flows in the flow table) with predictable performance. For example, having random delays in switches would determine suboptimal decisions from the controller.

3.5.2 Typical query and response of client server application exemplified with Openstack CLI

An example of Data Center application is the Openstack Cloud Management Platform itself as, in large scale deployment, it can include dozens of servers that communicate with each other to provide the needed functionality. Its architecture uses REST *microservices* and is scalable both in *functional decomposition* as different components perform different functions, and in *horizontal duplication* as multiple clones of the same component can be instantiated to increase total processing capacity of that component.

Openstack is a typical example of a datacenter application therefore analyzing it provides us with a glance into the traffic patterns of most REST and microservices applications that run inside a datacenter. With this information we can later provide better optimization to SDN networks.

Openstack's services provide REST HTTP APIs to client applications. These include CLI, GUI and any custom application that is needed to orchestrate the cloud. When executing a simple CLI command to get a list with the virtual networks a lot of communication happens between services in the system. For this example the list of virtual network itself is not important, nor the virtual network topology as the output is not dependent on it. Any topology would be sent as a list. What interest us is the messages that are sent and their characteristics.

A typical command output is exemplified bellow:

```
[root@controller root]# neutron net-list
       +--------------------------------------+------------------+-------------------------------------------------------+
       | id                                   | name             | subnets                                               |
       +--------------------------------------+------------------+-------------------------------------------------------+
       | 0f5c5504-3e0a-41c5-91a2-fb5fe16bd41b | external-net0    | e8a2f1f1-d230-470b-a3c5-d55e36975748 192.168.1.0/24   |
       | cb4fa0c6-e449-4997-af90-6c57067491d8 | internal0-net0   | 812fec68-022a-4919-9f76-d81793919efa 10.0.1.0/24      |
       | bd646fb1-a65b-4d1c-8148-b9829c0fbad8 | tenant1-mgmt-net | 07076c33-b857-44d7-8e75-45984b5eb754 192.168.101.0/27 |
       |                                      |                  | 8a64235d-dd68-41ee-a3cf-4b387267a0a2 192.168.101.32/27 |
       |                                      |                  | 12458c7e-4c41-41e7-b681-91552009fd1b 192.168.101.64/27 |
       |                                      |                  | ab3129a5-7c72-426c-a0b9-d71613129420 10.101.1.0/27    |
       |                                      |                  | 7848caf5-86b3-4ec9-a6b3-b6472fd5f4e2 10.101.1.32/27   |
       |                                      |                  | 84b59689-d0dc-4bce-ae56-7945ce10ca60 10.101.1.64/27   |
       | 41204b4e-4a2b-4ba7-844d-6508d02acc8d | tenant2-mgmt-net | 59750efc-fa2e-4c48-a941-fbe40023d3f4 192.168.201.0/27 |
       |                                      |                  | 72e52d21-fedf-4fe0-be36-cc29b1d8680b 192.168.201.32/27 |
       |                                      |                  | 4b67c09a-e3e6-4b41-aa2a-487b5295638a 192.168.201.64/27 |
       |                                      |                  | 62f31f95-0a5b-43da-b75d-8471a61229cb 10.201.1.0/27    |
       |                                      |                  | 93e6a95b-faa9-4eec-948b-513761bd1d5e 10.201.1.32/27   |
       |                                      |                  | 91887d02-f570-473b-b914-5103d6a01e7c 10.201.1.64/27   |
       | 6a241562-75cc-449c-aa02-d7aaaad8e403 | tenant1-net0     | c629dbb3-3280-44ab-90de-ba2a1cf24110 172.16.0.0/24    |
       | 83a6a6b5-4d4c-499e-b4d3-d43873a87010 | tenant2-net0     | 3feaf374-8520-4988-ab0a-94a27b4f1da8 172.18.0.0/24    |
       +--------------------------------------+------------------+-------------------------------------------------------+
```

In the above CLI command, a user requests the list of virtual networks and their subnets. The output of the command is formatted as a table. Behind the scene communication is complex (see Figure 27):

- **Step #1:** The CLI client checks his credentials with the authentication service (is called Keystone) and only after receiving a token (note the *X-Auth-Token: b03d88a418484ad0893e8bdec49c69b6* field in the detailed output bellow) it sends two requests to the service that manages networking (called Neutron).
- **Step #2:** CLI gets the list of networks from Neutron
- **Step #3:** CLI gets details about subnets (ip and netmask) of each network.

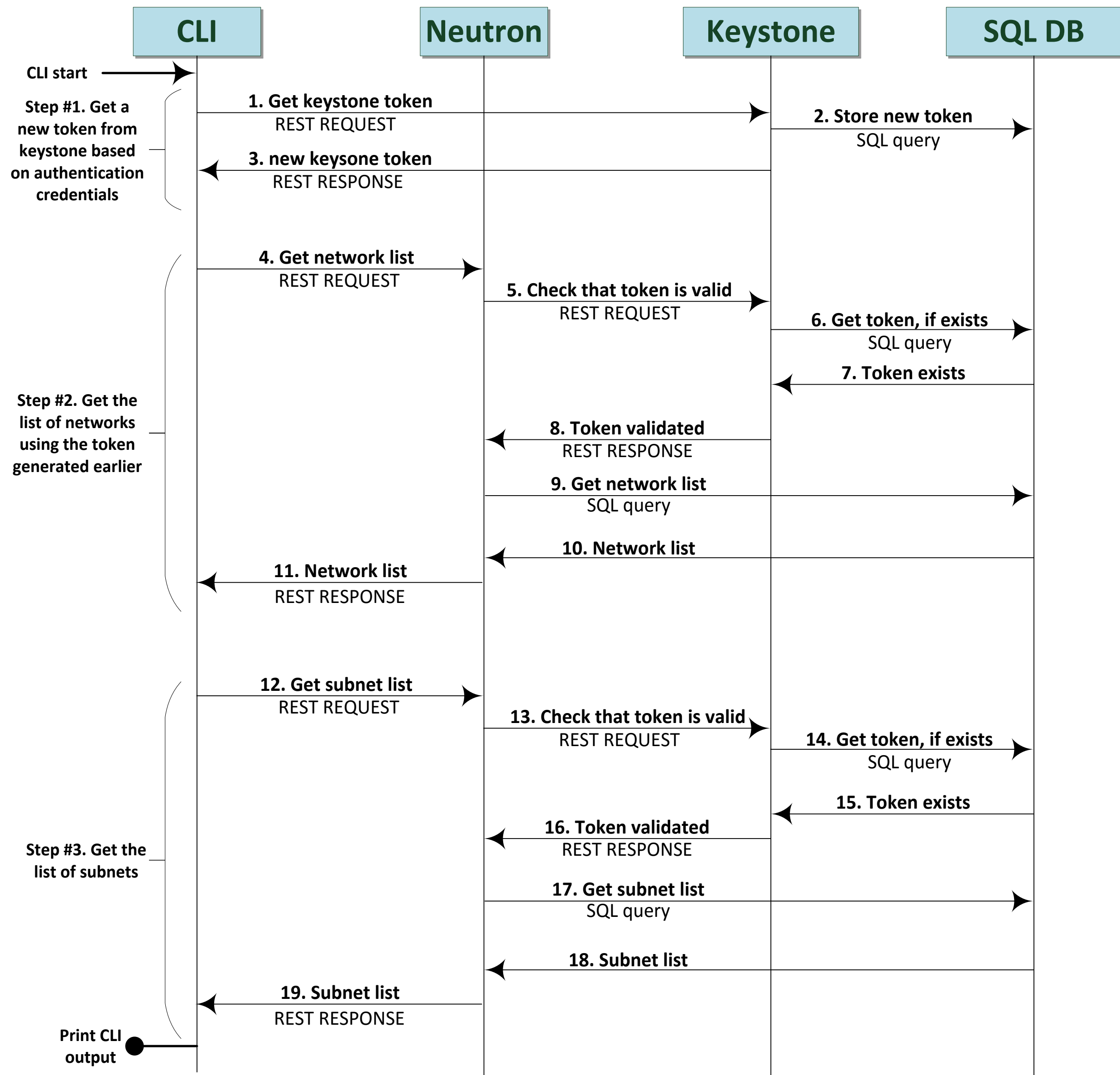


Figure 27: Neutron net-list sequence chart

Data transmitted is in HTTP format, and contains two requests (in `red`) and two responses (in `blue`). The payload is in JSON (JavaScript Object Notation) format. JSON is a human readable, open standard format used to transmit data objects of *attribute-value pair* defined in [73] [74] with support in many programming languages. In this example the CLI client runs on a terminal with IP address 192.168.204.91 an the Neutron server on 192.168.204.2 port 9696.

Request 1, from CLI client (192.168.204.91) to Neutron (192.168.204.2:9696):

```
GET /v2.0/networks.json HTTP/1.1
Host: 192.168.204.2:9696
Connection: keep-alive
X-Auth-Token: b03d88a418484ad0893e8bdec49c69b6
Accept-Encoding: gzip, deflate
Accept: application/json
User-Agent: python-neutronclient
```

The request is sent to host 192.168.204.2 on port 9696 and is querying the HTTP address /v2.0/networks.json the complete URL is: http:// 192.168.204.2:9696/v2.0/networks.json. And the response is a 200 OK with the network list in its payload. Both network and subnet resources are identified with a Universally unique identifier - UUID (e.g. e8a2f1f1-d230-470b-a3c5-d55e36975748 for subnet of *external-net0*).

Response to 1st request:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 3109
X-Openstack-Request-Id: req-800ea6b4-156a-4654-b633-690ddd73e66c
Date: Sat, 18 Jun 2016 03:44:03 GMT
Connection: keep-alive
```

{"networks": [{"status": "ACTIVE", "subnets": ["e8a2f1f1-d230-470b-a3c5-d55e36975748"], "wrs-tm:qos": "10e8bf1e-b392-475e-98f9-4db4dbe588dc", "provider:physical_network": "group0-ext0", "provider:network_type": "vlan", "id": "0f5c5504-3e0a-41c5-91a2-fb5fe16bd41b", "provider:segmentation_id": 10, "router:external": true, "name": "external-net0", "admin_state_up": true, "tenant_id": "6cf5bdcca24445f7ba3a9c140f8b28d8", "mtu": 1500, "vlan_transparent": false, "shared": true},

[… Cut 2220 characters …]

{"status": "ACTIVE", "subnets": ["3feaf374-8520-4988-ab0a-94a27b4f1da8"], "provider:physical_network": "group0-data1", "mtu": 1500, "id": "83a6a6b5-4d4c-499e-b4d3-d43873a87010", "provider:segmentation_id": 617, "router:external": false, "name": "tenant2-net0", "admin_state_up": true, "tenant_id": "56d80df0efe34364b05a6b6628190fba", "provider:network_type": "vlan", "vlan_transparent": false, "shared": false}]}

Request 2, from CLI client (192.168.204.91) to Neutron (192.168.204.2:9696):

```
GET /v2.0/subnets.json?fields=id&fields=cidr&id=e8a2f1f1-d230-470b-a3c5-d55e36975748&id=812fec68-022a-4919-9f76-d81793919efa&id=07076c33-b857-44d7-8e75-45984b5eb754&id=8a64235d-dd68-41ee-a3cf-4b387267a0a2&id=12458c7e-4c41-41e7-b681-91552009fd1b&id=ab3129a5-7c72-426c-a0b9-d71613129420&id=7848caf5-86b3-4ec9-a6b3-b6472fd5f4e2&id=84b59689-d0dc-4bce-ae56-7945ce10ca60&id=59750efc-fa2e-4c48-a941-fbe40023d3f4&id=72e52d21-fedf-4fe0-be36-cc29b1d8680b&id=4b67c09a-e3e6-4b41-aa2a-487b5295638a&id=62f31f95-0a5b-43da-b75d-8471a61229cb&id=93e6a95b-faa9-4eec-948b-513761bd1d5e&id=91887d02-f570-473b-b914-5103d6a01e7c&id=c629dbb3-3280-44ab-90de-ba2a1cf24110&id=3feaf374-8520-4988-ab0a-94a27b4f1da8
HTTP/1.1
Host: 192.168.204.2:9696
Connection: keep-alive
X-Auth-Token: b03d88a418484ad0893e8bdec49c69b6
Accept-Encoding: gzip, deflate
Accept: application/json
User-Agent: python-neutronclient
```

The second REST request queries details for the subnets received in the first message. Subnets are identiefied by UUIDs. The response arrives with network and mask for each UUID in the request.

Response to 2nd request:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 1206
X-Openstack-Request-Id: req-fb29cbb9-7e26-470a-b934-16a54cc42c61
Date: Sat, 18 Jun 2016 03:44:03 GMT
Connection: keep-alive
```

{"subnets": [{"cidr": "192.168.101.0/27", "id": "07076c33-b857-44d7-8e75-45984b5eb754"}, {"cidr": "192.168.101.32/27", "id": "8a64235d-dd68-41ee-a3cf-4b387267a0a2"}, {"cidr": "192.168.101.64/27",

[… Cut 718 characters …]

{"cidr": "172.16.0.0/24", "id": "c629dbb3-3280-44ab-90de-ba2a1cf24110"}, {"cidr": "172.18.0.0/24", "id": "3feaf374-8520-4988-ab0a-94a27b4f1da8"}, {"cidr": "192.168.1.0/24", "id": "e8a2f1f1-d230-470b-a3c5-d55e36975748"}, {"cidr": "10.0.1.0/24", "id": "812fec68-022a-4919-9f76-d81793919efa"}]}

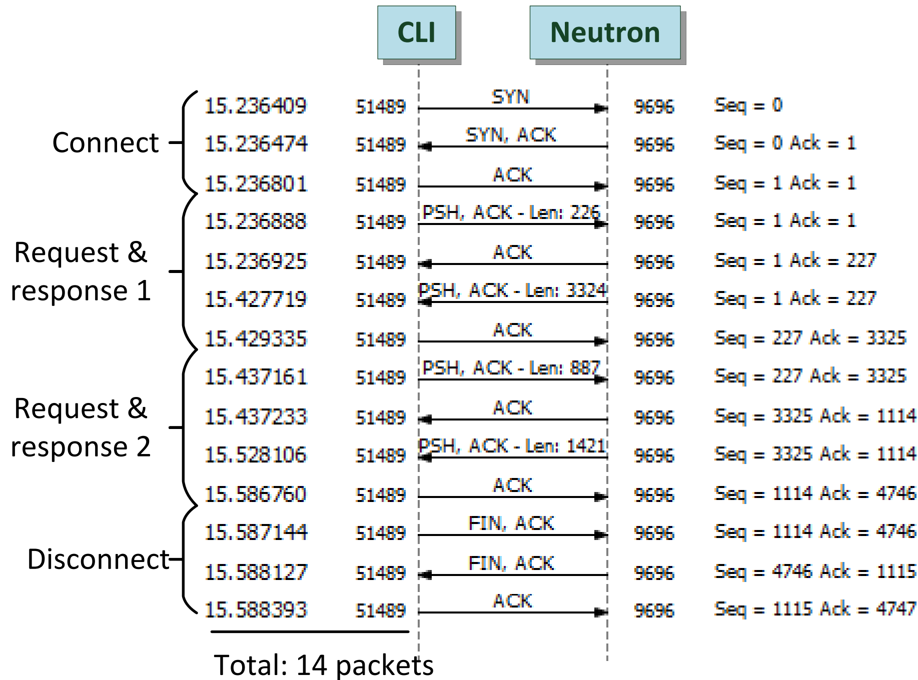


Figure 28: Openstack *neutron net-list* message sequence chart

The transaction message sequence chart shows that the TCP stream opens with our request and closes immediately after both requests have been completed, total duration is just 352ms (Figure 28).

Table 5: Openstack neutron net-list message count and sizes

Topic / Item	Count	Average	Min val	Max val	Rate (ms)	Percent
0-19	0	-	-	-	0.0000	0.00%
20-39	0	-	-	-	0.0000	0.00%
40-79	10	67.60	66	74	0.0284	71.43%
80-159	0	-	-	-	0.0000	0.00%
160-319	1	292.00	292	292	0.0028	7.14%
320-639	0	-	-	-	0.0000	0.00%
640-1279	1	953.00	953	953	0.0028	7.14%
1280-2559	1	1487.00	1487	1487	0.0028	7.14%
2560-5119	1	3390.00	3390	3390	0.0028	7.14%
5120 and greater	0	-	-	-	0.0000	0.00%
Total packets:	**14**	**485.57**	**66**	**3390**	**0.0398**	**100%**

Table 5 confirms the packet size distribution in [71] and [72]. Whith this confirmation and using the above list of classification criteria we can classify flows into:

1. By direction – east west traffic as it happens between services inside the data center, it can be north south if the client issues the request from outside.
2. By flow lifetime – short lifetime, less than 2s
3. By total bandwidth impact – mice flow
4. By transaction type – unicast query & response
5. By packet size – It conforms to the distribution in [71] and [72].
6. By application – HTTP traffic
7. By priority – soft real time or best effort - latency is important and a minimum level of throughput should be guaranteed if the application is critical.

3.5.3 Network topologies

As we saw in 3.3.1.3 network topologies are mainly influenced by capacity, scalability, cost and, to some extent, by cabling design. In 3.3.3 the hierarchical model is presented as a solution to these problems and, in theory, the model itself does not restrict a DC to a specific topology but in practice, due to cabling and scalability limitations of most topologies only variation of *Clos* network topology are used. From these variations two of them stands out: *Fat-tree* and *Leaf-spine* (Figure 29). These two are used on most data center cores as they better fulfill DC requirements.

Fat-tree is represented as a tree with the backbone connection at the root and servers as leafs. In its purest form (Figure 29-1) provides no oversubscription as, at each node, links connecting to their parent sum all of the capacity of the links connecting to their children in the tree. Be default it also provides no redundancy, if one node fails all of its children will be disconnected from the rest of the network. The tree height can be higher than the one exemplified.

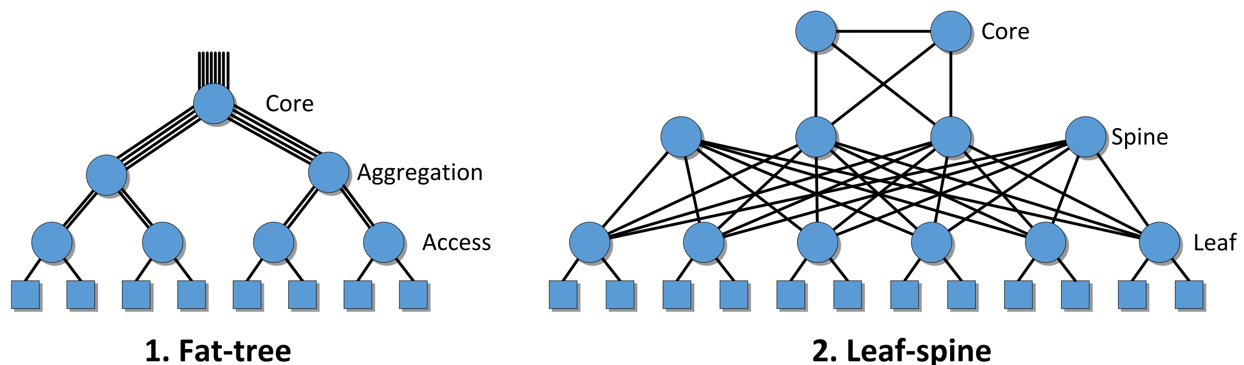


Figure 29: Most used *Clos* topologies

The diameter (longest distance between any two nodes calculated on the shortest path tree) depends on network height. The exemplified diagram has a diameter of 5 which means that a packet traveling from one far end of a network to the other need to traverse 5 nodes before reaching its destination therefore increasing packet latencies. This is usually the maximum accepted diameter of a data center core network as the delay introduced by crossing through too many host starts impacting application performance.

This topology is very good for *North-South* (i.e. traffic leaving the DC) traffic as it provides no oversubscription at Core level but can be worse for *East-west* (i.e. traffic between servers inside the DC) due to high diameter.

Providing no oversubscription with Fat-tree is only possible for a small number of nodes. On datacenters with hundreds of nodes it is impossible to achieve it as cost will increase dramatically as capacity multiplies as we go near its root. The more practical variant presented earlier at 3.3.3 leave space for oversubscription and provides redundancy (Figure 22).

The second most used topology, *Leaf-spine*, has a maximum height of 3, its layers are named *leaf*, *spine* and *core* and provides no oversubscription between *leaf* and *spine* (Figure 29-2). Its main characteristic is that each leaf switch connects to each spine switch (it its ideal form).

This topology is very good for East-west traffic as its diameter is 3. It is also resistant to *spine* failures as each leaf is connected to every spine in the topology. Therefore, multiple redundant paths between source and destination can be formed. *Leaf* failure resilience can easily be added by connecting each server to multiple leafs (see Figure 43 in 4.5 for a resilient leaf-spine example). Leaf-spine is advantaged in SDN context as multipath can be better used.

Except the two topologies presented above, in some Data Centers two more topologies are sometimes used: Torus and Hypercube (Figure 30).

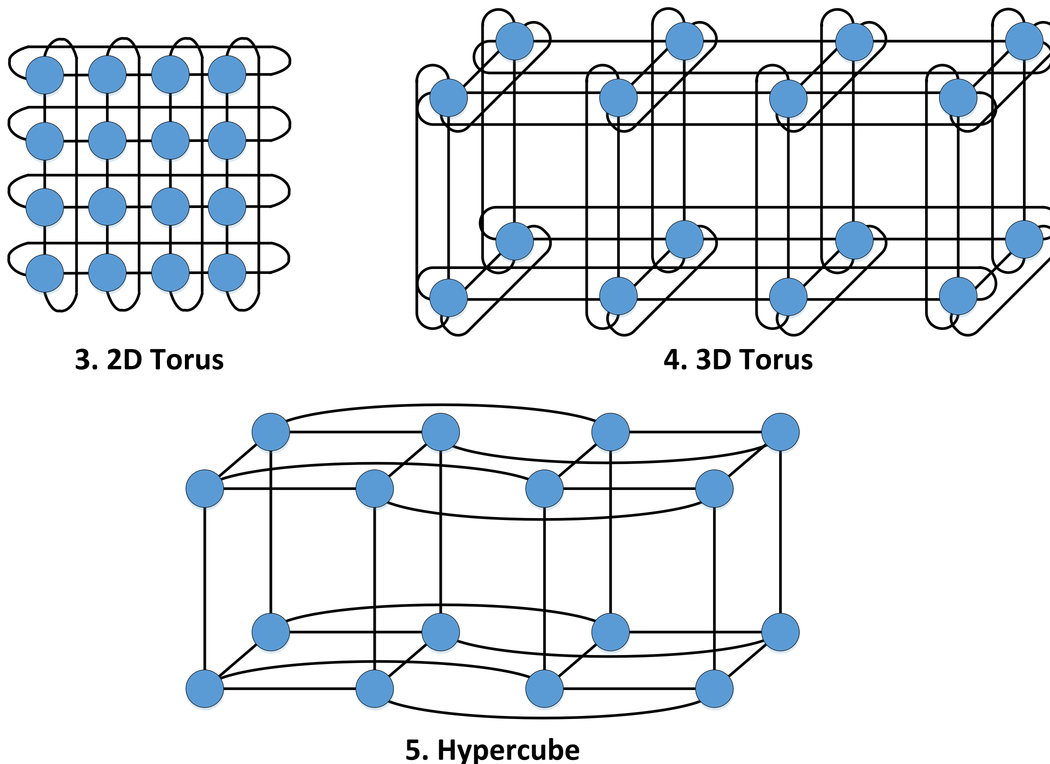


Figure 30: Torus and Hypercube network topologies

Both *torus* and *hypercube* are optimized for *east-west* traffic. They are usually used in application specific data-centers (e.g. HPC) for crunching large amounts of data (e.g. data analytics, MapReduce) from nearby nodes.

Their *diameter* increases with the number of nodes. For *torus* it increases exponentially while for *hypercube* increases linearly. Also, bisectional bandwidth is better for hypercube in topologies with more than 64 nodes. This are the results of our analysis from [75].

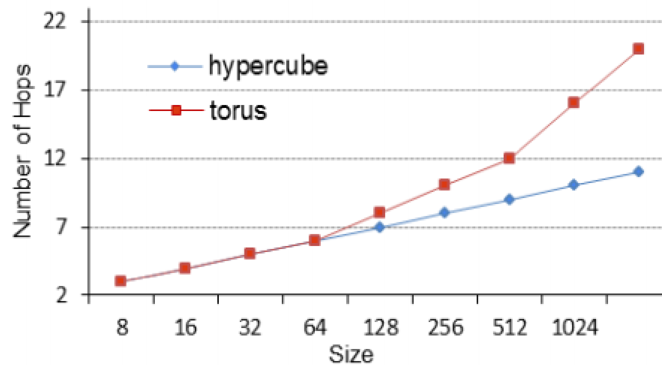


Figure 31: Network diameter of *3D torus* and *hypercube* [75]

Overall, when deploying topologies with many nodes (more than 64) *hypercube* has the advantage on *torus* both from performance and cost perspective yet, on topologies smaller than 64 nodes, *torus* is better as it has less links that hypercube therefore the cost of implementing it is smaller and performance at that number of nodes is similar.

These base topologies are the starting point of data center topology design. But they are adapted from data center to data center and from application to application. For example, for a data center dedicated to e-learning applications, the east-west traffic of a fat-tree topologies can be further improved by making connections between neighboring nodes. Of course, this improvement is a special case and resolves the problem of our application, therefore it cannot be generalized [PIST, 2014].

4 DESIGN AND IMPLEMENTATION OF THE CONTRON NS-3 OPENFLOW CONTROLLER

4.1 INTRODUCTION

To test new algorithms or concepts for SDN, the approach is to (1) either create real-life setups using OpenFlow switches or (2) emulated setups using virtual switches, such as Open vSwitch [31], with open source controllers. Both approaches provide good functional validation, yet they have some serious drawbacks. Using a real hardware setup is expensive and hard to scale, while using an emulated environment lacks performance fidelity as it provides no guarantee that a node ready to send a packet will schedule it promptly and also provides no guarantee that the switches will forward the packets at same rate [76]. For better and reproducible results, with large Data Centers or WAN setups, using a simulator can be a good solution.

The existing simulators such as Riverbed Modeler [35] [77], EstiNet [78] and NS-3 [79] provide support for OpenFlow yet lacks a simple integrated controller to provide the basic functionality needed for rapid prototyping. Even though connecting an external controller to a simulator is possible in most cases[36] it may be inappropriate for rapid prototyping due to high complexity of the controllers[37]. Another aspect to consider is the high cost, as both OPNET and EstiNet are commercial solutions while NS-3 is open source.

NS-3 is a mature discrete-event network simulator that provides necessary tools to create simulations. It primarily targets research and educational use. It has support for OpenFlow and has no real SDN controller although a module that provides a Layer 2 bridge implementation is provided. This module lacks any type of API support, adding and removing flows is currently done[38] by hand-crafting OpenFlow protocol packages and it is decentralized - each switch needs to have its own instance of this module. Also, connecting the OpenFlow switched to an external controller is not yet functional.

In this chapter we propose Contron, a new OpenFlow controller built from scratch, that provides the basic services for managing OpenFlow switches: topology model & discovery, flow management, statistics, processing packets received by the controller, shortest path routing and APIs such as: defining flow actions, matches and programming flows, reading and updating topology, sending and receiving event based notifications of topology changes etc. The controller is extensible, many services can easily be replaced with custom ones, and even more, custom controllers can be implemented by using only the desired services as the building blocks.

The implementation's main focus is on simulating SDN in Data Centers and WANs and does not consider SDN in wireless networks.

We begin by presenting several non-functional characteristics that were taken into consideration when designing and implementing *Contron* followed by a short architectural overview of it. After that, we go

[35] Previously known as OPNET Modeler
[36] NS-3 integration with an external controller is not yet functional
[37] Especially for OpenDaylight and ONOS
[38] As of NS-3 version 3.25

into the details of services by analyzing different aspects such as interaction with each other and with the entire system or highlighting their limitations. Further we present a possible use case of the model accompanied by simulator results. In the end we conclude with implementation status, its limitations and future work.

4.2 Requirements

As researchers, we want to validate our ideas and assumptions as fast and correct as possible. The time to reach a conclusion should be short, this way we can reduce the number of errors in the early design stages and avoid following wrong research paths.

Given the above, when we intend to implement a new SDN Controller, we focus on the following non-functional requirements:

- **Simplicity:** A simulator is simpler than a real world implementation, therefore is better for the researcher to spend his time on the problem he is trying to solve rather than on dealing with the complexity of emulated or hardware setups.
- **Extensibility:** New components should easily be created on top of the ones already implemented. Also, existing components should be extensible either by modifying their inner workings or by applying OOP concepts such as object associations and class inheritance.
- **Usability:** The services should provide a minimum set of functionality to make them usable and they should work in a default configuration, thus the developer focuses on his objectives not on fixing the model.
- **Minimality:** How a component operates is less important than what it does. This keeps the component simple and easy to use. More realism can be added later to certain components if they fall in the area of interest.
- **Loose coupled:** Other than keeping a strong dependency on the *Core* services and functionality provided by the simulator, controller services should be as independent from each other as possible. This allows them to be disabled if not needed, replaced or easily modified.
- **Reproducible:** Results should be consistent between multiple executions if the developer desires so.
- **Unit tested:** The execution routines of a simulator are independent of any external factors or hardware. Execution on one platform should yield the same results as the execution on another platform, therefore, in a simulator, Unit Tests can and should provide more coverage than those of a real system.
- **Documented:** The code itself needs to be well documented facilitating its understanding. Also, the concepts should be well presented with code samples and use cases to avoid any confusions.

4.3 Architecture

The architecture is structured on three layers: a *Core* layer providing functionality that other components depend on, a *Services* layer providing essential functionality that is useful yet not critical and an *Application* layer with its high level components, mostly independent from each other. Contron's architecture is modular and components can be disabled if not used. The core components are always

needed while the services can be cherry-picked based on the application needs. Contron's architecture with full set of services enabled is presented in Figure 32.

Contron depends on the NS-3 *Core* mechanisms and libraries. Separating the controller from NS-3 as a stand-alone SDN controller would increase its complexity, therefore our intention is to keep it tight-coupled with NS-3 *Core*.

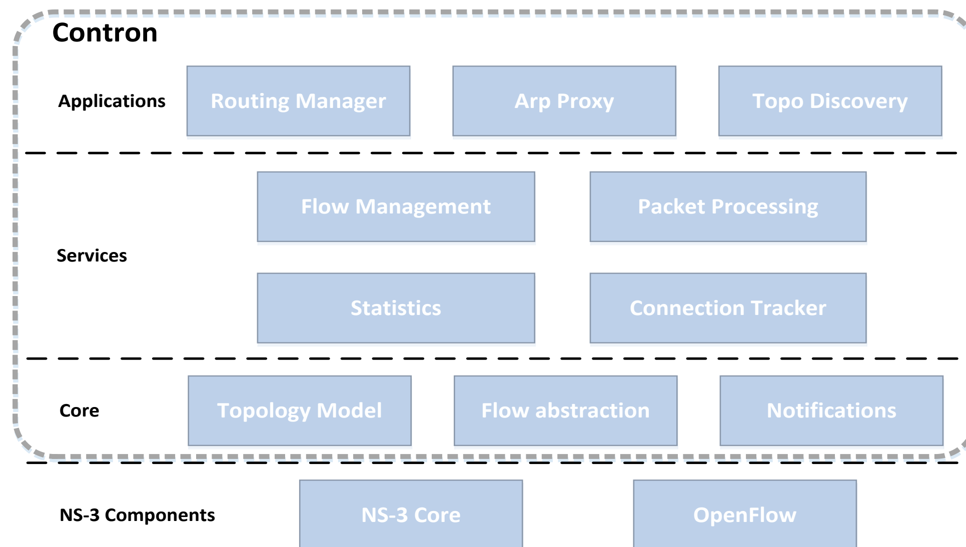


Figure 32: Contron's Architecture

Before going into the design details of each component we present a short functional description:

- **Notifications:** Applications and services need to react to changes in the global view of the network. Notifications are primarily used for topology and statistics updates that need to propagate to services which use them. For example, routes may need to be adjusted and flows updated when a topology change occurs.
- **Topology Model:** Represents the centralized view of the network. It can be seen as an in-memory database containing all switches and hosts properties, flows and statistics. This database provides accessors to the needed data, yet the data itself need to be populated by other services. The model also makes use of notifications to send topology updates to all of the interested services.
- **Flow Abstraction:** Used for defining flow entries and for adding or removing them from switch tables. For each flow, the user can define *Actions* and *Match*, the *Switch* where the flow is going to be added and some flow properties such as *Priority* or *Route ID*[39].
- **Flow Management:** Keeps track of all flow entries in the topology. Two categories of flows are supported: normal and management. The latter category is represented by those entries which need to be present in all switches. Their usual action is to either completely block a certain class of packets or to forward packets from the network management protocols, such as ARP or LLDP, to the controller for further processing. The management flows are added by services (e.g. ARP proxy adds a management flow to capture the ARP packets) and usually have a higher priority than normal ones so that protocol packets are processed first.

[39] *Route ID* can be defined only if the flow is part of a route

- **Statistics:** Updates the Topology Model with statistics gathered from all switches through the OpenFlow protocol or directly through NS-3 tracing mechanism, see [80] for more details. The component takes shortcuts when getting the statistics mostly because they are more important than the mechanism itself.
- **Packet Processing:** Packets received by the controller are classified and forwarded to services for processing. Each service receives packets for which it has been registered based on packet types.
- **Topology Discovery:** Provides a basic topology discovery service. It iterates all nodes defined in NS-3, identifying their type (i.e. switch or host) and connection to each other, while updating the topology model. The component does not implement real discovery, this is usually done through some external protocol[40].
- **Arp Proxy:** Provides proxy ARP support to hosts without the need to broadcast ARP requests through the network. It receives and responds to them on behalf of the hosts.
- **Connection Tracker:** Keeps track of all connections in the network. This is useful for a reactive approach. Connections are added when the first packet sent by the source arrives at the controller. This packet is analyzed and a unique entry is added based on its source and destination header fields. Notifications are sent when new connections are detected.
- **Shortest Path Routing:** Using Dijkstra algorithm it provides shortest path routes.

4.4 DESIGN DETAILS

Contron's design is mainly conditioned by features that NS-3 provides such as *Core* libraries, network interfaces, data channels, packet tracing, OpenFlow protocol and by the assumptions we made around the topology and types of devices in a data center or WAN SDN network.

From device type perspective, any SDN network topology is composed of OpenFlow *switches* and *hosts*. Mixed types – switch with IP stacks – conventional switches or routers are not supported by Contron. From NS-3 perspective, switches are *Node* objects aggregated with one or more *OpenFlowSwitchNetDevice* bridges[41]. Hosts are implemented as usual with a full L2/3/4 TCP/IP stack.

From the topology perspective, by looking at datacenters and WANs, we see switches, hosts, routers and special functions devices. Switches are connected with Ethernet cables (UTP or fiber optics) forming a graph at the network's core. Hosts are connected at the edge of core representing leafs of the graph. This configuration is the one supported by the controller. Special devices (i.e. Network Functions) are not supported by default, but can be added either by extending the implementation or by generalizing them as hosts or OpenFlow switches.

4.4.1 Topology model

The most important component of an SDN controller, represents the centralized view of the network. It can be seen as an in-memory database containing all switches and hosts properties, flows and statistics.

[40] Usually encapsulated in LLDP

[41] *OpenFlowSwitchNetDevice* is the NS-3 bridge model that provides support for Openflow

This database provides accessors to the needed data, yet the data itself need to be populated by other services. Any topology can be represented

In NS-3, complex topologies can be built by defining network nodes with different functions and connecting them together. Nodes are defined by aggregating multiple simulator objects (i.e. C++ objects, see [81]). The first needed object is the *Node* itself, then Network Interface Cards (NICs) are aggregated to it as *CsmaNetDevice* objects. After defining interfaces, nodes are specialized as OpenFlow switches if they get one *OpenFlowSwitchNetDevice* object or as hosts if they get the TCP/IP stack and applications for sending and receiving traffic. Interfaces are connected by channels – either full duplex CSMA or P2P serial links. Contron supports CSMA channels only[42]. A sample leaf-spine topology generated in NS-3 is presented in Figure 33[43]. The sample topology has 9 racks each with 5 servers and, on each servers, we have 3 VMs.

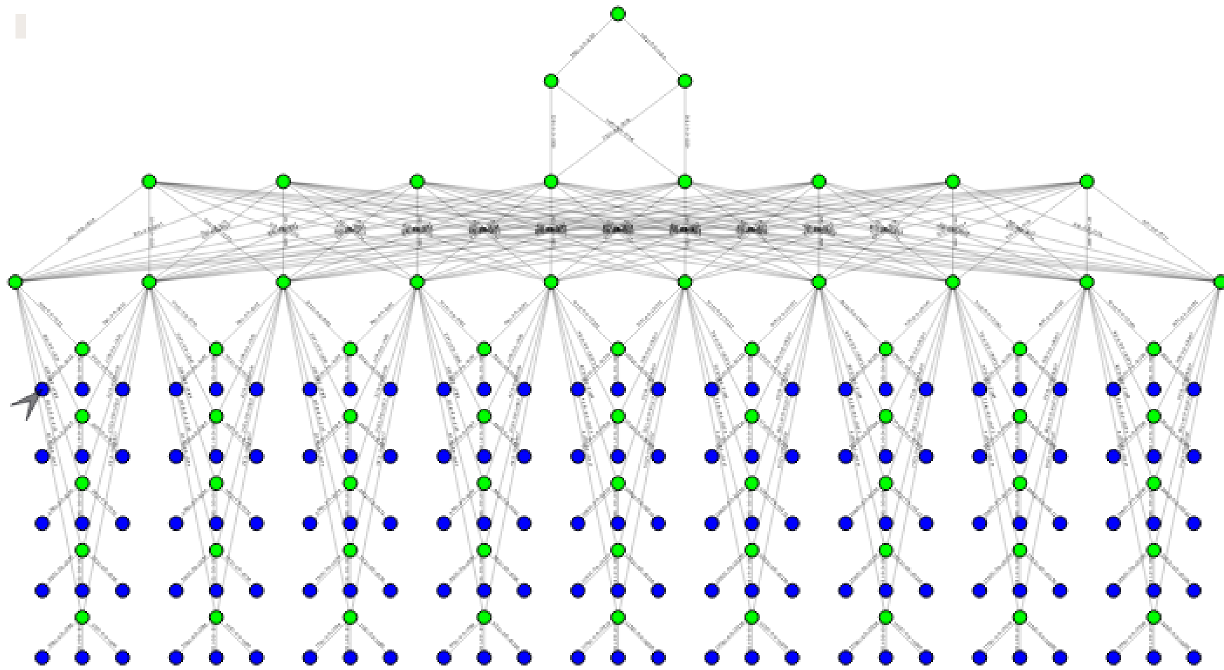


Figure 33: A leaf-spine in a Data Center with 9 racks and 5 servers per rack

The topology defined using the NS-3 *Core* APIs does not cover the needs of an SDN controller. It is true that most of the topology aspects used in SDN can be extracted from NS-3, nevertheless we still need to have it separated because:

1. the controller topology view may differ from the physical topology;
2. it is much too complex for our use cases and fails to meet the simplicity and extensibility requirements[44];

[42] Support for full-duplex *CsmaChannel* is under review and scheduled for 3.26, we used the patch under review.

[43] To note that NS-3 does not provide algorithms to create topologies. It provides *Node* objects. How these nodes are configured and connected has to be specified programmatically, either by listing nodes one by one or through an algorithm.

[44] The NS-3 APIs were not conceived for the type of runtime access needed by an SDN controller: rapid access to neighbors, getting link costs, reading packet counters and other device attributes.

3. because nodes are independent, NS-3 lacks a simple mechanism that can provide events and notifications on topology changes (e.g. when a link goes down), currently events are communicated through protocol messages but we need a centralized mechanism that can propagate events inside the controller.

The Topology Model implemented by Contron is a graph of *SdnNode* objects connected through *Port* objects (Figure 34). SDN Nodes can either be *switches*, identified by their Datapath ID or *hosts* identified by their MAC addresses[45]. The developer can navigate the graph either linearly by iterating through all of the nodes or by querying the list of neighbors and traversing the graph. Any topology can be represented this way.

Table 6: Topology Model Notifications

Type	Subtype	Description
Topology	ADD NODE	A *Node* is added
Topology	DEL NODE	A *Node* is deleted
Node	ADD PORT	A *Port* is added
Node	DEL PORT	A *Port* is deleted
Node	ADD STATS	Statistics are attached
Port	UPDATE LINKSTATUS	Link Up/Down event
Port	UPDATE LINKCOST	Link cost change
Port	SET NEIGHBOUR	A neighbor is connected or disconnected
Port	SET STATS	Stats are aggregated or updated

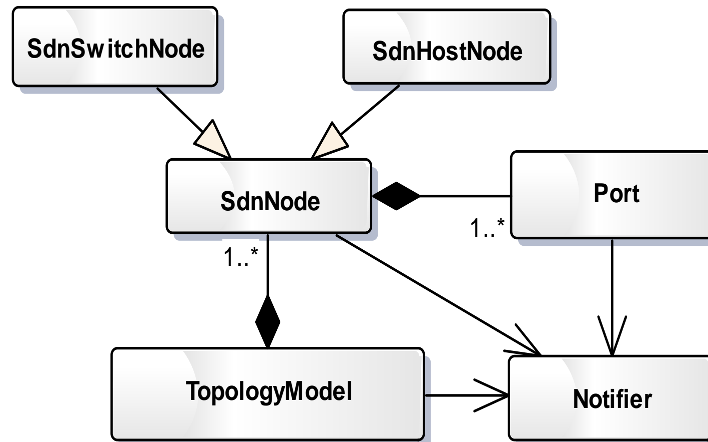


Figure 34: Topology model

Services and applications in SDN environments are expected to adapt rapidly to the network changes. For addressing this situation we implemented a simple and easy to use notification mechanism based on NS-

[45] For *SdnHosts* A single port is supported per NS-3 node. NS-3 nodes with multiple interface should instantiate an *SdnHostNode* object per interface - this matches other controllers behavior (e.g. Opendaylight)

3 callbacks and the observer pattern[46], see [82]. By using this mechanism, services are able to create new notifications or register to existing ones. Each notification is accessible by a *type*, usually the service name, and a *subtype*, the notification itself. The topology model exports the notifications shown in Table 1.

4.4.2 Simple Topology Discovery

Contron implements a simple discovery service that updates the SDN Topology Model by collecting information directly from topology's nodes defined in the NS-3 simulation[47]. The service can either process all of the nodes from the NS-3 topology, filtered by type (i.e. switches or hosts) or process a user defined subset of nodes. The service then identifies the types of nodes and creates or updates the *SdnSwitchNode* and *SdnHostNode* that corresponds to the NS-3 node (Figure 35).

Topology Discovery component provides the desired functionality – having the global view of the network – yet is implemented as a workaround to a full implementation. In real SDN systems the controller discovers connections between switches by sending control packets on all of the ports that are up, of the switches in the topology. The switches have a management flow that captures these packets and sends them back to the controller creating sort of a loop. As an example, if we have two switches, A and B, connected by a link, then the packet will be sent by the controller and received back by it. The packet takes the following path: *Controller → A → B → Controller*. These packets contain information that allows the controller to uniquely identify each packet if it arrives back[48]. This way the controller knows that the two switches are connected together by a link and can add this link to the topology or, if it already exists, updates its expiration counter. This packet is usually a LLDP message with custom TLVs[49] that the controller needs [83].

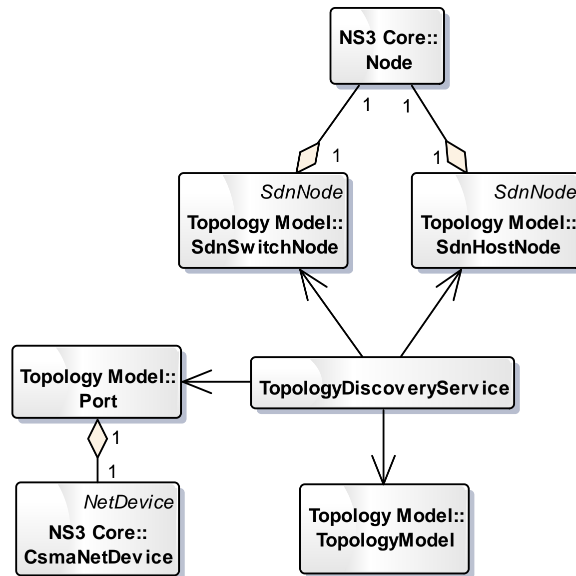


Figure 35: Topology Discovery

[46] Contron's Notifications are similar to listeners in Floodlight or Beacon
[47] NS-3 simulator keeps the topology as a list of *Node* objects defined at the beginning of simulation.
[48] Or at least determine the sender's port
[49] Type-legth-value datastructures are a way to encode variable datatypes in a packet payload

4.4.3 Flow abstraction and management

Flow entries is one of the most important concepts in the open SDN architecture [84] therefore, from a controller implementation perspective, providing a good abstraction of this concept is key to its overall usability.

The current version of NS-3 provides the OpenFlow protocol but without any abstraction. So, to create, modify or remove flows, users need to fill protocol data structures directly, which leads to complex and error prone development tasks.

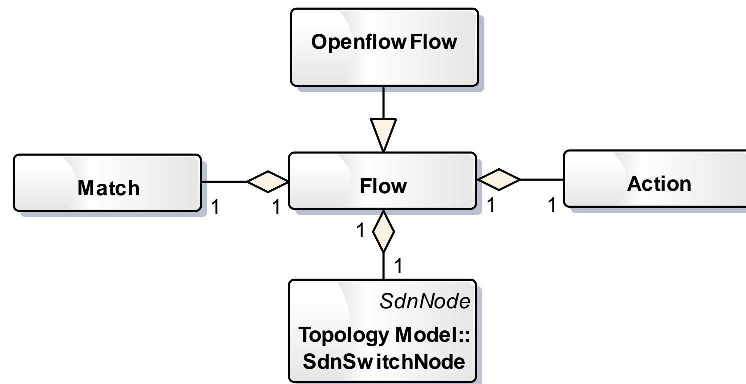


Figure 36: Flow Abstraction

The Flow Abstraction provides classes for *Match*, *Actions* and *Flow* to users (Figure 36). The data structures of matches and actions are in-line with NS-3 structures. To create a *Flow* the user first defines the *Match* then the *Actions*, and then he creates the *Flow*. Matches and actions can be reused from one flow to another. He can also specify priority per flow or its expiry time.

Flows are divided into normal and management flows. Normal flows are those that guide, filter and modify data traffic. They are managed by services or directly by the user (static flows). Management flows are those flows that forward certain types of packets to the controller or that block them. They need to be programmed in all of the switches within the network before normal flows and usually have a higher priority than them. For example, ARP or LLDP packets usually need to be forwarded to the controller for further processing.

Support for management flows is provided by Flow Service, which sets them when a new node is added to the topology. This service also keeps track of user flows (Figure 37).

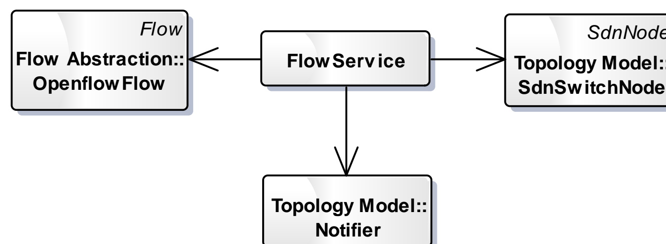


Figure 37: Flow Service

4.4.4 Statistics

The global view of the network includes statistics for nodes, ports, queues and flows. It is worth to mention that NS-3 does not have specific SDN statistics per node and currently supports only one queue per port, but it provides some basic infrastructures for printing them.

Given the above, Contron does provide *Port* and *Flow* statistics, since these are an important piece of information for decisions taken by applications and for system monitoring. Queue statistics are stored per port. For each counter we provides cumulative (i.e. total) and averaging per interval values.

For NS-3's users, one of the most important aspects is observing simulation outcome through plots, thus we have integrated basic support for Netanim and Gnuplot directly in Contron. Netanim's support for graphical representation is currently limited as each plot can show only a single data source from each node. There is no possibility to display data per port nor to allow users to select what data sources to appear on same plot representation. The output helps in forming a basic image on how a data source behaves over time but for better plots Contron uses *Gnuplot*. Compared to Netanim, *Gnuplot* output requires more CPU cycles to generate, therefore the number of data sources should be reduced to a minimum. For more information about the two applications check their homepages: [85] and [86].

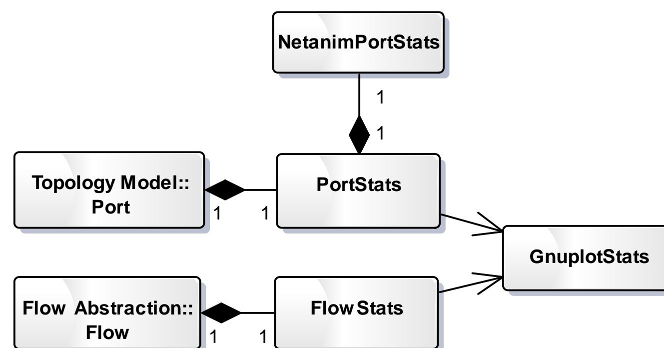


Figure 38: Statistics

The average is calculated with 10ms, 100ms and 1 second precision. The intervals are currently fixed, but the user can modify them by changing the controller's code. A more flexible approach that allows to modify the averaging interval through an API is planned.

Flow statistics are gathered from all switches and aggregated by the controller.

4.4.5 Packet Processing

The controller receives packets from OpenFlow switches. These packets have to be classified and forwarded to respective services or applications. This service converts a packet received from a switch into an *SdnPacket*, it classifies it based on header fields and then it checks the internal table for any service that has to process it further, otherwise it is dropped. This is how Contron's packet processing works.

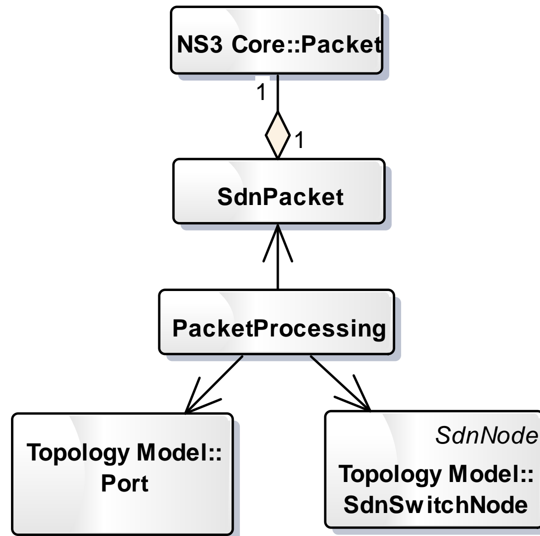


Figure 39: Packet Processing

Packet processing service also receives OpenFlow management packets, such as *FlowRemoved*[50] forwarded to Flow service or *StatsResponse* forwarded to Statistics Service for further processing.

Services register to Packet Processing module using a mechanism similar to that in Flow Abstraction. A *Match* and a *Callback* are passed as parameters at registration. On packet reception, the first step is to checks if it is an OpenFlow management packet and take immediate action[51], otherwise it checks if the packet hits one of the matches programmed by the services (Procedure 1). If no match is hit the packet is dropped.

Procedure 1 Process a packet

Input:

$p \leftarrow$ a new packet is received

$V \leftarrow$ a valid vector of (*Match, Callback*) tuples

Algorithm:

if $p \subset$ OpenFlow management packet

then

forward p to OpenFlow processing

else

for each $e \in V$ **do**

if p hits e_{Match} **then**

$e_{Callback}(p)$ {forward p to service}

end if

end for

end if

---

[50] Contron does not yet support full asynchronous functionality

[51] These actions are hardcodded

4.4.6 Proxying the ARP packets

ARP requests are sent by hosts trying to find the destination MAC address for an IP address. In classic networking these requests are broadcasted by switches on all of their ports creating short packet floods through the network[52]. A host receiving an ARP request asking for its address sends a response back which takes a unicast path back to the host that sent the request.

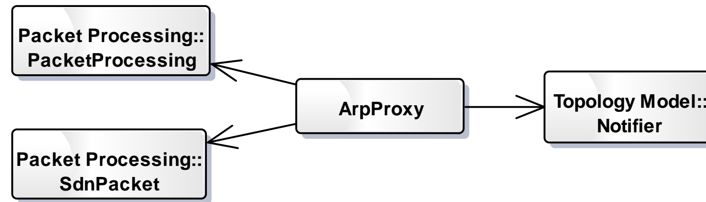


Figure 40: Arp Proxy

Also in SDN environments, ARP packets are used to learn where hosts are connected. For this, ARP requests are forwarded to the controller. When one host needs the destination address of another host it sends a request that is received by the controller; if the controller already knows the address, it responds on behalf of the destination, otherwise it sends the request to all of the hosts in the same network. In this case, after sending the request to all hosts, the controller waits for one of them to respond. After one of the hosts responds, it learns the address and then forwards it to the host that needed this address in the first place (Procedures 2 and 3).

Procedure 2 Process ARP requests

Input:

$r \leftarrow$ an ARP request received from Packet Processing

$V \leftarrow$ vector of ($s_{IP}, s_{MAC}, s_{node}, s_{port}$) tuples {$V$ keeps previously learned entries}

Algorithm:

($s_{IP}, s_{MAC}, s_{node}, s_{port}$) $\leftarrow$ EXTRACT-SRC-INFO(r)

$ip_{target} \leftarrow$ EXTRACT-TARGET-IP(r)

$e \leftarrow$ the reference to the tuple matching s_{IP} in V

if $e \notin V$ **then**

$V \leftarrow V \cup e$

else if $s_{MAC} \neq e_{MAC}$ **or** $s_{node} \neq e_{node}$ **or** $s_{port} \neq e_{port}$ **then**

$e \leftarrow (s_{IP}, s_{MAC}, s_{node}, s_{port})$

end if

$f \leftarrow$ the reference to tuple matching ip_{target} in V

if $f \notin V$ **then**

$arp_{response} \leftarrow$ CREATE-RESPONSE(f)

[52] If the destination address is not yet learned by switches

SEND-TO-HOST($arp_{response}$, s_{node}, s_{port})
else
for each *host* in *Topology* **do**
SEND-TO-HOST(r, $host_{node}$, $host_{port}$)
end for
end if

Table 7: ARP Proxy registered notifications

Type	Subtype	Description
Topology	DEL NODE	A *Node* is deleted
Node	DEL PORT	A *Port* is deleted
Port	UPDATE LINKSTATUS	Link Up/Down event

Procedure 3 Process ARP responses

Input:
$r \leftarrow$ an ARP response received from Packet Processing
$V \leftarrow$ vector of (s_{IP}, s_{MAC}, s_{node}, s_{port}) tuples
Algorithm:
(t_{IP}, t_{MAC}) $\leftarrow$ EXTRACT-TARGET-INFO(r)
$e \leftarrow$ the reference to the tuple matching t_{IP} in V
if $e \notin V$ **then**
return {may happen if entry is removed due to one of the events in Table 2}
else
SEND-TO-HOST(r, e_{node}, e_{port})
end if

When starting, ARP proxy handler sets a management flow that forwards ARP requests and responses to Contron. Then, once arrived at the controller, packets are identified by Packet Processing and forwarded to ARP proxy which then calls procedure 2 for requests or 3 for responses.

Currently, the proxy learns source and destination addresses only from ARP packets but other sources can be used: data packets that arrive at the controller contain source addresses and also, the topology model has a list of hosts that are connected.

Entries are removed from *V* when nodes or ports are deleted from the topology or a status for a link changes. These changes are notified by the events in Table 2. An ageing mechanism for entries is not implemented; this aspect may be subject for improvement. Timeouts of ARP entries are dependent on the packets that arrive at the controller so, if flows to forward these packets are provisioned in switches, no packet will arrive at the controller to reset the timeout and ARP entries will always expire. Also, other than ARP entry expiration, route entries themselves should expire if packets no longer flow through them.

Therefore the ARP ageing mechanism need to be tied to route expiration so that, as long as there are routes originating from a source node its ARP entry should be kept.

4.4.7 Connection Tracker

When a host starts sending packets to another host we can consider that a connection has been realized between those two. This information is then used by applications to make decisions - especially to determine if a route needs to be created or destroyed.

Besides unicast there are also multicast connections. Unicast connections track a single destination, while multicast connections track multiple destinations. The code treats these two types differently. Broadcast connections are considered a case of multicast but should be avoided as it increases the number of flows and the complexity of routing.

For simplicity, Contron only implements unicast IP packet tracking. The code can accommodate other protocols if desired along with multicast support which will be implemented in future work.

Connections are detected by inspecting packets received from Packet Processing service, therefore Connection Tracker service register itself to Packet Processing with a *Match* for all the IP packets and a *Callback* that gets called when a packet hits the *Match* (Figure 9).

Support for adding and removing connections manually is also important. An ageing mechanism for connections will be added in future work.

Connection tracker generates two notifications, see Table 8.

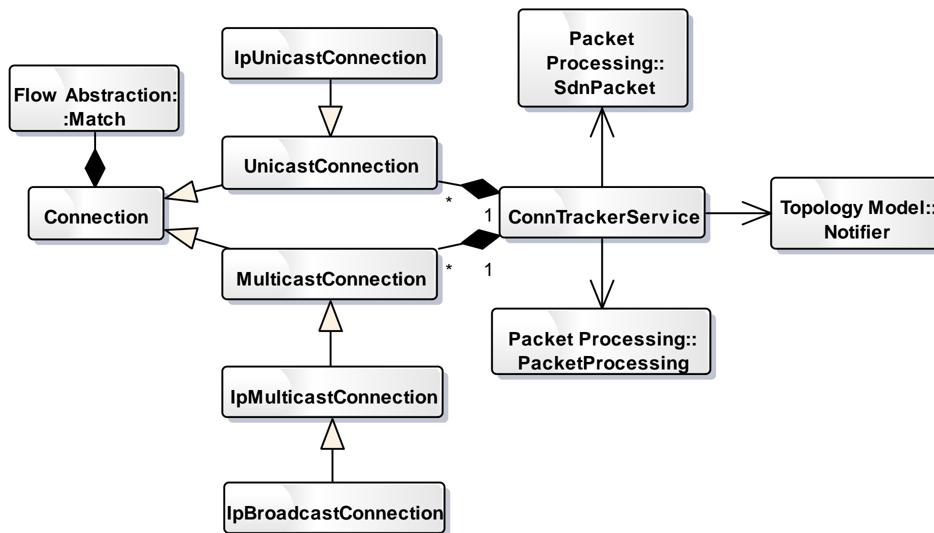


Figure 41: Connection Tracker

Table 8: Connection Tracker Notifications

Type	Subtype	Description
ConnTracker	ADD CONNECTION	A *Connection* is added
ConnTracker	DEL CONNECTION	A *Connection* is deleted

4.4.8 Shortest Path Routing

In SDN, for any two hosts to communicate, two routes need to be established between them before communication can take place: one route forwards traffic between source and destination and another one between destination and source.

A route is characterized by a *Match* that detects the traffic and an ordered set of *SdnSwitchNodes* that form a path through the topology.

Routes can be created proactively – users create routes by providing the paths – or reactively by creating paths when the controller receives the first packet from the source and is able to determine it as a new *Connection* (See section 4.7).

The Routing component is composed of a *RoutingManager* and one or more *RoutingAlgorithm* classes (Figure 47). The *RoutingManager* keeps track of routes and creates or destroys them using the algorithms. Developers can also access these algorithms directly to create the paths without needing a *RoutingManager*.

The *RoutingAlgorithm* that we implemented is a slightly modified version of the standard Dijkstra's algorithm. From a performance perspective it can easily be optimized, but the existing implementation provides desired results for current developing stage.

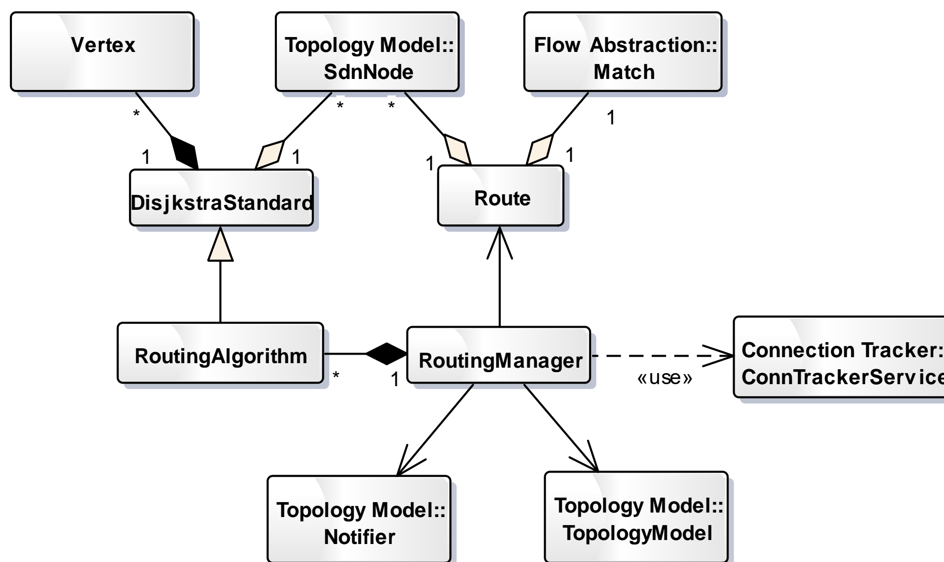


Figure 42: Routing

The original Dijkstra Shortest Path starts with all of the nodes in a priority queue with infinite cost. The cost decreases during execution. In our case we only store nodes with cost less than infinity. The issue is that the C++ STL *priority queue* does not support reordering, so we add nodes as we calculate their cost. Also, due to the same limitation, we add the same node to a queue multiple times, with different costs (we can't reorder and we can't remove values already added). This results in nodes being present multiple

times in the queue. To avoid processing them multiple times we added a *boolean* value to the *Vertex* to mark if the node was processed. If the node is found again in the queue and is processed we just skip it.

4.5 Case Study: Observing Queue Size and Traffic Profile at the Receiver on Congested Links When Routes Are Dynamically Updated

Let's suppose that we intend to validate a new SDN congestion aware routing mechanism when reading blocks of data from a Distributed File System. For this, we select one good Cloud Data Center topology, Leaf-spine, and we start simulating.

Let's also assume that, in our topology, we have three racks (R_1, R_2, R_3), each with three servers (S_{x1}, S_{x2}, S_{x3} where *x* is the rack number). Servers are connected to leaf Top Of Rack switches (Tor_1, Tor_2, Tor_3, Tor_4) by two links for redundancy and are running virtual machines (VM_1,VM_2,VM_3). To complete the leaf-spine Tor switches are connected to spines (Sp_1, Sp_2). To simplify things, all of the connections are 10Gbps links. The topology is depicted in Figure 43.

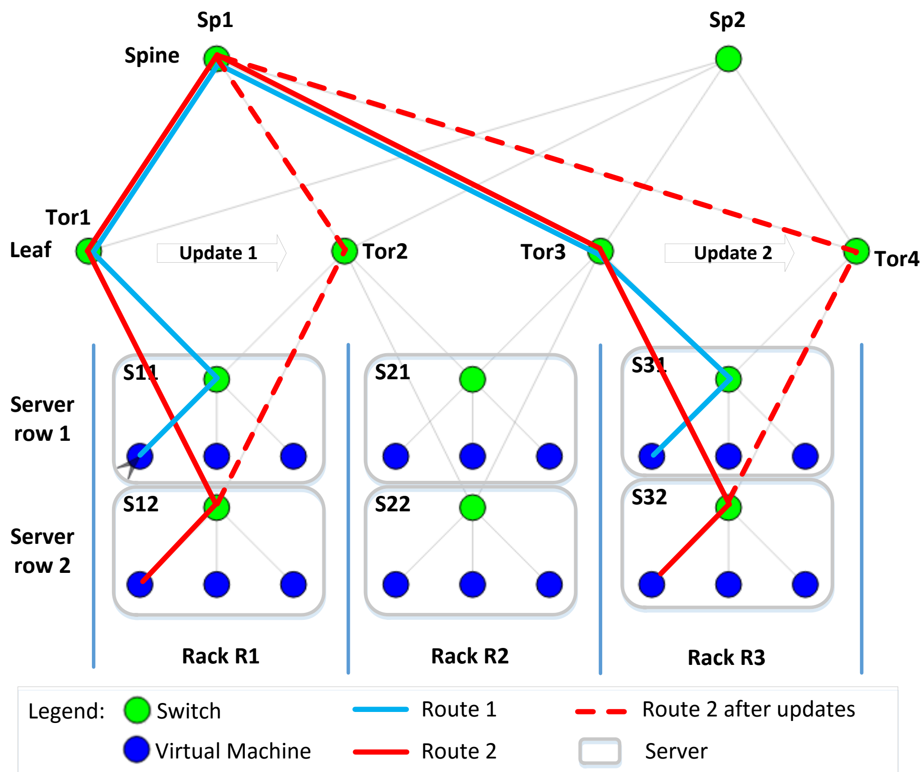


Figure 43 Case Study: Topology and Flow Routes

Sizing the port buffers is a complex issue [87] but we settle for a single small 1.43MB transmit queue (1500*B/packet* × 1000*packets*). The buffer size is a common value that 10Gbps switches have [88]. One of the reason for having a small queue is that speed mismatch between a high capacity core and a low capacity edge can cause a condition where a server sends small requests but the responses come from multiple sources at much greater speeds congesting its downlink. This massive speed mismatch can easily

lead to buffer (queue) exhaustion in a switch [89]. A common case is in distributed storage systems where requests for data sent by a single host are small but spread among multiple storage nodes.

Random Early detection (RED) queueing discipline [90] is enabled and configured to a minimum threshold of 70% and a maximum of 90% and REDs Exponential Moving Average (EMA) weight is set to aggressively follow the instant queue size. With the link operating at 10gbps, on a 1.43MB sized queue, the maximum waiting time of a packet is ≈$1.1ms$ when the queue is at 90%.

Data transfer is done over UDP, VMs in R_3 requests a block of 32MB of data from VMs in R_1 (Figure 43). Requests happen almost at the same time and the responses create congestions in the network. This traffic pattern is similar to that of Ethernet & IP storage infrastructures (e.g. FCoE) and Distributed File Systems protocols (e.g. NFS over UDP). We limited the simulation to 2 requests so that we can better observe the behavior of queues and impact of route changes on the traffic.

4.6 DESCRIPTION OF THE SIMULATION

The simulation starts with VM_1 of S_{11} responding to the request from VM_1 of S_{31}. We will call this $Flow_1$. Five milliseconds later VM_1 of S_{12} also starts responding to a request ($Flow_2$). The responses are sent at 7 Gbps which approximates to 700 MB/s of data, a value easily achieved by today's storage arrays (Figure 44).

As seen in Figure 43 the routes taken by the flows overlap on two links: $Tor_1 \rightarrow Sp_1$ and $Sp_1 \rightarrow Tor_3$. The summed throughput of the two flows is 14Gbps which is more than the capacity of a link (10Gbps). Therefore, the queue at Tor_1 starts to fill (Figure 44 *Right*) and link $Tor_1 \rightarrow Sp_1$ gets congested. The controller then detects that and reroutes $Flow_2$ around the congested spot: from $S_{12} \rightarrow Tor_1 \rightarrow Sp_1$ to $S_{12} \rightarrow Tor_2 \rightarrow Sp_1$ (Update 1 in Figure 43).

After the first route update (at t_1), another link ($Sp_1 \rightarrow Tor_3$) starts to congest and the controller reroutes $Flow_2$ a second time: from $Sp_1 \rightarrow Tor_3 \rightarrow S_{32}$ to $Sp_1 \rightarrow Tor_4 \rightarrow S_{32}$ (Update 2 in Figure 43 - in t_2 is detected and in t_3 rerouting is complete).

Switches report links congestion by sending OpenFlow[53] congestion notifications to the controller when the transmit queue usage goes above 50%. In our simulation, both Tor_1 and Sp_1 switches send congestion notifications. The controller reacts to the notifications by updating the cost of links ($Tor_1 \rightarrow Sp_1$ and $Sp_1 \rightarrow Tor_3$ respectively). Updating the cost generates an *UPDATE LINKCOST* notification which triggers Shortest Path Routing service into searching for better routes.

After finding a better route, the controller sends new set of flows to the OpenFlow switches: moment t_1 when the first link gets congested and t_3 when the second link gets congested.

[53] This is a custom NS3 implementation as OpenFlow does not support this

Table 9: Key simulation moments

#	Moment	Description
1	0ms	Start of 1st response
2	5ms	Start of 2nd response
3	t_0	1st congestion is detected
4	t_1	1st congestion is resolved
5	t_2	2nd congestion is detected
6	t_3	2nd congestion is resolved

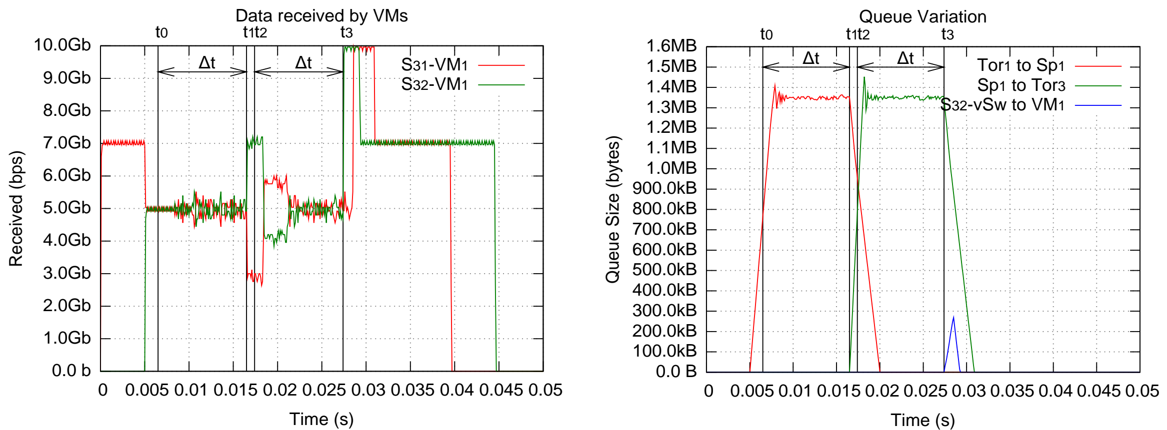


Figure 44 Left: Raw Data Received by the Destinations; Right: Queue Size Variation

The interval between a congestion notification is sent to be processed and a new route programmed into the switches was configured to 10*ms* (Δ*t*). In real Data Center deployments this interval is dependent on the controller processing speed and on the speed of flow updates in OpenFlow switches. To be more exact, configuring Δ*t* needs to take into consideration the following factors:

1. **Processing speed of the controller**, from the moment a congestion notification is received until the OpenFlow messages that updates the route are transmitted to the switches; this includes the processing time of our congestion aware algorithm;

2. **The time a switch needs to activate a flow into hardware** from the moment it receives the flow-add or flow-mod from the controller; this was measured to be between 1 and 5ms [91];

3. **Latency of the underlying control network**; this interval is negligible in most cases (i.e. bellow 1ms).

The processing speed depends on the controller used, the number of nodes in the topology and the number of requests queued by the controller (i.e. controller congestion). Previous works show that, with a high number of switches (more than 256) the processing time is between 2 to 10ms, depending on the controller, and increases linearly with the number of nodes. Extrapolating for more than 1000 nodes we can assume that processing can easily go above 10ms for all of the tested controllers in [92] (i.e. Pox, Ryu, Nox, Floodlight and Beacon). Also, tests with OpenDaylight controller show that, with many switches,

when storing information into its DataStore (i.e. its Topology Model), the processing time goes even higher than 20ms as shown in [93] and [94].

4.7 Result analysis and conclusions

Looking at the received data (Figure 44 *Left*) we see that, between 5ms and t_1, both VMs get 50% share of the congested link capacity. During this interval packets get dropped because $Tor_1 \rightarrow Sp_1$ queue fills.

At t_1, the first route update, an interesting effect occurs: the throughput received by $S_{32} - VM_1$ (i.e. $Flow_1$) suddenly goes to maximum while the throughput of $S_{31} - VM_1$ (i.e. $Flow_2$) drops! We would expect it to remain 50/50%. After $\approx 0.2ms$, even more interesting, the plot show a reverse, with S_{31} higher than S_{32}.

The above situation is caused by Sp_1 receiving packets at the same time from two sources:

1. at line rate (10Gbps) it receives old packets from Tor_1 that were already in the $Tor_1 \rightarrow Sp_1$ queue and
2. because of the route update it receives new packets via Tor_2, at 7Gbps. The packets in $Tor_1 \rightarrow Sp_1$ had a 50/50% ratio.

Therefore, as Tor_1 continues to transmit 5Gbps of $Flow_1$ traffic and 5Gbps of $Flow_2$ traffic (this is already in the congested queue, and is not dropped!), the new route updated by the controller will bring fresh traffic from S_{12} ($Flow_1$) via Tor_2 directly into $Sp_1 \rightarrow Tor_3$ queue at 7Gbps. The result is that the real throughput from S_{12} ($Flow_1$) is 12Gbps while the throughput from S_{11} ($Flow_2$) is still 5Gbps with a grand total of 17Gbps! This burst will practically fill the next link queue. Therefore, in the $Sp_1 \rightarrow Tor_3$ queue, the share each flow has changes to 70% for $Flow_2$ and 30% for $Flow_1$ matching the measurements in Figure 44 *Left*.

After a while ($\approx 20\mu s$), the 50/50% share traffic in $Sp_1 \rightarrow Tor_3$ queue left from before the migration ends but, till that happens, packets from $Flow_1$ arriving through the new migrated route will also accumulate on top of the existing packets in the queue. Therefore, when the 50/50% share ends, the queue will not be empty but contain packets from $Flow_1$. These $Flow_1$ packets are going to be transferred to Sp_1 at line rate which is higher than the 7Gbps rate received from $Flow_2$. This explains the reverse of flow shares in the traffic profile.

After 2nd reroute at t_3, both flows go to line rate for a short interval while queues are emptying. Also, there is a short burst in the transmit queue of S_{32} Virtual Switch to VM_1. In conclusion, if we look at the plots we can see that:

1. Solving congestion on one link tends to move it to the next link in the route.
2. Packets accumulates in congested queues and, when the flows are migrated, they cause short burst of traffic that fill the next queue in the path.
3. Packets arriving at destination through congested links are delayed due to the time they wait in the queues. At the same time, packets coming through rerouted paths arrive at the same destination faster. This creates a lot of unordered traffic which may cause problems for some protocols.
4. Control plane reaction time to congestions (Δt = $10ms$) is too high if our intention is to completely eliminate packet loss in burst conditions.

A solution to reduce Δ*t* could be to increase the responsiveness of the control plane to less than the interval it takes for a queue to fill to 90% after the congestion notification is sent. This gives a control plane less than 1-2ms to process a packet and update the routes. This can prove challenging in real conditions, especially for Data Centers with topologies composed of thousands of switches (see above how Δ*t* interval was chosen).

Another solution to the above problem could be to increase the size of the queues to accommodate for this delay. Therefore, the switches will need to have queues ten times bigger. This seems to solve the issue but, if there are speed mismatches in the network (e.g. going from a 10gbps to a 1gbps) and many topologies have that, then the 10ms queue buffer may lead to much higher packet delays at the low speed links and even more delays as congestion accumulation may be present. Virtual switches also make things worse as their queues are big (they use server RAM), yet throughput is variable as it depends on the processing speed of the server CPU and on its load. Therefore, the maximum packet delay would become in the order of hundreds of milliseconds which usually is unacceptable as it triggers retries by the higher level protocols. So, even if the network manages to eventually transport those packets to the destination, is already too late as the retry was already sent and the receiver just ignore them.

A more intelligent solution is to use a mechanism similar to QCN for real time feedbacks and combine it with congestion metrics similar to those we proposed in [PIST, 2015].

The unordered packets may be solved by intelligent queuing disciplines that drop older packets of the same flow at input. This is hard to achieve in hardware but controllers that are protocol aware may choose to drop packets that are old for protocols that are not resilient to high latencies.

The final conclusion of the case study is that, even if our brand new SDN congestion aware protocol seems to solve the problem, it still has a long way to go. And this was proven by having an OpenFlow controller inside a simulator.

4.8 IMPLEMENTATION STATE AND FUTURE WORK

The implementation of Contron in NS-3 emerged from our need to study network congestion in Data Center topologies and we added features as dictated by our day-to-day development needs (e.g. in [PIST, 2015]). Even though from a researching point of view our Controller is completed and proved to be useful in our own simulations, but in order to make it generally available, more work is needed especially testing and developing use cases that we did not need but are highly desired by other researchers.

The current code size is around 6KLOC we also have more than 50 Test Cases, half of them are Unit and the other half are Integration tests with complex scenarios. From an architectural perspective Contron is complete as we do not plan to add more components to it.

From an implementation perspective there is some work in the Flow Abstraction as not all the matches and actions have been implemented and tested (e.g. VLAN support). Implementing them is simple yet requires time for unit testing and they are not essential at this moment. Components that needs more work, and even some rework, are Packet Processing, Connection Tracker and ARP Proxy. The need for rework came as we gained experience and understood better the problems.

Also, since we are developing our models in *C++*, we decided to skip *Python* bindings for now.

One feature is ageing for Flows, Routes, Connections and ARP entries. We avoided this altogether as it needs coordination between multiple components. Adding proactive or reactive entries is straight forward, the problem is on how to detect when there is no traffic and remove Flows, Connections or ARP entries. First we need to implement it for flows and notify the controller if there was no counter increment (hence no packet arrived) in a configurable amount of time. The *FlowManager* will then have to notify the affected services of the timeout and they, in turn, should take actions – either keep the flow/connection/route or remove it. Also, notifications should be sent if, after sending the timeout notification, packets start flowing again. ARP proxy may need more than this as, even no packet arrives from a host for some time, that host may still be there, and hence the entry will still be valid.

4.9 Conclusions

Our main goal was to provide an NS-3 flexible simulation platform for a large variety of SDN research topics, including: OpenFlow protocol extensions, low level monitoring of traffic, understanding the limitations of SDN, creating routing protocols, and most important identifying and understanding some of the challenging problems of SDN such as: effect of flow update delays on traffic, effects of migrating flows on congestion and developing better routing algorithms than in traditional networking. Other than these topics, extensibility and ease of use creates the premises for expanding the use case domain to many of the current SDN research topics such as:

- **security** – dynamic threat mitigation, multi-tenant traffic isolation;
- **scalability** – to tens of thousands of node;
- **efficient load balancing** – load balance traffic through the controller;
- **integration with classic networking** – NS-3 already supports many of the standard protocols;
- **service-chaining** – create applications that manage simulated chains of services and optimize them;
- **quality of service** – add multiple queues with different priorities and identifying classes of service;
- **Network Function Virtualization** – by simulating Virtual Nodes with specific functions, migrating them to other nodes, and steering traffic through them.

Also, by running the same scenarios countless times, with minor changes, we observed that one advantage of our approach compared to real life or emulated scenarios is the possibility to reliably reproduce the same results for specific situation. Effect of any modification can be tracked and its root cause easily identify.

In the end, through the API abstractions, services for managing OpenFlow switches, extensibility and simplicity, our approach provides remarkable benefit for NS-3 users dedicated to the research of SDN technologies.

5 QCN-WFQR & SDN: MAXIMIZE NETWORK THROUGHPUT AND REDUCE PACKET DROP

Ethernet has become the primary network protocol used due to its undeniable advantages such as low cost, high speeds or ease of management. Being a best effort protocol, the IEEE Data Center Bridging Task Group developed a series of Layer 2 enhancements including Quantized Congestion Notification (QCN) [95] and enabled a lossless environment.

QCN is defined as a standard IEEE protocol for traditional networks yet its principles can be adapted to SDN and extended further by leveraging controller centralized model.

In the following sections the thesis proposes a method to maximize network throughput and create a lossless environment by reducing packet drop at the network core. This is achieved in three ways:

1. Reducing throughput at the source by monitoring queue size. This approach provides the same benefits as standard QCN in traditional networks and works in a similar way. The only difference is the presence of a controller that *may* enhance this functionality by making better decisions than in a decentralized model.
2. Distributing traffic workload between multiple servers hosting the same application. Load balancing can be based on link cost. Cost is factor of multiple parameters such as link capacity, throughput over different interval and on congestion measurements. Making this kind of decisions requires monitoring and processing of link parameters, including congestion measurements, in a central location. In traditional networking decision requires specialized custom applications that gathers data from switches and present them to applications and protocols. Practically they centralized the data so that it can be used to distribute workload. In SDN, as it already provides a central location, the custom applications leverage existing mechanisms. E.g. northbound/southbound interfaces and Controller's APIs to gather congestion data and centralize it.
3. Migrating already established flows to alternate and less congested paths thus reducing the need to slow down the traffic at the source. The method is well suited for a modern multitenant data center. This is a feature reserved to SDN.

Implementation for 1 and 3 was done in NS-3 and simulated using Contron (presented in Chapter 4).

The proposal is based on QCN Weighted Flow Queue Ranking (QCN-WFQR), work we previously presented in [PIST, 2015]. QCN-WFQR algorithm provides several congestion indicatives, per node (local) and system related, based on which decisions can be taken to achieve a better balanced traffic load in a congestion aware network while having an overall system increased performance.

Focus of this thesis is on adapting QCN from standard networking to SDN networks. Therefore the first two sections (5.1 and 5.2) presents a short introduction to QCN and QCN-WFQR indicatives. While the following sections provide more in-depth details on the SDN proposal and simulation results.

More details on QCN-WFQR congestion indicatives and applications to standard networking are presented in [PIST, 2015].

5.1 QCN: Quantized Congestion Notification

QCN is an end to end congestion management defined in IEEE 802.1.Qau [95] with the purpose of ensuring that congestion is controlled from the sending device, through the network, to the receiving device.

I/O protocols, such as SCSI, do not have contention or retransmission support and it requires a lossless transmission environment, like Fibre Channel – a high speed, low latency and lossless network. Ethernet by design is a best effort communication environment and with IP protocol it provides an end-to-end network for reliable transport protocols, such as TCP. In absence of reliable protocols, Ethernet has been enriched with a set of enhancements (Data Center Bridging – DCB) that enabled a lossless medium. Also, Ethernet became a viable solution due to supported high speeds (up to 10Gbps, 100Gbps, 400Gbps or even new Intel's 800Gbps, while FC supports up to 2, 4, 8, 16 or 32Gbps just arriving), lower capital-costs and ease of management. One of the protocols in DCB is QCN.

QCN monitors the transmit queue (buffer) on a switch port and, if the usage is above a specified limit, it sends a feedback to the traffic sources to slow down transmission so that port queues avoids getting congested. Without this mechanisms the queue usage will increase and packets will get dropped[54]. With SDN, this congestion information can be centralized and used to load balance between multiple providers of a service and to reroute existing flows to alternate and less congested paths thus reducing the need to slow down the traffic at the source. The main problem with this approach is that the queue usage statistic varies wildly in time and, in order to keep this information relevant for any decision it needs to either be transmitted to the controller at short intervals risking to overload it or to filter and condense the information at the source before forwarding it to the controller.

In standard networking, in order to reduce or even eliminate packet loss QCN employs two mechanisms:

1. **TX queue monitoring and feedback transmission in switches.** Switches monitor their TX queues for each port and, if a defined threshold is exceeded (e.g. 60%), then notification messages are sent back to the hosts that have packets on those congested queues to slow down their transmission. The notification messages (Congestion Notification Messages – CNMs) are plain Ethernet packets that contain in their payload a value indicating by how much the defined threshold was exceeded (called feedback or Fb). The standard calls these switch nodes Congestion Points (CP). Queue monitoring is done either automatically by the ASIC which notifies the CPU through an interrupt or, in case interrupts are not available, periodically pooling queue sizes (sampling) from the CPU (e.g. once every 1 ms or once every 100 frames depending on ASIC capabilities) and generating notifications messages when queue size is exceeded.
2. **A rate limiter implemented at the packet source.** Servers (CPs) implement a rate limiter that reduces throughput once a notification messages (CNM) has been received from a switch (from a CP). Throughput reductions takes into account the value received in the notification message. Nodes that implement rate limiters are known as Reaction Points (RPs).

The mechanism is similar to that of a *feedback loop* [96].

[54] Queues are small in data center switches as shown in section 4.6 and by empiric measurements & extrapolations in [96].

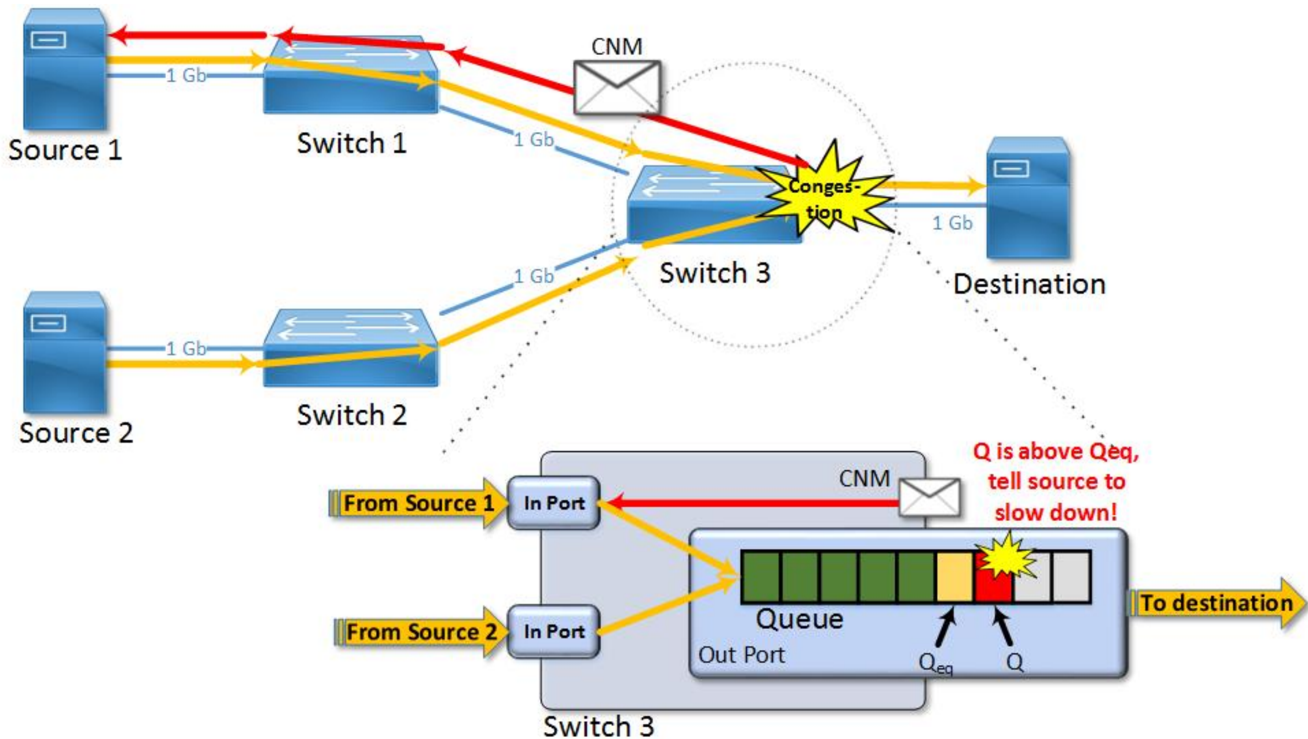


Figure 45: Standard QCN mechanism

In Figure 45 two traffic sources send traffic to a destination through three switches. All connections are 1Gb and traffic is sent at link maximum speed, therefore the transmit queue to the destination in switch 3 starts filling with packets because we try to squeeze 2 Gb of traffic on a 1 Gb link. The system has QCN enabled. After the transmit queue above a specified limit (Q_{eq}) a congestion notification (CNM) is sent back to one of the sources[55] indicating it to slow down traffic.

Switches (Congestion Points) compute feedback indicatives by combining the first derivative of queue utilization (i.e. rate excess) with the instantaneous queue utilization against a considered equilibrium (i.e. queue size excess):

$$F_b = -(Q_{offset} + \omega Q_\delta) \tag{5.1}$$

where $Q_{offset} = Q - Q_{equilibrium}$, $Q_\delta = Q - Q_{old}$ and ω is a constant (taken to be 2 for baseline implementation).

Congestion is considered for negatives Fb values. The probability with which a congestion message is sent to the source is reflected by sampling cycles (e.g. 150KB/sampling cycle (i.e. 100frames/sampling-cycle) reflects a 1% sampling probability).

Reaction points (e.g. *Source 1* in Figure 45) implements rate limiters that reacts to CNM feedbacks. Each RP rate limiter has two important parameters: *targeted rate* (TR) represents the throughput imposed by the rate limiter just *before* a CNM is received and *current rates* (CRs) represents the throughput at any time. CR can be limited by TR or may be lower than that.

QCN rate limiter has 2 phases as presented by [97]:

[55] CNM is sent to the source of the last packet found in the queue.

1. **Rate decrease -** when an FB is received:

$$TR = CR, and\ CR = CR * (1 - G_d|F_b|) \quad (5.2)$$

2. **Rate increase -** with 3 sub phases:
    a. **Fast recovery (FA): R=0.** Throughput recover slowly trying to avoid any new congestion.
    b. **Active increase (AI): R=5Mbps.** Throughput recover slowly faster – algorithm is more confident that probability of congestion has dropped.
    c. **Hyper-active increase (HAI): R=50Mbps.** Throughput recover to maximum, algorithm is confident that congestion will not happen.

Parameters for all three phases are computed using the same formula:

$$TR = TR + R, and\ CR = \frac{1}{2}(CR + TR) \quad (5.3)$$

The rate limiter steps (i.e. TR and CR computation cycle) are controlled by a *byte counter* or a *timer* if bytes received during an interval is higher than a defined threshold. The *timer* counts 5 steps of T (T = 10ms for baseline simulation) in fast recovery and $\frac{T}{2}$ in *active increase* and *hyper-active increase* [95].

5.2 CONGESTION INDICATIVES IN QCN-WFQR

WFQR stands for Weighted Flow Queue Ranking and represent a set of *indicatives* that present raw congestion information into a compact format better suited for congestion based decisions.

In SDN networks the following congestion indicatives from [PIST, 2015] are used:

Local indicatives – characterize a single switch:

1. Queue rank (R_{queue}) –the number of flows in a queue:

$$R_{queue} = COUNT\langle flows\rangle \quad (5.4)$$

2. Flow share ($Share_{flow}$) – a congestion measure of a flow in a queue:

$$Share_{flow} = F_{b:EMA_t} \times R_{queue} \quad (5.5)$$

where $F_{b:EMA_t}$ represents the average of the feedback values transmitted in CNMs calculated using an Exponential Moving Average function:

$$EMA_t = \alpha_t Y_t + (1 - \alpha_t)EMA_{t-1}, \text{ and} \quad (5.6)$$

$$\alpha_t = 1 - e^{-\frac{\Delta t}{T}}, \text{ therefore} \quad (5.7)$$

$$F_{b:EMA_t} = \begin{cases} \left(1 - e^{-\frac{\Delta t}{T}}\right) \cdot F_b + e^{-\frac{\Delta t}{T}} \cdot F_{b:EMA_{t-1}}, F_b \in flow \\ e^{-\frac{\Delta t}{T}} \cdot F_{b:EMA_{t-1}}, F_b \notin flow \\ 0, if\ t = 0 \end{cases} \quad (5.8)$$

where T is the measurement interval and Δt is the sampling interval;

System indicatives – computed at controller level.

3. Flow weight ($Weight_{flow}$) – a congestion measure of a flow from a system points of view.
   A flow weight is computed as the sum of flow shares received from all topology nodes, as shown below:

$$Weight_{flow} = \sum_{1}^{N-nodes} Share_{flow} \quad (5.9)$$

4. Reaction point weight ($Weight_{RP}$) – a congestion measure of an RP. A reaction point weight is computed as the sum of all flow weights of a specific reaction point, as follows:

$$Weight_{RP} = \sum_{1}^{N-flows/RP} Weight_{flow} \quad (5.10)$$

5.3 Quantized Congestion Notifications and SDN

As presented earlier, the source for information on link congestions is queue usage. Low usage means that there is no congestion while usage above a threshold (Q_{eq}) is considered a congestion. In QCN this information reaches the source only through CNMs and the source reacts to these messages by reducing throughput. In contrast, SDN has the advantage of a centralized controller, therefore congestion data may reach the source through different mechanisms or not reach it at all if controller decides to drop the notification. Therefore, our research proposes three communication models that can be applied to congestion management:

1. **Minimal integration** – QCN concepts integrate to the minimum with SDN. Switches send CNMs to sources through the data network, same way as in classic QCN, and store local indicatives. The difference is that indicatives are sent to controller on events, periodically or on request (Figure 46).
2. **Partial integration** - Switches forward CNMs to the controller through the management network and do not compute any local indicatives. All local & system indicatives are computed inside controllers. When a CNM is received controller forwards them to sources through a local, very fast, code path bypassing data path (Figure 47).
3. **Full integration** - Switches forward CNMs to the controller and the controller computes local & system indicatives. The controller do not forward any CNM to the source but directly acts on traffic by updating paths or configuring rate limiters (Figure 48).

These models are described in detail bellow.

5.3.1 Minimal integration model

This model is similar to standard QCN. CNMs keep their format and pass through the network unmodified. Servers (RPs) do not need any modifications as long as they support standard QCN but switches (CPs) have to keep tables with local congestion indicatives.

Controllers use local indicatives to compute system indicatives and to make decisions. Therefore they keep a *mirror* of the local indicatives stored in switches. For this controllers need to get them from switches. This can be done by three procedures:

1. **On events** – When a new congestion happens after long intervals of uncongested traffic.
2. **Periodic** – From time to time switches report congestion status. Reports happen fast on congested queues and slower on uncongested ones.
3. **On request** – Sometimes the controller may decide when to get a new set of congestion indicatives. For example on startup or when it estimates that traffic may spike on a port (e.g. on ports with many flows).

In this model, congestion indicatives arrive to the controller through OpenFlow using *new* queue and flow statistics protocol messages formats. Therefore some effort is spent defining and implementing these new OpenFlow extensions.

Also, controllers have to configure flow rules so that CNMs are able to travel back to their source. This may be an issue in multitenant, virtualized networks where communication happen through tunnels or multiple layers of indirection (e.g. MPLS, VxLAN or GRE) and determining the real source of a packet, with all of these extra headers, may be a challenge for a switch.

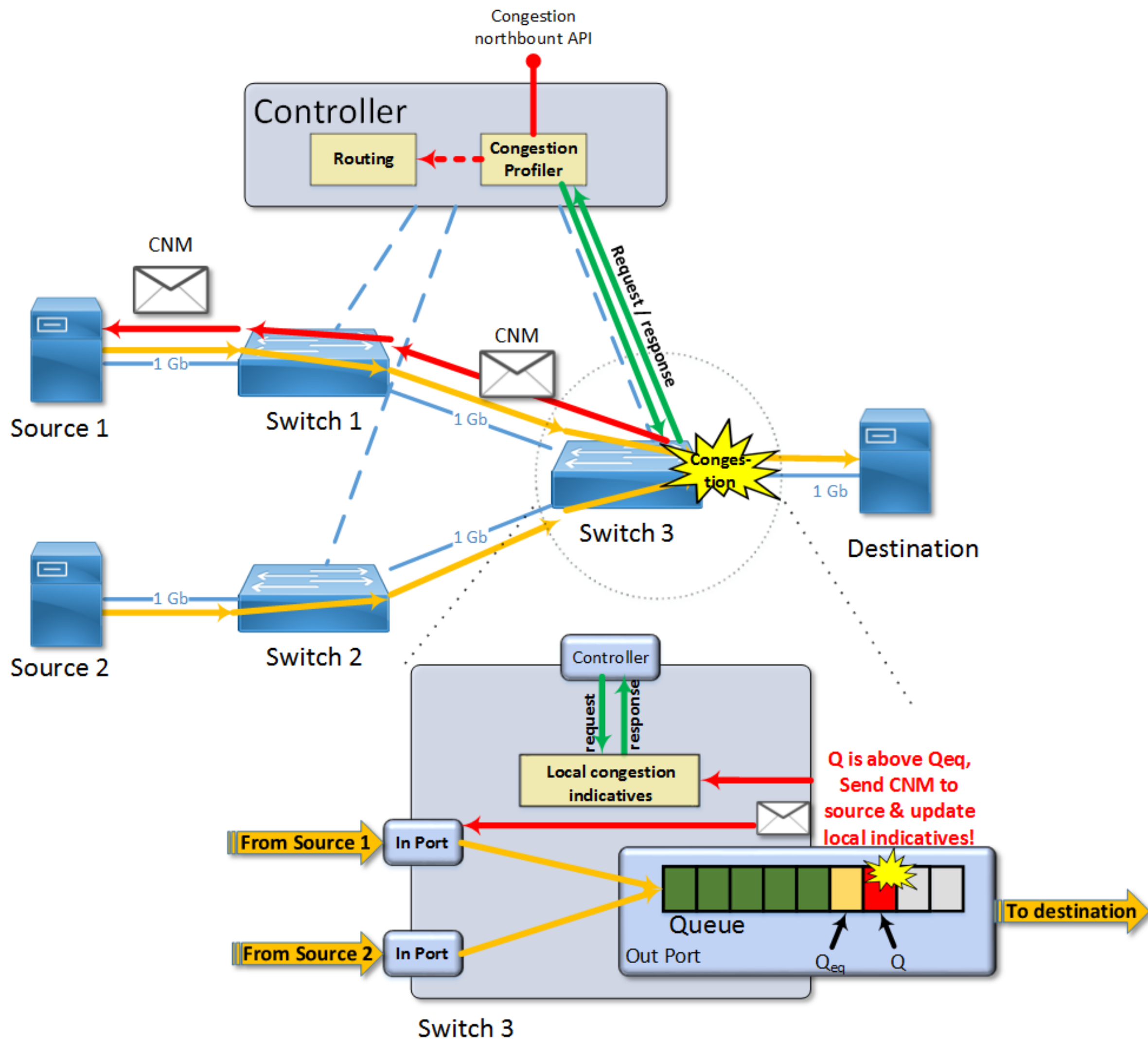


Figure 46: QCN and SDN – Minimal integration model

The SDN controllers needs a new service. We will call it the Congestion Profiler. This service has to be fast therefore it should be executed in the controller context not as a separate application. It role is to gather and centralize WFQR local indicatives from switches. Based on them it computes different system wide congestion indicatives which are then fed to a routing algorithm (e.g. Dijkstra's shortest path) which decides the path of each flow. The system indicatives are then filtered and exposed to applications through

a special *northbound* congestion API. Apps are interested in choosing the least congested server, therefore the API will present them with a list of servers (RPs) and their congestion. Apps may request the entire list or filter it by service type: usually an ID that uniquely identifies a service (a unique number or an UUID).

Congestion Profiler communicates with Routing either by modifying the cost of different links (i.e. congested ones have a higher cost) or, sometimes, by directly changing the routing table. An optimal approach would be for Congestion Profiler to update link cost and also generate an event to the Routing Service so that it can start rerouting flows. One such approach will be presented later in section 5.5.

This approach is simple and it has the advantage that it uses the same, time proven, mechanisms of QCN. Because QCN mechanism is still in place, delays in controller operation affect little the functionality of the system as Rate Limiting traffic will still happen regardless of controller decisions. Disadvantage is higher complexity of software in switches and the need to extend OpenFlow. This breaks compatibility with the OpenFlow standard and may not be desired.

5.3.2 Partial integration model

When compared with the minimal model the main difference is that CNMs are sent directly to the controller, therefore switches need minor modifications to their software. Local indicatives are no longer kept in switches but computed directly by the controller. Switches still need to monitor queues and generate CNMs. There is also no need to modify OpenFlow as CNMs can be transmitted as *packet-in* messages to the controller and back from the controller to servers (CPs) as *packet-out* messages.

This approach has multiple difficulties but each of them can be solved:

- CNMs need to reach back to the traffic source (i.e. servers or RPs) with minimal delay. Any delay will cause traffic to continue building up at switches (in CPs) and, due to the fact that transmit queues are small, packet loss will be highly probable (see Ch 4.5). This can be an issue as controllers have high internal delays due to long software processing paths, as shown earlier (see 4.6). Therefore, in order to mitigate this, a special, very fast, processing path has to be present at controller level so that CNMs can be forwarded back to the traffic source with the shortest delay possible. If this path does not exists then this communication model should be avoided.
- In case of network load, controller may be overloaded with CNMs at a rate that is too high for successful processing inside a single centralized controller. This may be mitigated with a distributed controller where packet forwarding components are on separate nodes and congestion profiler itself is divided into multiple components for improved parallelism. Of course this only means that a considerable effort has to be put into developing the profiler itself but effort may be worth as this will create a truly scalable system.
- CNMs are sent over the management network. If this network is physically separated from data network then considerable traffic may be sent over it as congestions generate short but considerable burst of CNMs that may choke a small capacity management network (e.g. a switch with many 100Gb ports may be connected by a small capacity 1Gb port to the controller). Luckily the usual configuration is to use management tunnels over data network therefore alleviating this issue.

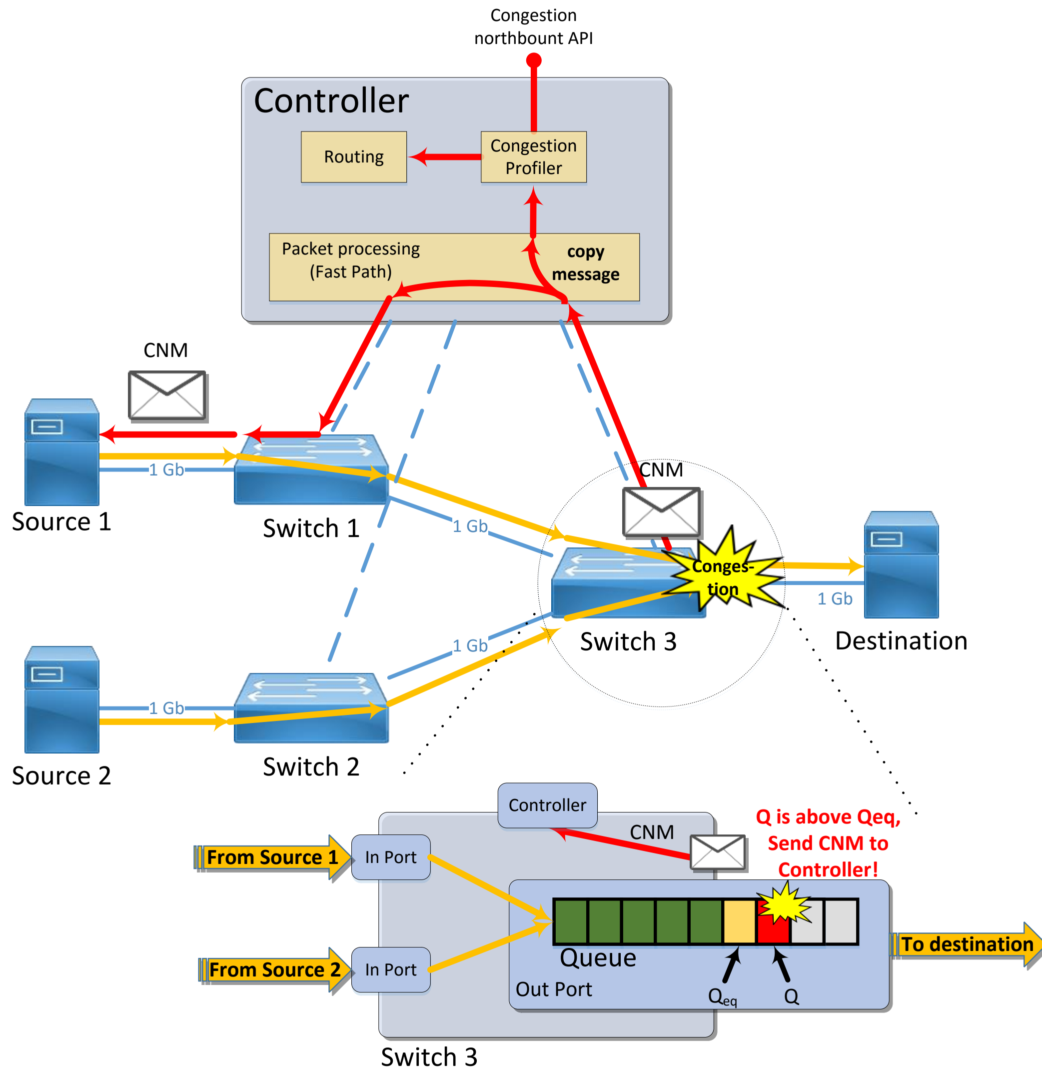


Figure 47: QCN and SDN – Partial integration model

5.3.3 Full integration model

Full integration goes further away from QCN to the point that its implementation according to standard is no longer required. QCN concepts remain valid yet compatibility with other solutions is no longer an issue. In this model CNMs[56] are sent from switches to controller. Congestion Profiler parses these packets, updates link costs, sends events to Routing and, if necessary, sets flow rate limit to the *edge switches* (edge CPs). Note that in previous solutions rate limiting is done at the *source* while in this model it is done in the first switch on the path of a packet (at *edge*).

If the routing service is able to migrate flows from congested path to less congested ones then setting a flow rate limit may not be necessary at all. Otherwise, how much to limit a flow rate is given in the feedback provided by the routing service back to Congestion Profile. For example, configuring the rate limiter of a flow in Virtual Switch 1 (Figure 48) is necessary:

[56] Still called CNMs even though packet format can be completely different from that of standard QCN.

1. if the routing service tells Congestion Profile that it was unable to migrate flows or
2. there is a small probability that the migrated flows will remove congestion on a link.

This solution has two major problems:

1. It only works when Edge switches are virtual switches. Our target is to obtain a lossless environment. Therefore, when limiting flow rates at edge arriving packets have to be kept in a very large queue. This queue can store hundreds if not gigabytes of traffic to eliminate packet loss and *hope* that higher level protocols at the source and destination compensates for slowing down traffic at edge. This kind of buffer is easily implemented in software and very complex to do in hardware. Moreover, slowing at *edge switch* is not a good idea due to delays (in the order of seconds) that may be added to packets due to high buffering. These may not be manageable by the higher level protocols. This problem is hard to fix but, sometimes, this model may only be the only one that can be applied. The reason is that, in virtual environments, network administrator may not always be able to install operating systems that have support for QCN, and therefore limiting traffic at *edge* instead of *source* (i.e. first hop after source) may be the only solution.
2. Packet processing is done entirely by the controller. In this case the controller needs to react fast. But, in case of high congestion, overloading it may be easy (e.g. Dijkstra is costly on large topologies). Creating a smart, distributed congestion profiler will alleviate this problem but creating a scalable solution for the entire datacenter is a challenge.

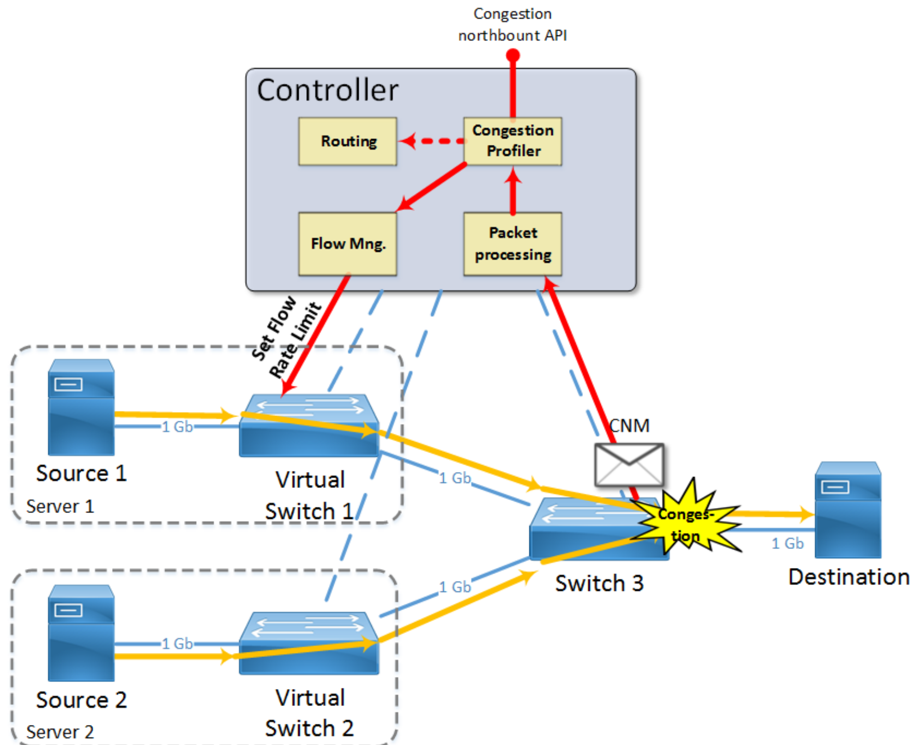


Figure 48: QCN and SDN – Full integration model

5.3.4 Conclusions

The main advantage of the last model, not relying on any implementation of the QCN standard is overshadowed by its problems. These make it impractical for generic data centers implementations but it may be the only option if VMs themselves do not support QCN rate limiters (VMs are the sources and not many virtual network interfaces provide QCN). Therefore, to work around this limitation this model may be combined with Minimal and/or Partial Integration. Both models would need to be implemented by the same controller but used only when needed:

1. Partial integration would be used for nodes that support QCN (and switch to minimal in case controller is overloaded with traffic);
2. Full integration would be used only as a last resort for nodes that do not support QCN.

5.4 Case study: Load Balancing based on congestion

The previous section presented models of implementing congestion management in SDN, this section will present an election algorithm for load-balancing traffic between multiple QCN Reaction Points RPs (i.e. servers) in a distributed file system (Ceph).

In distributed file systems, files are divided in data chunks (or objects) and replicated on multiple storage devices, therefore a chunk or an object can be transferred from any replica. For example in Ceph each object has a PG (Placement Group), associated to it. Each PG contains the list of storage devices (OSDs) that have that object (RPs from a QCN perspective). Therefore, using QCN-WFQR the replica may be chosen dynamically from the available OSDs in the PG thus improve system responsiveness by trying to avoid network bottlenecks. In this case storage devices are the reaction points (RPs).

The load balancing algorithm performance is directly influenced by the latency with which the controller gets the relevant congestion indicatives, decides and move flows while achieving a better balanced traffic within the system topology.

The controller design pays an important role and we distinguish two main directions:

- Centralized controllers – runs on single server;
- Distributed controllers – multiple entities cooperate to provide the functionality of the control plane. Global view of the network is still centralized but each entity partially manages it ([98] pp. 29).

Each method has its own pros and cons:

1. A centralized design has scaling limitation, bigger bottleneck probability, single point of failure, but it has a simpler complexity (basically it follows a multi-threaded design over SMP systems) and it has strong semantic consistency
2. A decentralized design scales up easy and meet performance requirements, handle better data plane resilience and scalability, fault tolerant, but it has a weak consistency semantics (and it is worth to mention that a strong consistency implies complicated implementation of synchronization algorithms) and it has much a much complex implementation ([98] pp. 30).

In case of a logically centralized controller a *system profiler* is needed for clients (RADOS clients) to get their data. The System Profiler is made of two modules, Figure 49:

1. A **Congestion Profiler** service presented in the previous sections (first in 5.3.1).
2. A **Service Manager** application unique to each service. Its role is to help clients go to the least congested RP. It can do this in two ways:
    a. by providing the entire list (or a filtered number) of RPs to clients and let them decide.
    b. by acting as a load-balancer - instructing the controller to forward traffic to the best RP (e.g. load balancing http traffic). This method is transparent to the client as it is not aware of service manager existence.

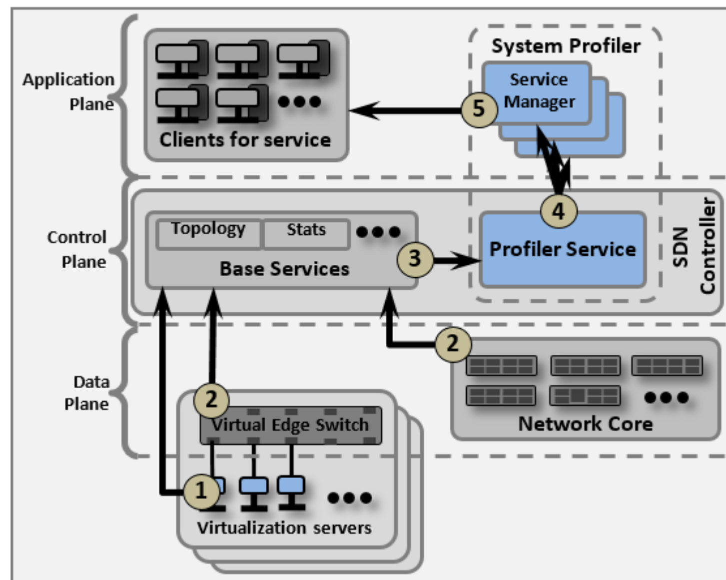


Figure 49: QCN-WFQR SDN traffic balancing

In this case, the 'election' steps of the best RP are:

- **Step 1:** The services are registered to the controller: they register themselves, are registered by an external application or manually by the admin. Either way the controller need to know the list and service provided by each client (based on its IP address). In our case each service is identified by the IP, protocol and port number but can be identified by any L2 or L3 fields supported by OpenFlow.
- **Step 2:** Local congestion indicatives are gathered from core and virtual switches into the topology and statistics databases of the controller.
- **Step 3:** The Profiler Service uses the congestion indicatives previously stored in controller's databases to compute system congestion indicatives and decides, if necessary, to migrate the paths of the existing flows in order to reduce congestion.
- **Step 4:** Each Service Manager requests the congestion indicatives from the Profiler Service through the controller northbound API and prepares an ordered list of RPs based on congestion.
- **Step 5:** The client requests a RP from Service Manager.

Communication between Service Manager and Client is required only for clients that have the capability of electing the RP, for other clients it may be more appropriate to do the election in the service manager itself and just expose them the best RP or, for stateless services it may be better to just distribute the workloads across multiple RPs without informing the client – i.e. act as a standard load balancer.

From a security perspective the communications between the service managers and the controller have to abide by the security policies of the system in general and by those of the controller in particular – service managers will be able to access the controller as long as their access is allowed.

In a multi-tenant environment each tenant that has access to the controller can run its own Service Manager instances thus distributing workload across services will take into account the status of the underlying network.

5.5 QCN-WFQR SIMULATION OF SDN FLOW MIGRATION

In context of SDN, flows migration based on QCN-WFQR achieved a better balanced traffic load within the topology pictured in Figure 50.

In our simulation, the topology is tree based with alternative paths, so each of the four device flows ($Flow_1$ to $Flow_4$) has 2 alternative paths.

In the first 5 seconds of simulation time, the congestion point 3 forwards 3 flows (Flow 1, 2 and 3), while congestion point 4 forwards 1 flow ($Flow_4$). It can be seen from Figure 51 (*Left*) that the CP_3's weight queue is significantly higher compared to CP_4, which initially servers only 1 flow.

After 5 seconds, $Flow_1$ is migrated from CP_3 to CP_4 by SDN controller. Figure 51 (*Right*) shows that the weight queues for the two congestion point converge, also the rates for all four flows converge as well Figure 51 (*Left*) achieving fair rates and a better balanced load.

All of the flows try to reach maximum capacity – the link between RP_1 and CP_1, RP_2 and CP_2 & CP_3 and *Destination* have infinite capacity[57]. All the other links have a limited capacity of 1.5Gb/s therefore, at some point, all of them will congest one of the links.

Figure 51 (*Left*) shows that RP_1 received more CNMs than RP_2. In the beginning (before flow migration) RP_2 receives some CNMs because of Flow 4 (the only one on the path) tries to get more capacity than it is available. While on the other path 3 flows compete at the same time for the same link.

[57] This is common for links between VM and their virtual switches. Everything being virtual there is no hard limit on link capacity.

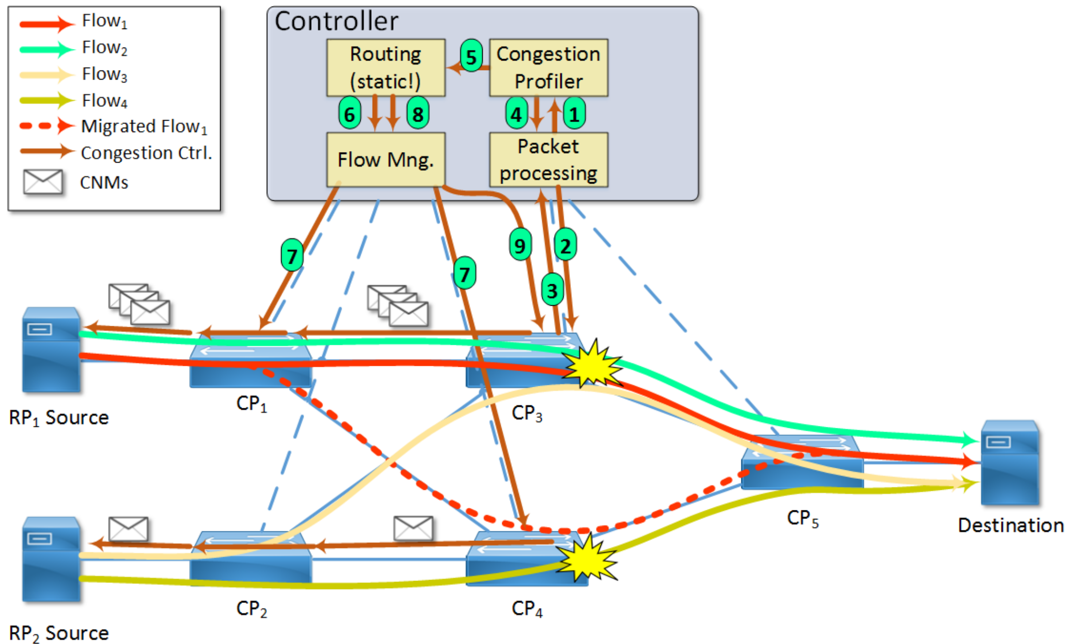


Figure 50: QCN-WFQR Route Migration

For a simplified simulation the Profiler Service was implemented as a stateless application and prove that flow rerouting can be done efficiently with QCN-WFQR. No northbound interfaces were implemented and no path determination algorithm was simulated – the routes were statically modelled. Migrating a flow to an alternative path was decided at runtime based on congestion indicatives gathered by Congestion Profiler from all nodes[58].

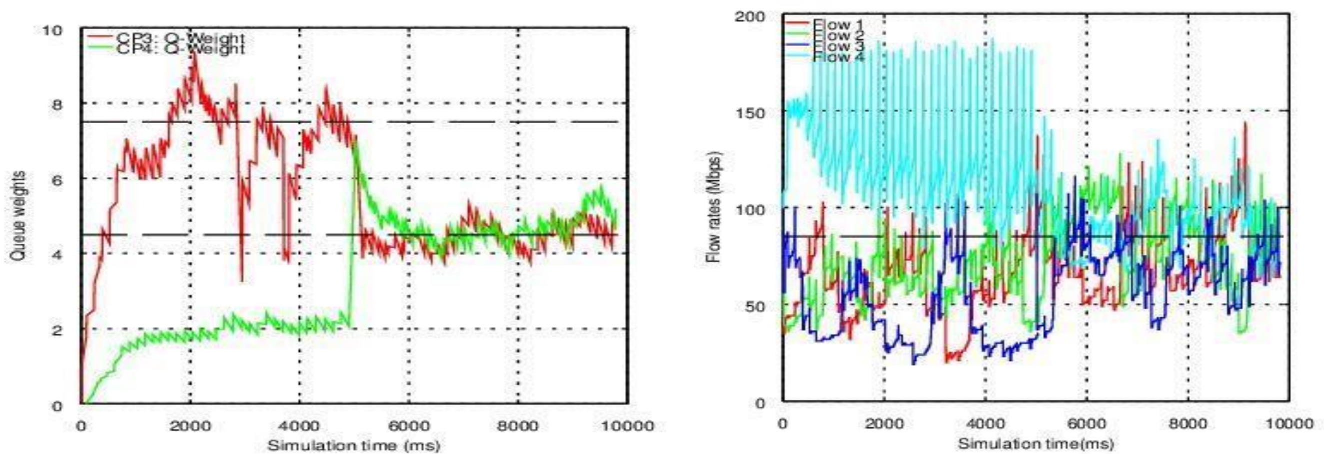


Figure 51: *Left* - CP3, CP4 queues weight. *Right* - flow rates variations

[58] These simulations were done before the SDN controller presented in Ch. 4 had support for dynamic routing.

The Routing Manager had 4 routes programmed, two direct ones and two indirect[59]:

- **Route 1:** RP_1 → CP_1 → CP_3 → CP_5 → Destination
- **Route 2:** RP_2 → CP_2 → CP_4 → CP_5 → Destination
- **Route 3:** RP_1 → CP_1 → CP_4 → CP_5 → Destination
- **Route 4:** RP_2 → CP_2 → CP_3 → CP_5 → Destination

When the simulation starts:

1. Flow 1 and Flow 2 are mapped to Route 1;
2. Flow 3 is mapped to Route 4;
3. Flow 4 is mapped to Route 2;
4. Route Manager generates flows for these routes and pushes them to the switches through Flow Manager which converts them into OpenFlow protocol messages.

Our simulation did a single read of congestion indicatives from the CPs and decided what flow migrations are necessary. In Figure 50 Controller operations are highlighted as steps (from 1 to 9):

- **Step 1-4**: Controller requests congestion indicatives from switches. Congestion Profiler initiate the request (step 1) which goes through packet processing to all of the switches (2). And all switches respond with their data (3 and 4).
- **Step 5**: Based on congestion indicatives the Congestion Profiler decides that one of the links is too congested and need to move one of the flows from it. This is transmitted to routing manager.
- **Step 6-7**: Routing manager statically moves one flow away from the congested link. It first pushes the new route to all of the switches (7). The new route has a higher priority than the old ones therefore messages go through it. The old route is kept for a while as old packets may still be in transit.
- **Step 8-9**: The old route is removed.[60]

5.6 Conclusions

QCN Weighted Flow Queue Ranking (QCN-WFQR) algorithm computes congestion indicatives that are measures of the load generated by flows in different points in the network. The system is capable to compute the contribution of a flow, $Weight_{flow,}$ and the contribution of a congestion source (RP), $eight_{RP,}$ to the network load. Based on these congestion indicatives clients can determine the best suited server (RP) for services in order to cooperatively balance the load in the network.

Simulations show how an SDN controller can benefit from WFQR when optimizing traffic routes in a Data Center.

---

[59] Actually there were 8 routes as their reverses were needed for ARP responses but the simulation did not touch them.

[60] Flow expiry may be configured at step 7 and just let the flows expire instead of removing them through a special request from the controller.

Analysis of the solutions presented in this chapter can be continued with an analysis of algorithm relevance for different traffic profiles (e.g. bursts and streaming) and with analysis on how different topologies influence the algorithm decisions.

From a controller perspective dynamic routing based on congestion indicatives and different algorithms for traffic balance may be further researched.

6 VLAN-PSSR: PORT-SWITCHING BASED SOURCE ROUTING USING VLAN TAGS IN SDN DATA CENTERS

Source Routing allows a node in the path of a packet to specify partially or completely the route taken by that packet. The route is appended to packet header as a list of nodes to traverse, therefore making simple, stateless forwarding decisions. In this chapter we present a novel approach of Source Routing in SDN that uses stacked VLAN tags. Our solution was validated in Mininet using the Ryu controller and proven to have multiple advantages over other forwarding methods.

6.1 INTRODUCTION

Many custom implementation of source routing exists (mainly in HPC – High Performance Computing solutions) such as Myrinet [99], Quadrics [100] and IEEE 1355.

In standard computer networking, IP provides a special option, *Loose Source Routing* (RFC 791 [101]), which can be used to specify a list of routers that a packet can take. At each node, packet destinations are replaced with information from this list so that a packet can tunnel through a network that otherwise is unable to forward it. This is intended to provide mobility for users through multiple provider networks. To note here that this option can become a security hazard as it may be used to piggyback packets to destinations that would otherwise be unreachable. The solution is limited to IP only and most internet routers disable it.

In SDN, the controller keeps the global view of the network, it controls the forwarding nodes, and knows what hosts are connected to edge switches. Therefore, it is much easier for it to build and set routes at the edge of the network for all the packets entering it. Furthermore, OpenFlow has the necessary mechanisms to create this kind of behavior without modifications.

In Data Center computer networking in general, and SDN in particular, there are three main techniques for packet forwarding:

1. **Destination based forwarding** – packets destinations (e.g. MAC or IP destination addresses) are matched against a list of destinations and, when an entry matching the searched address is found, packet is forwarded on the correct port using forwarding information from the matched list entry.
2. **Label based forwarding** – packets entering the network are classified and a label is appended to each packet, then forwarded based only on that label (e.g. ATM and MPLS).
3. **Source routed based forwarding** – packets are forwarded based on a list of nodes specified in the packet itself.

A list of different forwarding techniques is presented in [102]. The paper analyzed CONGA [103], Shadow MACs [104], XPath [105], FastPass [106], SlickFlow [107] and SecondNet [108].

The question is, why source routing? SDN can already forward a packet through the network so, what benefits can source routing bring? The following sections will explore this and present a novel source routing solution that provides both unicast and multicast.

The proposed source routing solution is based on stacked VLAN tags that are pushed at edge and popped at each node after forwarding is decided. A similar solution using MPLS tags is presented in [108]. The main advantage of VLAN stacking over MPLS is that support for MPLS is limited in core switches while VLAN is much more common. Some switches along the path do not have MPLS support while the majority only support 3 levels of MPLS tags. Even if this number can be increased in software switches, since hardware ones are limited it can only be used for networks with small diameter. Also, header sizes are smaller with VLAN tags – 2 bytes versus 4 bytes for MPLS (+2 bytes for header type in both cases).

VLAN based source routing only needs VLAN forwarding and popping – a common feature in current generation OpenFlow hardware switches. Even though these switches are usually unable to *push* more than 4 tags, they are able to easily *pop* a single tag and forward based on it while keeping the rest of tags intact. Pushing many tags is only required by edge switches which usually are software switches, therefore easier to implement[61].

Our solution also provides solutions for broadcast domains and multicast traffic.

6.2 APPROACHES TO SOURCE ROUTING USING OPENFLOW

OpenFlow version 1.3 or higher provides mechanisms to implement source routing without modifications to software or hardware. To benefit from it switches have to implement OpenFlow 1.3 and some of the optional extensions in the standard (i.e. VLAN tag *push* & *pop* and *wildcard matching*).

In source routing next hop may be identified either by a port number on the current switch – this method is called Port-Switching based Source Routing (PSSR) – or by an ID of next hop switch. In the latter case switches need to learn the ID's of their neighbors and what ports they are connected to we will call this ID based Source Routing (IDSR).

Source routing itself has multiple approaches based on how the list of nodes is processed, in SDN three approaches have been identified [102] [109] [108]:

First, we have *stacked labels* solutions where all labels are pushed by the *source* node (or edge switch) and each hop then uses one label in the stack (usually the last one) to make the forwarding decision. After deciding where to forward the packet the label is removed and packet forwarded. Next hop in the path repeats the same procedure. This approach is usable with MPLS and VLAN tags. Forwarding may be PSSR or IDSR and depends on how *tags* are interpreted.

Second, we have *masked bits with pointers* where each route is represented as set of bits in the header (usually a reused Ethernet or IP header) plus a pointer that identify which bits to use at current hop. These can be either a set of bit flags or a counter. The pointer need to change at each node, for example a counter may be incremented at each node or a bit flipped. This will create additional processing at each switch which may decrease performance. To avoid this, the approach is to reuse the TTL area of a header as it is incremented in hardware. TTL is actually a back counter and can be used to identify what bits to match. This solution works with PSSR.

[61] Open vSwitch code changes that add support for multiple VLAN tags are under review and will be available in the next official release. We used this code for our implementation.

Main advantage of the method is that an existing header could be better used than *stacked labels* as there is no header overhead introduces with each header (both VLAN and MPLS have a 2 bytes overhead for each tag, therefore for a maximum of 8 tags we will get 16 bytes overhead) but for networks with small diameter (less than 5 hops) the overhead may actually be similar as there are always parts of the reused header that are unused (e.g. for an IP header we have two 32 addresses that can be used, each describing 4 hops, therefore for a 4 hop diameter we would leave at least 32 bits unused from the 64 available). Disadvantages are limited diameter – 8-10 hops maximum, depending on the header type (IP or Ethernet), increased complexity and a high number of entries needed in flow tables.

For example, in case PSSR is used, if we consider that the maximum number of ports of any switch in a network is 256 then we can express that with 8 bits, this means than if the 32 bits of an IP address are reused for port number then we could represent a 4 hop route ($4 \text{ hops} * 8 \text{ bits} = 32 \text{ bits}$) in it. The TTL would then be used to choose which 8 bits to use for the current hop. This approach has a major disadvantage, the flow table size is proportional with network diameter because a switch need to match on each (pointer value, byte in reused header) tuple. Therefore for a diameter of five each switch needs $5 * 256 = 1280$ entries. Therefore table sizes can get quite large.

Third, we have *masked bits based on switch ID* where the supplementary IP or Ethernet headers contain a list of switch IDs therefore one switch will have to match on its own ID and then use the next ID in the list for forwarding or, if IDs are unique, just check if any neighbor ID is present in the header and forward to id. This is an IDSR solution meaning that each switch has to know where all of its neighbors connect and their IDs to make forwarding possible are. The size of each *bitfield* entry is dependent on the total number of switches in a network (a 64 nodes network needs 6 bits per host while a 1024 nodes needs 10 bits to cover all possible IDs). Matching can also be challenging as the number of entries may be high and is directly dependent on the network diameter: 5 tags max diameter results in $5 * 256 = 1280$ entries similar to the previous approach. This method can be advantageous on small networks (less than 256 nodes) as it reduce the number of bits per node.

Another disadvantage of second and third method is that adding queue based traffic engineering requires additional bits which are hard to manage as more matching wildcard rules need to be set for queue selection.

6.3 Limitations of destination/labels based routing and advantages of source routing

Source routing reduces the following limitations of SDN standard destination based forwarding or tag based forwarding:

1. Limitations of flow tables – with source routing table usage drops dramatically [102]. We will show bellow that our method needs a static number of flows in core switches. Number of flows increases in edge switches but, since they are software, this is not a major issue.
2. Slow network updates – network updates can be slow when many flows need to be updated at once. The slowness come both from the controller and from the switches themselves (more details and argumentation in section 4.6). Source routing can reduce this issue as the number of flows in core switches is reduced. Our approach does not involve any update of flow tables in switches other than ones from the edge.

3. Traffic engineering can be complex and multipath routing hard to implement. With source routing flows can easily be scheduled from edge to go on multiple paths and packet distribution can be better controlled. In fact, the edge can choose a different route on a per-packet basis.

Limitations of source routing:

1. Failover can be hard to implement, once a device on the path of a source routed packet fails, the switches neighboring the failed device no longer know how to forward those packets; a solution would be to just send them back to the controller but this may overwhelm it.
2. Source routing based on switch ID's may be more resilient to failover than port based ones because, if a port fails, packets may be forwarded to a neighbor that is aware of the next ID in the path and may reroute around the failed device so that packets returns to the previous hop in the route.

Full broadcasts and multicast is not supported with source routing as multiple destinations are almost impossible to specify. Our solution is partial yet usable as it covers most use cases in real world Data Centers.

6.4 Description of the VLAN-PSSR solution

In the Ethernet header, VLAN tags sit between source MAC address and higher protocols headers, usually IPv4 or IPv6. Multiple tags are appended to the packet one after the other. In the Ethernet header, these tags are identified by an Ethertype of 0x8100, therefore each tag contains these two bytes. Only after Ethertype we have VLAN specific information:

- Priority code point (PCP), a 3 bit-field which maps a packet to a priority queue,
- Drop Eligible Indicator (DEI) – single bit that indicates if packets are eligible for dropping in case of congestions and
- VLAN ID (VID) – a 12 bit field specifying the VLAN to which the packet belongs.

Our solution, VLAN-PSSR reuses VID for source routing. One bit is used for specifying if the tag is multicast or unicast. For unicast 8 bits specify the port number of a switch (0 to 255) while 3 bits are not used (Figure 52).

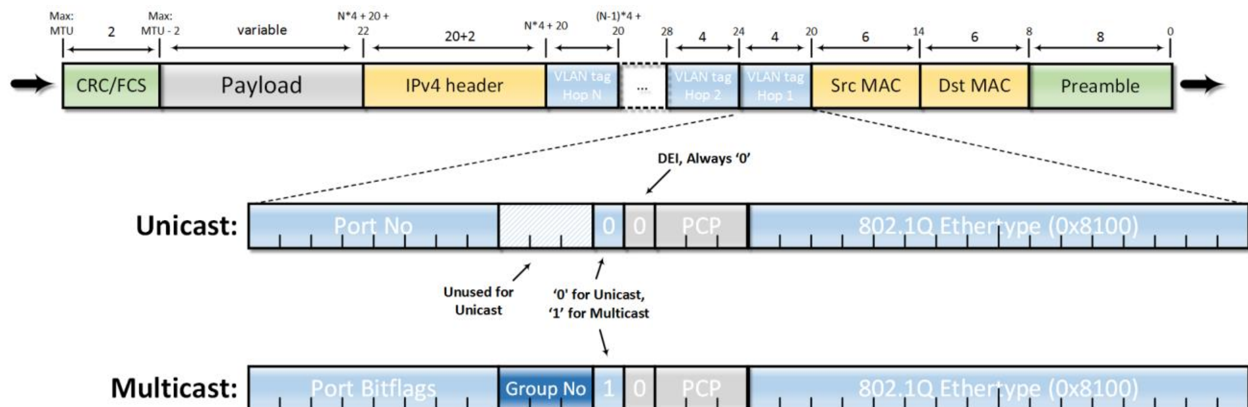


Figure 52: VLAN-PSSR Packet Format

For multicast we divide the 11 bits of VID in two parts:

1. Group *number* – an unsigned integer pointing to a subset of ports. If we divide the total number of ports of a switch in groups, this number represents one such division.
2. Port bitflags – a set of bits representing all ports of a group. Each bit corresponds to a single port. Multiple ports can be selected at the same time. Therefore, a packet is forwarded on all ports that are part of that group and have their bits set to 1 in port bitflags

For example, in Figure 52, 3 bits are reserved for group number and 8 for port bitflags. In this configuration we can define 8 groups, each with 8 ports, for a total of 64 ports.

Note that we can remove the multicast bit and consider any message that has a group number higher than '0' to be multicast. In this case we can multiplex (duplicate) a packet up to 120 ports. If we need more ports, we can go further and increase the group bitflag to 5 bits and decrease the port bitflags to 7 bits resulting in 217 ports. A higher multiplexing also increases the number of tags needed which increases packet size. Therefore, the optimal number of tags should be selected based on the number of ports that a switch has.

The maximum number of ports (*P*) and max groups (*G*) can be computed with:

$$\begin{cases} P = (2^{Gs} - 1) * Ppg \\ G = 2^{Gs} \\ Gs + Ppg = 11 \end{cases} \tag{6.1}$$

Where *Gs* is the number of bits reserved for group number, *Ppg* is the number of bits in *Port bitflag* (i.e. ports per group) and 11 is the number of bits in VID.

6.5 FUNCTIONAL VALIDATION IN MININET

To validate our model we first implemented it in Mininet [32] with a small configuration (Figure 53) and verified that *ping* is successful and that UDP and TCP data connections can be successfully established. We then validated the message content with Wireshark [110].

To make the setup work we used the latest version of Open vSwitch [31] from the development branch (v 2.5) and applied a patch for allowing multiple VLAN tags[62]. Open vSwitch was then connected to Ryu SDN controller [111] and on top of Ryu we implemented our VLAN-PSSR application.

6.5.1 Unicast solution

Mininet setup consists of three hosts H_1, H_2 and H_3 and 5 switches: S_1, S_2 and S_3 at the edge and C_{11} and C_{12} at core. Hosts are Linux containers instances[63] and switches are Open vSwitch instances (bridges) connected to Ryu controller. Flows are then managed by our VLAN-PSSR implemented on top of Ryu.

[62] This patch is currently under review and will be available in next official release

[63] This is an *operating system level virtualization* solution provided by the Linux kernel. It offers logical isolation of process, networking and file system resources between containers so that any one container is unable to access resources from other containers.

We configured Ethernet MAC addresses equal to host number (for easier identification) and OpenFlow *datapath IDs* (dpid) to the switch number (1, 2, 3, 11=0xb & 12=0xc). Communication between H_2 and H_3 uses destination based forwarding, without source routing, as they are only one hop away while communication between H1 ⇔ H2 and H1⇔ H3 uses Source Routing. The validation configuration is presented in Figure 53.

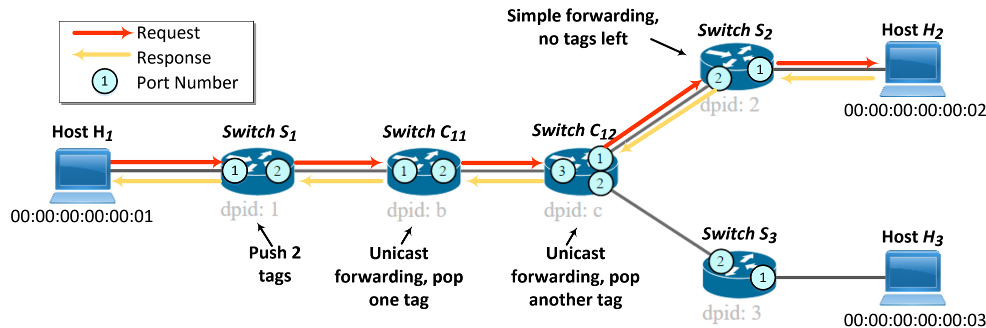


Figure 53: VLAN-PSSR unicast validation setup

PSSR tags are added in the edge switches (S_1, S_2 and S_3) and removed (*popped*) in core at each switch. As an example, in Figure 53, we have a request and response path between source H_1 and destination H_2 where tags are pushed in S_1 with the flows in Table 10 and forwarded between core switches with flows in Table 11. Flows in `blue` are management flows and flows in `black` are the ones implementing VLAN-PSSR. These are real flows available in switches when functional validation was done (in `gray` is the CLI command executed from Linux to get the output). Management flows forward packets to controller if no match is found, flood ARP requests and allow LLDP neighbor discovery[64].

Table 10 shows the process of matching a packet header and pushing all needed VLAN-PSSR tags. When a packet enters the switch it is matched against its destination (e.g. request in Figure 53 has a destination address of 00:00:00:00:00:02) and a *tag* is added, then it is sent to the next table in the pipeline, matched again and *another tag* added. In the end it is forwarded to next hop (C_{11}) that is connected to port 2:

- **flow #3:** destination 00:00:00:00:00:02 is matched, first tag is added (for crossing C_{11} → C_{12}) & sent to table 1
- **flow #7:** destination matched again, second tag is added (for crossing C_{12} → S_2) & packet is sent to port 2.

Table 10: Unicast flow table of switch S_1

```
No  mininet@mininet-vm:~/ryu-scripts/ryu-client$ sudo ovs-ofctl -O OpenFlow14 dump-flows s1 | awk
    '{$1=$2=$4=$5=""; print $0}' |  sed 's/^ *//g' | sed 's/  //g'
    (OF1.4)
1.  table=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
2.  table=0, priority=60000,dl_dst=ff:ff:ff:ff:ff:ff actions=FLOOD
3.  table=0, in_port=1,dl_dst=00:00:00:00:00:02 actions=push_vlan:0x8100,set_field:4097->vlan_vid,goto_table:1
4.  table=0, in_port=1,dl_dst=00:00:00:00:00:03 actions=push_vlan:0x8100,set_field:4098->vlan_vid,goto_table:1
5.  table=0, in_port=2,dl_dst=00:00:00:00:00:01 actions=output:1
6.  table=0, priority=0 actions=CONTROLLER:65535
7.  table=1, in_port=1,dl_dst=00:00:00:00:00:02 actions=push_vlan:0x8100,set_field:4098->vlan_vid,output:2
```

[64] We did not use multicast or broadcast at this time

8.	table=1, in_port=1,dl_dst=00:00:00:00:00:03 actions=push_vlan:0x8100,set_field:4098->vlan_vid,output:2

At core switches, in Table 11, packets are matched against their VLAN tags and forwarded to corresponding ports (e.g. match on VID 1 will forward to Port 1); see flows #3 to #17. Before forwarding packets to their outputs a tag is *poped* from the stacked list of VLANs.

Table 11: Unicast flow table of switch C_{11} & C_{12}

No	mininet@mininet-vm:~/ryu-scripts/ryu-client$ sudo ovs-ofctl -O OpenFlow14 dump-flows c11 | awk '{$1=$2=$4=$5=""; print $0}' | sed 's/^ *//g' | sed 's/ //g' (OF1.4)
1.	table=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
2.	table=0, priority=60000,dl_dst=ff:ff:ff:ff:ff:ff actions=FLOOD
3.	table=0, dl_vlan=1 actions=pop_vlan,output:1
4.	table=0, dl_vlan=2 actions=pop_vlan,output:2
5.	table=0, dl_vlan=3 actions=pop_vlan,output:3
…	[cut 12 entries]
17.	table=0, dl_vlan=15 actions=pop_vlan,output:15
18.	table=0, priority=0 actions=CONTROLLER:65535

Looking at the tables in core switches we see that they are static in size and depend linearly on the number of ports, so for a 64 port switch only 64 static entries are needed. With destination or label based forwarding managing thousands of flows in each core switch is normal but, with source routing, we can substantially reduce this number to a maximum of a few hundred.

6.5.2 Multicast solution

For VLAN-PSSR multicast packets are transmitted on a unicast path until penultimate hop where packets are multiplied and sent to the last hop for final forwarding. The reason for doing this is that VLAN-PSSR can only do a single multiplication and this needs to be close to the packet destination. In data centers usually the penultimate hop is the Top of Rack (ToR) switch while last hop is the virtual switch of servers. Therefore, our solution is providing multicast inside a single rack but, since we are targeting multitenant Data Centers with edge virtual switches, multicasting inside the same rack represents the majority of use cases. Broadcast domains are usually small in multitenant Data Centers with only a few VMs connected to the same domain (around 10 - 20) which, to reduce bandwidth usage of the core network, are kept closely together, rarely spanning multiple racks.

For multicasting between *racks* a single flow needs to be sent to each nearby racks yet this is much better than sending a separate flow to each destination. Therefore, two multicast approaches are possible (Figure 54):

1. **By starting a flow from the source server to each rack** – this has the advantage of reduced latency but higher bandwidth and processing is needed in the source server (Figure 54 – 1);
2. **By jumping from rack to rack** – this has less impact on processing because the controller can chose the servers that send packets across racks based on resource usage. This adds more latency (Figure 54 – 2).

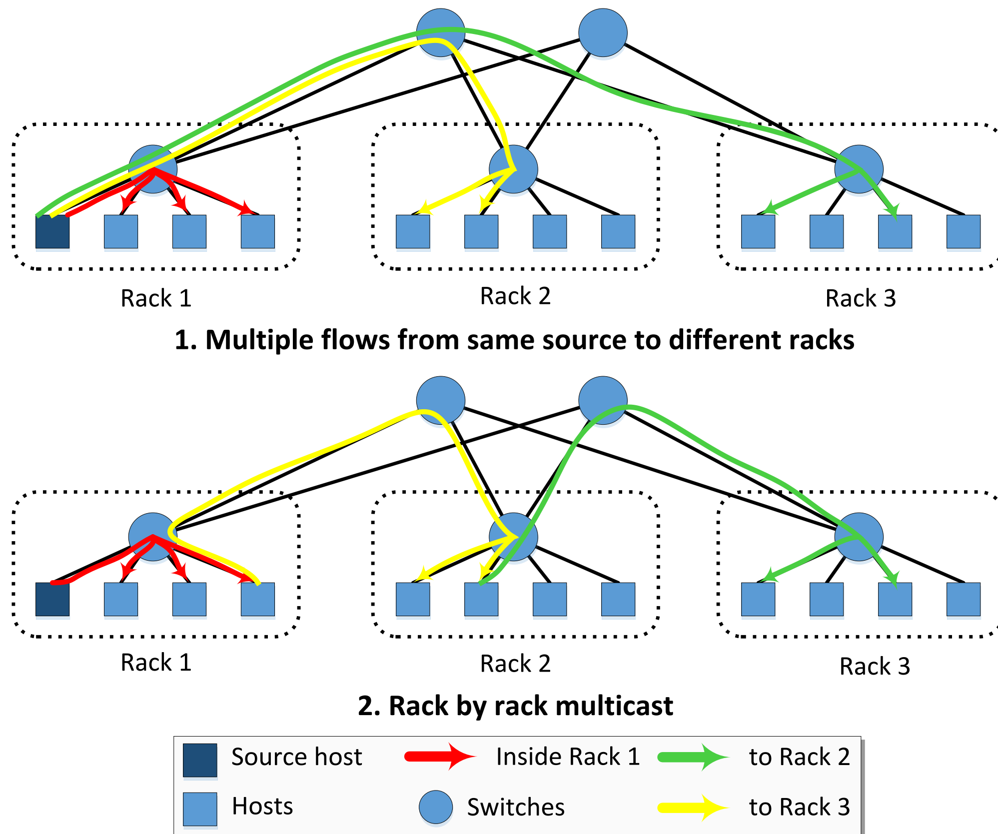


Figure 54: Multicast between *racks*

For validating our approach, we used the same setup as before, with a multicast stream originating in H_1 and sent to both H_2 and H_3 (Figure 55). From H_1 to C_{12} packets are transmitted using unicast and multiplication is done by C_{12}.

At edge switch S_1 both unicast and multicast tags are added, unicast first and multicast last so that, when packet arrive at C_{12}, only the multicast tags are left. Then C_{12} multiplies the packet and forwards it to both destination edge switches. To note that C_{12} is unable to drop multicast tags so the edge switches need to pop any remaining tags before sending the packet to the destination host[65].

[65] This is only valid for OpenFlow 1.3. Higher versions have two pipelines one on *ingress,* which processes packets when they enter the switch and another one on *egress*, which processes packets after *output* port action has been decided. Therefore, multiplication is done on *ingress* and VLAN tag drop can happen on *egress*. We used version 1.3 capabilities as these are more widespread.

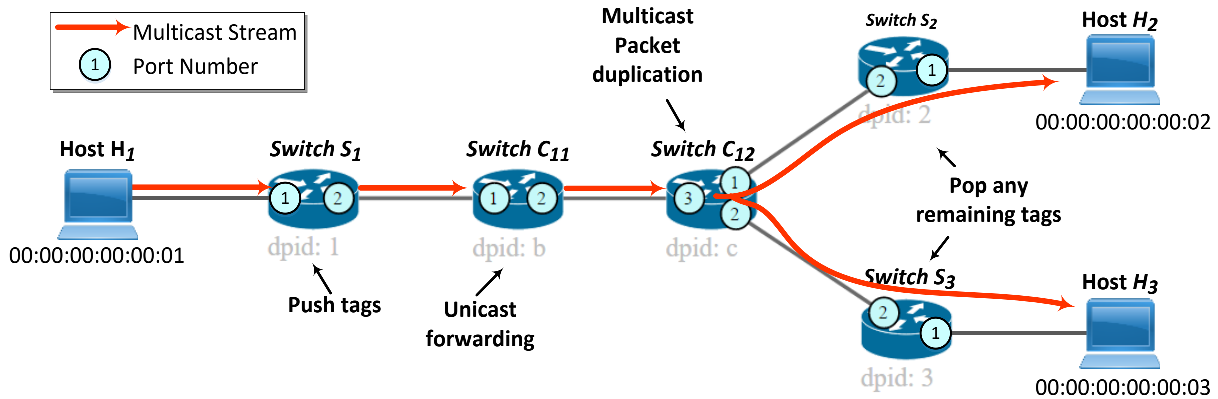


Figure 55: VLAN-PSSR Multicast setup

Flow table of edge switch S_1 is presented in Table 12. Management flows in `blue`, flows that add VLAN tags in `black`, flows that drop all VLAN tags remaining are `red`. For validation only two tags are added, one for unicast and another one for multicast, we experimented with 1 to 8 tags in next section (6.6).

Table 12: Flow Table of Edge switch S_1 in multicast case

No	
	`mininet@mininet-vm:~/ryu-scripts/ryu-client$ sudo ovs-ofctl -O OpenFlow14 dump-flows s1 | awk '{$1=$2=$4=$5=""; print $0}' |  sed 's/^ *//g' | sed 's/  //g'` `(OF1.4)`
1.	`table=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535`
2.	`table=0, priority=65535,vlan_tci=0x1800/0x1800 actions=pop_vlan,TABLE`
3.	`table=0, priority=60000,dl_dst=ff:ff:ff:ff:ff:ff actions=FLOOD`
4.	`table=0, priority=0 actions=CONTROLLER:65535`
5.	`table=0, in_port=1,dl_dst=00:00:00:00:00:02 actions=push_vlan:0x8100,set_field:4097->vlan_vid,goto_table:1`
6.	`table=0, in_port=1,dl_dst=00:00:00:00:00:03 actions=push_vlan:0x8100,set_field:4098->vlan_vid,goto_table:1`
7.	`table=0, in_port=1,dl_dst=01:00:5e:00:00:01 actions=push_vlan:0x8100,set_field:6150->vlan_vid,goto_table:1`
8.	`table=0, in_port=2,dl_dst=00:00:00:00:00:01 actions=output:1`
9.	`table=1, in_port=1,dl_dst=00:00:00:00:00:02 actions=push_vlan:0x8100,set_field:4098->vlan_vid,output:2`
10.	`table=1, in_port=1,dl_dst=00:00:00:00:00:03 actions=push_vlan:0x8100,set_field:4098->vlan_vid,output:2`
11.	`table=1, in_port=1,dl_dst=01:00:5e:00:00:01 actions=push_vlan:0x8100,set_field:4098->vlan_vid,output:2`

Flow tables of nodes doing multicast forwarding (penultimate hops) are complex (Figure 56). Processing is done in a pipeline starting at table 0 and each pass processes a single multicast tag, therefore if multiple tags are present multiple passes of the same packet through the pipeline are needed, which decreases performance when used in software.

Processing takes the following steps (*P* is the number of ports in a group and *G* the number of groups):

1. **In table #0,** multicast tag is identified. If the packed is multicast tagged processing continues in tables 10.
2. **In table #10** packet is matched against a single entry and sent to a port otherwise is sent to next table.
3. **In table #11 to #10 + (P-1)** packet is matched against other group/port pair until reaching end of pipeline
4. **In table #10 + P** if packet still contains multicast tags it is sent back to the beginning of pipeline to process another tag otherwise is considered processed and dropped.

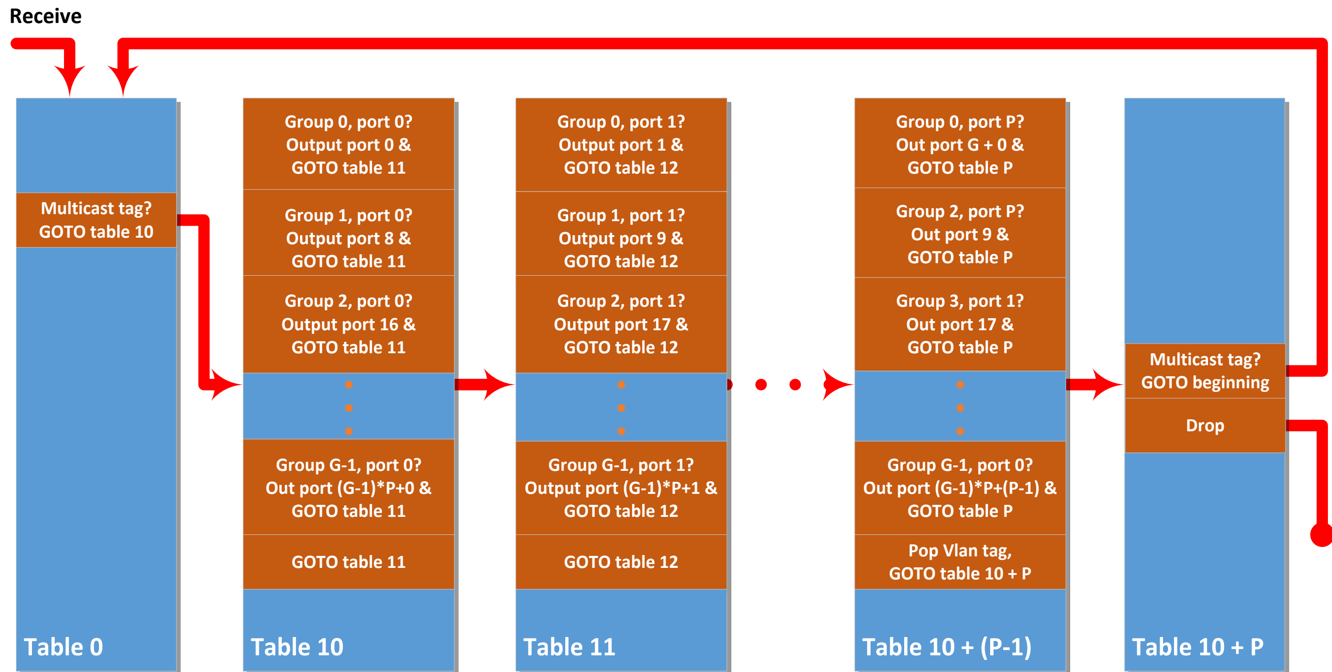


Figure 56: Multicast flow entries of core switches

A sample output of a multicast forwarding is presented in Table 13 where we have 64 ports in 8+1 tables.

Table 13: Flow Table of Edge switch C_{12} in multicast case

```
mininet@mininet-vm:~/ryu-scripts/ryu-client$ sudo ovs-ofctl -O OpenFlow14 dump-flows c12 | awk '{$1=$2=$4=$5="";
print $0}' |  sed 's/^ *//g' | sed 's/  //g'
(OF1.4)
table=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
table=0, priority=60000,dl_dst=ff:ff:ff:ff:ff:ff actions=FLOOD
table=0, priority=0 actions=CONTROLLER:65535
table=0, dl_vlan=0 actions=pop_vlan,output:0
table=0, dl_vlan=1 actions=pop_vlan,output:1
table=0, dl_vlan=2 actions=pop_vlan,output:2
table=0, dl_vlan=3 actions=pop_vlan,output:3
table=0, vlan_tci=0x0800/0x0800 actions=goto_table:10
table=10, vlan_tci=0x1001/0x1701 actions=output:0,goto_table:11
table=10, vlan_tci=0x1101/0x1701 actions=output:8,goto_table:11
table=10, vlan_tci=0x1201/0x1701 actions=output:16,goto_table:11
table=10, vlan_tci=0x1301/0x1701 actions=output:24,goto_table:11
table=10, vlan_tci=0x1401/0x1701 actions=output:32,goto_table:11
table=10, vlan_tci=0x1501/0x1701 actions=output:40,goto_table:11
table=10, vlan_tci=0x1601/0x1701 actions=output:48,goto_table:11
table=10, vlan_tci=0x1701/0x1701 actions=output:56,goto_table:11
table=10, priority=0 actions=goto_table:11
table=11, vlan_tci=0x1002/0x1702 actions=output:1,goto_table:12
table=11, vlan_tci=0x1102/0x1702 actions=output:9,goto_table:12
table=11, vlan_tci=0x1202/0x1702 actions=output:17,goto_table:12
table=11, vlan_tci=0x1302/0x1702 actions=output:25,goto_table:12
table=11, vlan_tci=0x1402/0x1702 actions=output:33,goto_table:12
table=11, vlan_tci=0x1502/0x1702 actions=output:41,goto_table:12
table=11, vlan_tci=0x1602/0x1702 actions=output:49,goto_table:12
table=11, vlan_tci=0x1702/0x1702 actions=output:57,goto_table:12
[cut from table 12 to 16]
table=17, vlan_tci=0x1080/0x1780 actions=pop_vlan,output:7,TABLE
table=17, vlan_tci=0x1180/0x1780 actions=pop_vlan,output:15,TABLE
table=17, vlan_tci=0x1280/0x1780 actions=pop_vlan,output:23,TABLE
```

```
table=17, vlan_tci=0x1380/0x1780 actions=pop_vlan,output:31,TABLE
table=17, vlan_tci=0x1480/0x1780 actions=pop_vlan,output:39,TABLE
table=17, vlan_tci=0x1580/0x1780 actions=pop_vlan,output:47,TABLE
table=17, vlan_tci=0x1680/0x1780 actions=pop_vlan,output:55,TABLE
table=17, vlan_tci=0x1780/0x1780 actions=pop_vlan,output:63,TABLE
table=17, priority=0,vlan_tci=0x1000/0x1000 actions=pop_vlan,TABLE
```

Edge switch tables (Table 14) only pop VLANs and provide standard forwarding; for our experimentation we used multicast Ethernet group 01:00:5e:00:00:01. With `blue` we marked management flows, with `red` unicast flows.

Table 14: Flow Table of Edge switch S_2 in multicast case

```
mininet@mininet-vm:~/ryu-scripts/ryu-client$ sudo ovs-ofctl -O OpenFlow14 dump-flows s2 | awk '{$1=$2=$4=$5=""; print
$0}' |  sed 's/^ *//g' | sed 's/  //g'
(OF1.4)
table=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
table=0, priority=60000,dl_dst=ff:ff:ff:ff:ff:ff actions=FLOOD
table=0, priority=0 actions=CONTROLLER:65535
table=0, priority=65535,vlan_tci=0x1800/0x1800 actions=pop_vlan,TABLE
table=0, dl_dst=01:00:5e:00:00:01 actions=output:1
table=0, in_port=1,dl_dst=00:00:00:00:00:03 actions=push_vlan:0x8100,set_field:4098->vlan_vid,output:2
table=0, in_port=1,dl_dst=00:00:00:00:00:01 actions=push_vlan:0x8100,set_field:4097->vlan_vid,goto_table:1
table=0, in_port=2,dl_dst=00:00:00:00:00:02 actions=output:1
table=1, in_port=1,dl_dst=00:00:00:00:00:01 actions=push_vlan:0x8100,set_field:4099->vlan_vid,output:2
```

Flow table size (*Tsize*) in penultimate hops (i.e. the Top of Rack switch) is proportional with number of groups used (*Gu*) from the total (*G*), ports per group (*Ppg*) and total number of ports of that switch (*P*):

$$Gu = \left\lceil \frac{P}{Ppg} \right\rceil \quad (2)$$

$$Tsize = 3 + (Gu + 1) * Ppg \quad (3)$$

$$Ntables = Gu + 2 \quad (4)$$

Therefore, for a 64 ports Top of rack switch, where $Ppg = 8$, $Tsize = 75$, which is an easily manageable number.

6.6 PERFORMANCE EVALUATION OF VLAN-PSSR WITH OPEN VSWITCH ON XEON SERVERS

To better understand the impact of our solution to real world deployments we evaluated the CPU usage of Open vSwitch when adding tags and multicasting traffic. Performance evaluation was conducted on a real data center server that uses a Xeon class CPU, an Intel Xeon E5-2670 at 2.60GHz with 16 core.

We connected the *server* to a *traffic generator* system (an Intel Core i7-4820K at 3.70GHz with 4 cores) using two 10Gbps fiber optics NICs. Both systems were running Ubuntu 14.04 LTS Linux distribution and were managed remotely through *ssh* (secure shell) connections from a Management Terminal (i.e. our laptops).

The setup we used is detailed in Figure 57. It has three Open vSwitch *bridges* (S1, S2 on the traffic generator and C10 on the benchmarked server) and two OS level virtualized *hosts* (H1 and H2). The two hosts are running LXC containers [112] with the same Linux distribution as the server. H1 is connected to S1 through a virtual interface (h1eth1). S1 is connected to C10 by a 10 Gbps physical link, same as C10 to S2 connection. And finally, S2 is connected to H2 through another virtual interface (h2eth1). Both servers have one port connected to the management network (eth1 and eth0 respectively).

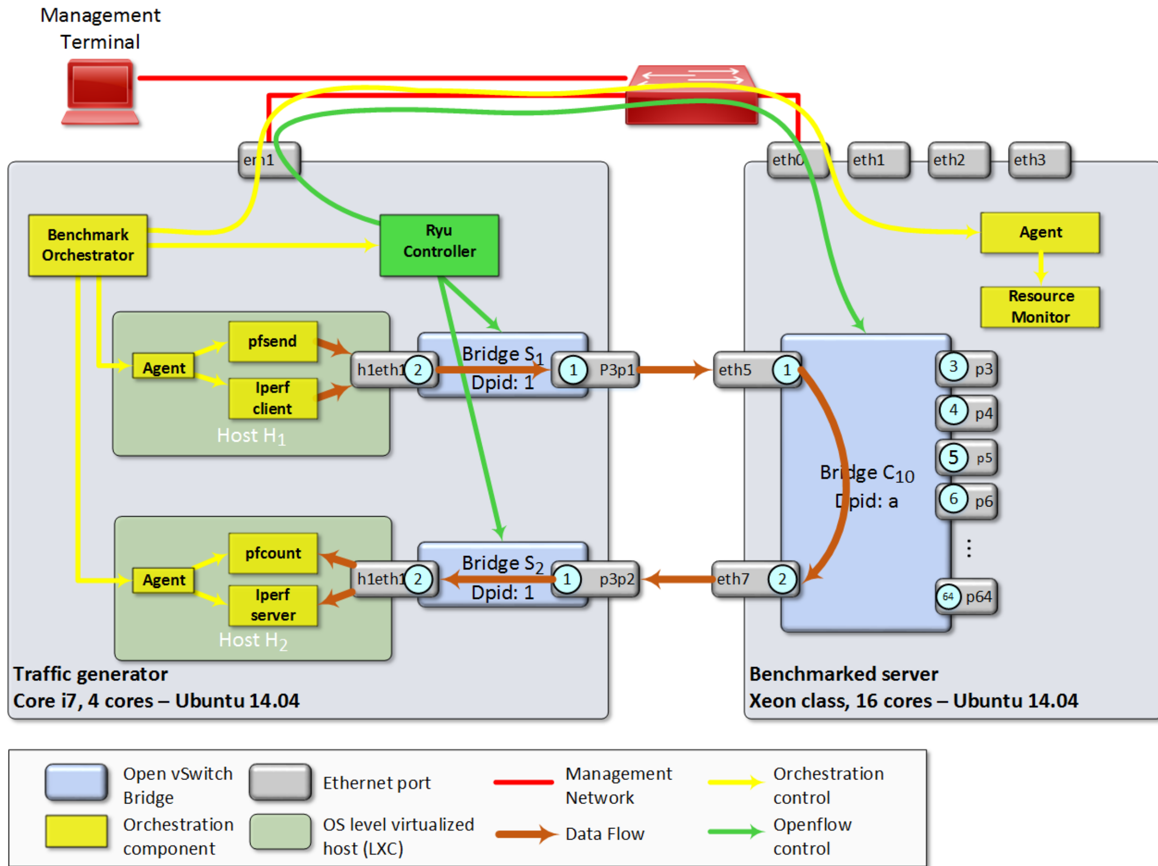


Figure 57: Benchmarking setup

All three Open vSwitch *bridges* are opening OpenFlow 1.3 connections to Ryu controller running on the *traffic generator* (green colored links in Figure 57).

For benchmarking VLAN-PSSR we built three components:

1. **VLAN-PSSR Application** – is built on top of Ryu controller and pushes source routing flows in the switches. The application exports a small set of REST APIs that allow the benchmark orchestrator to change simulation scenarios.
2. **Benchmark orchestrator** – provides automated benchmarks. Once started from the CLI, it configures a scenario in the VLAN-PSSR application and connects to agents for starting traffic or gathering results.
3. **Agent** – a component that provides RES APIs for simplifying access to traffic generation components and resource monitoring, mainly CPU usage. This component can also set the affinity of processes on the server to certain cores (i.e. pin a process to specific core) so that we can monitor only the usage of Open vSwitch without interference from other components.

Our Benchmarking orchestrator is started from CLI, it connects to the *VLAN-PSSR application* and configures the SDN flows. Then it connects to the *agent* on H1 to start traffic and to another *agent* on the server to monitor CPU load. Connecting on the *agent* on H2 to check if the throughput is correct is optional.

For generating traffic on one end, and monitoring it on the other, we used two simple yet very reliable traffic generation applications: *pfsend* and *pfcount* from PF_RING high-speed packet generation and capture framework [113] [114]. We initially tried to use *iperf* but it does not provide the desired fidelity: a constant stream of same size packets at a constant rate (with less than 1% variation per second). *Iperf* streams have too much jitter and throughput varies over time (sometimes over 10% variation per second) resulting in unreliable CPU readings.

Open vSwitch has two components, a *Linux kernel module* for fast packet processing and forwarding and a *user space application* that controls the kernel module and performs slower, but more complex, processing. So, to determine resource usage we have to monitor both components and eliminate interference from other components that are running at the same time.

To gather statistics as correct as possible we went to great length and (1) disabled all nonessential services, (2) forced Linux scheduler to only schedule for a single core (isolated scheduler to core 1) and (3) moved all remaining processes not part of Open vSwitch, such as the *ssh* server and our monitoring Agent, to another core (core 9). Also, to avoid impact of cache thrashes generated by these processes we chose this core to be on another *node* of our Xeon's NUMA [115] architecture[66].

Two main benchmarks where executed:

1. **We benchmarked the impact of VLAN tag pushing at the edge against a flow without tags**. We pushed from 1 to 64 tags and measured CPU usage and standard deviation. Each test was executed over a 30 seconds period and was repeated 5 times. These tests were executed for 100, 200 and 300 thousands packets per second. This reaches the limit of a single core (we isolated Open vSwitch to one core).
2. **We benchmarked multicast forwarding from two ports up to 64**. Each test was executed over a 30 seconds period and was repeated 5 times. These tests were executed for 10, 20 and 30 thousands packets per second.

In Figure 58 we observe that usage increases linearly with the number of *tags* but increases exponentially with *packet rate*. This is caused by hardware interrupt overload and failover to software. This happens as we are only benchmarking a single core. In general CPU usage is low, and, since Open vSwitch is multithreaded, it can easily scale out to support the entire bandwidth of a 10Gbps link[67]. This proves that our VLAN-PSSR solution is feasible.

Also, pushing 64 tags is an unrealistic usage scenario, since, in a data center, usually a packet is routed through no more than 5 - 6 hops. This means that for unicast the maximum number of tags would be 6. For 64 ports multicast, in the same conditions it would add another 8 tags for a maximum of 14. In this case the packet rate can go up to 300 thousands pps / core yet a good value, without interrupt overload, is around 200 thousands pps / core. Therefore, on a 16 core system, with CPU usage below 20% on all

[66] First 8 cores are on first NUMA node while the other 8 are on second node so we chose core 0 for Open vSwitch and 9 for all other processes.

[67] We tested that but correct performance statistics are hard to obtain once we remove single core isolation.

cores it can easily provide a theoretical limit above 3.2 million pps. This multiplied by 512B (packet average size), leads to 11Gbps, above the 10Gbps threshold of a single link.

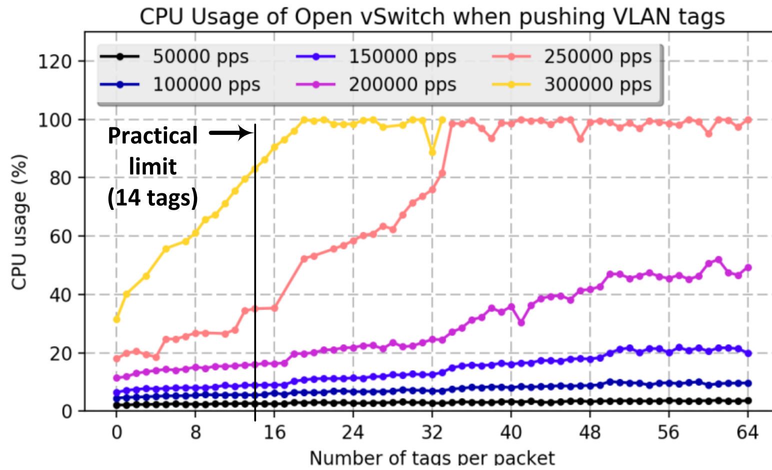


Figure 58: Performance when pushing tags at edge on a single CPU core

For multicast case, in Figure 59, CPU usage is much higher. For only 30 thousand packets it can consume around 30% of a single core, which is a very high number. This is mainly caused by the fact that packets with more than one tag are moving multiple times through the same pipeline (one pass per tag). Packet size does not have a notable impact on performance as processing is done using zero copy buffers. A single copy is done at transmission and that uses Direct Memory Access (DMA).

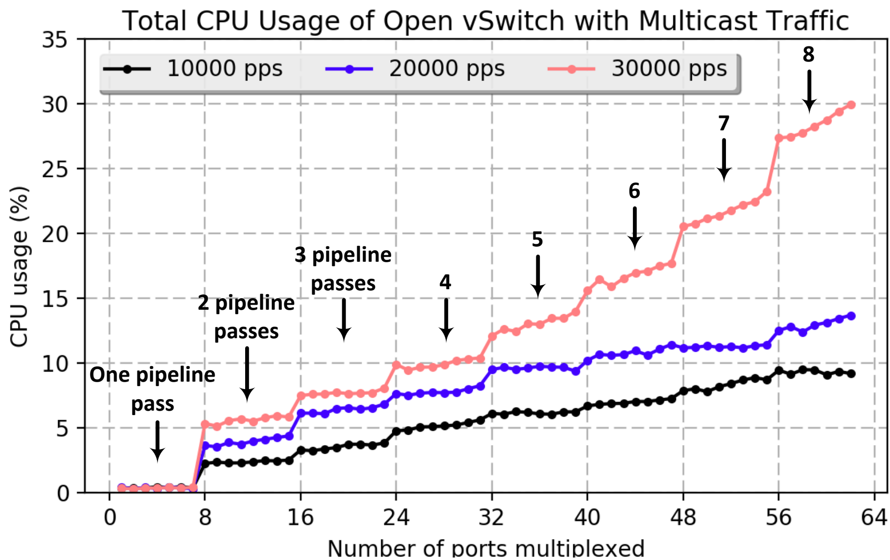


Figure 59: Multicast performance

In our benchmark transmission happened without loses but we observed that loss started to appear at just 40000 pps, for final pipeline passes (7 and 8th pass). We can speculate that this is caused by small buffering used inside Open vSwitch pipeline. So, for a single core, 30000pps seems to be the limit even

though CPU is only at 30%. With four cores we can go to 100000 (around 1.2Gbps with 1500B packets) but then too much processing is spend for packet forwarding instead of computing. After all these are servers, not switches.

As we were expecting these results prove that multicasting with source routing is not a good solution for high throughput streams when done in software[68]. Therefore, hardware acceleration of ASICs is needed.

6.7 Conclusions

Source Routing based on VLAN tagging can easily be implemented using existing OpenFlow 1.3 functionality. This solution can be enabled by default in an entire Data Center, thus simplifying flow tables in core switches or, if packet size is an issue, only when the number of flows approaches the maximum capacity (flow resources are limited by hardware). The controller can decide when to apply it. This may also be used to improve multipath routing, as packets can be directed on different paths from the source using fine grained distribution algorithms and switches in the core will not even be aware of it.

Also, given the fact that switching tables are static and if hardware customizations are possible, then very simple and fast hardware that only needs to support VLAN-PSSR can be built and this would provide an impressive cost reduction per switching unit.

[68] The software solution may still be used to isolate small broadcast domains (i.e. of a tenant) when traffic is negligible but that's all that it can do.

7 Conclusions and Future Work

7.1 Original contributions of this thesis

This thesis brings the following original contributions:

Contributions to the analysis of SDN (chapter 2):

Based on existing documentation in the field such as historical projects (section 2.6), differences between traditional and SDN networking (sections 2.1 & 2.4) and approaches to SDN (section 2.5) I proposed a relevant set of requirements and identified advantages of SDN over traditional networking (section 2.7). Also, I analyzed the activity, based on the number of code contributions per month, of ten most important open source SDN controller projects and I compared the most active two implementations: OpenDaylight and ONOS (section 2.8).

Contributions to the analysis of Data Centers (chapter 3):

In this chapter I proposed a comprehensive set of criteria for Data Centers classification that considers eight important characteristics: availability, purpose, orchestration model, service model, deployment, network topology, storage model and compute model (section 3.3). Then I continued with an analysis of Data Center design constraints imposed on networking by the four most important aspects: facility, compute, storage and management applications (section 3.3.1). Also, I provided a detailed analysis of the most common deployed Data Center networking architecture: The Hierarchical Model (section 3.3.3). I analyzed how it restricts our choice of topologies, in most cases, to a single type: the Clos topology with two of its variants - Fat-tree and Leaf-spine. Furthermore, I compared the two topologies with other two that are less used, *torus* and *hypercube*, from a height, diameter and scalability perspective and, based on this analysis, I provided recommendations on best usage scenarios for each of them (section 3.5.3) [ANDR, 2015]. An example of a *fat-tree* topology is presented in [PIST, 2014].

Most of the data center traffic is created by *compute* and *storage* servers or arrives from the outside world. For better understanding the storage traffic I benchmarked a real distributed storage solution based on Ceph, analyzed its impact on networking and provided recommendation on storage uplink capacity scaling (section 3.4.3). Then, regarding traffic originating in servers and outside traffic, I proposed a comprehensive set of classification criteria of its patterns inside a data center (section 3.5.1). Also, I analyzed the communication pattern of OpenStack, a *microservices* Cloud Management Platform, which concludes that most of its traffic is composed of small packets that would benefit from fast paths with short delays and few packet losses (section 3.5.2).

Design and implementation of the Contron NS-3 OpenFlow controller (chapter 4):

The main contribution of this chapter is a new controller design adapted for network simulators. While there are many controllers that can be used either for researching new SDN features (e.g. Ryu or Pox), or for production deployments (e.g. ONOS or OpenDaylight), there is no implementation for an open source networking simulator. This proved to be a problem as physical or emulated setups are unable to provide a controlled environment where we could reliably reproduce the same

conditions that would generate a particular set of results. Furthermore, complexity of software in real deployments, scarcity of hardware and lack of performance fidelity in emulated (virtual) environments aggravate the problem.

Therefore, to solve it, I proposed a new OpenFlow controller integrated with NS-3 that is simple, modular and, through its abstractions and services for managing OpenFlow switches, provides a controlled SDN environment optimized for experimentation and research.

This new controller was used in my research of network congestion and, besides this, represents a good platform for validating solutions in SDN areas such as: scalability, efficient load balancing, service-chaining, quality of service, multipath routing and integration with classic networking.

Contron provides a global and centralized view of network topology as a graph of nodes (section 4.4.1). It has a *Simple Topology Discovery Service* that populates this topology with hosts and switches (section 4.4.2), a *Flow Abstraction Component* that allows to easily add and remove flows (section 4.4.3), a *Connection Tracker* that monitors connections between hosts (section 4.4.7), a *Statistics Service* to monitor switches (section 4.4.4), a *Proxy* to manage ARP requests and responses (section 4.4.6) and a *Shortest Path Forwarding Service* that uses Dijkstra's shortest path algorithm for connecting nodes with flow based routes (section 4.4.7).

Next, using Contron I made a case study on migrating traffic away from congested links in a Leaf-spine topology (section 4.5 and 4.6). I observed its most important effects: (1) solving congestion on one link tends to move it to the next link in the route, (2) high packet reordering and (3) slow reaction times of control plane. Then I proposed three solutions to these problems: increase the responsiveness of control plane, increase the size of the queues in switches or use a congestion notification mechanism (section 4.7). The last one is detailed in chapter 5.

Results of the research described in this chapter were presented in [PONC, 2016/1].

Reducing network congestion in Data Centers by leveraging SDN, QCN and WFQR indicatives (chapter 5):

In this chapter I proposed several methods of reducing network congestion in Data Centers by adapting Quantified Congestion Notification (QCN) and Weighted Flow Queue Ranking (WFQR) indicatives to SDN.

QCN reduces congestions by monitoring transmit queues (buffers) on a switch ports and, if usage is above a specified limit, it slows down transmission at the traffic source to avoid queue overruns. Besides adapting QCN to SDN, my proposal improves the original QCN mechanism by providing controllers the capability to migrate flows to less congested paths thus improving transmission speed (section 5.5). Moreover, I proposed a load balancing method based on congestion information that increases network performance by exposing these indicatives to SDN applications (section 5.4).

In section 5.3 I proposed three models of adapting QCN communication to SDN, which have advantages and disadvantages, depending on existing controller capabilities, desired performance, and required development effort.

The first model provides minimal integration (section 5.3.1). It keeps QCN protocol in place and improves it by computing congestion indicatives in switches and forwarding them to controllers. Its main advantage is reliance on the validated and standardized QCN mechanism therefore, minimal integration effort is required. The solution has three downsides. First, software in switches has to be modified to keep congestion indicatives. Second, the southbound interface also needs to be updated to forward these indicatives to the controller (e.g. in OpenFlow). Third, the real time decisions are harder to make since congestion indicatives are computed by switches and they are received by controllers with some delay.

The second model adds real-time decision capabilities and removes the necessity of making modifications to southbound interface by forwarding congestion notifications directly to controllers as OpenFlow *packet-in* messages. However, it has its own limitations: controller may delay congestion messages, it may be overloaded by them and can congest the network management. Luckily, these issues can be avoided with careful design (section 5.3.2).

The third model provides full integration and goes further away from QCN to the point that its implementation according to the standard is no longer required. The model has two downsides: it only works with edge virtual switches and requires much more design and implementation effort, as packet processing is done entirely by the controller and needs to be fast (section 5.3.3).

Results of this research were published in [PIST, 2015].

A Source Routing solution in SDN using stacked VLAN tags: VLAN-PPSR (chapter 6):

In this chapter I propose a Source Routing Packet Forwarding solution for both *unicast* and *multicast,* that reduces flow table usage in core switches, improves speed of network updates and simplify multipath routing. Moreover, the proposal can be implemented with existing OpenFlow v1.3 devices without necessitating costly modifications to switches.

VLAN-PSSR solution is based on stacked VLAN tags that are pushed at edge and popped at each hop after forwarding is decided. This provides some advantages over similar solutions implemented using MPLS tags such as using smaller packet overhead and avoiding limited MPLS support in core switches (section 6.5).

Even though *multicast* is limited to a single rack, it still covers most use cases of multitenant Software Defined Data Centers and, for uncovered cases, I provide two indirect solutions that can be applied to the entire data center (section 6.5.2).

Both *unicast* (section 6.5.1) and, more importantly, *multicast* (section 6.5.2) proved to be functional. I validated both of them by connecting *Open vSwitch* topologies created in *Mininet* with *Ryu* controller and managed by my VLAN-PSSR implementation.

Performance proved to be very good for pushing tags on real hardware (Xeon class servers), with only a small percentage of a core CPU being used even in worst case scenarios (pushing 64 VLAN tags). In case of multicast forwarding, performance is less than what is desired, and proves that hardware acceleration provided by ASICs is a must for SDDCs with high multicast throughput (section 6.6). Results of this research were published in [PONC, 2016/2].

7.2 Future Work

My research into simulated SDN has focused on implementing a basic, yet functional, controller model using NS-3 existing OpenFlow implementation. This model can be further improved. First, the OpenFlow version in NS-3 should be upgraded from 0.98 to at least 1.3, then Contron itself should be extended with features for the new OpenFlow version and its existing features enhanced.

Then with the improved controller and using results gained from this thesis we can research an end-to-end multipath routing solution. This would combine multiple metrics such as: link capacity, average throughput and congestion information with flow classification and traffic characteristics to provide a fully automated and better optimized routing solution for an entire datacenter. The algorithm would be able to choose the best path for new flows, reroute existing flows to avoid congestions and optimize their throughputs and latencies.

Then, this routing solution may be implemented in a production ready controller such as ONOS or OpenDaylight to provide optimized routing. Moreover, to decrease the time of rerouting many flows and to reduce flow table usage, the multipath routing algorithm can be further extended to use source routing through VLAN-PSSR. This would also provide fine grained multipath routing from the edge.

8 ACRONYMS

API – Application Programming Interface ... 17
ASIC – application-specific integrated circuit ... 14
ATM - Asynchronous Transfer Mode ... 21
BGP - Border Gateway Protocol ... 23
BIOS – Basic Input/Output System ... 46
CAPEX - Capital Expenditure ... 11
CLI – Command Line Interface ... 36
CMMS – Computerized Maintenance Management System in Data Centers ... 49
CNM – Congestion Notification Messages ... 93
COTS - Commercial off-the-shelf ... 22
CP – Congestion Point ... 93
DAS – Direct Attached Storage ... 46
DC – Data Center ... 53
DCIM – Data Center Infrastructure Management ... 49
DDoS - Distributed Denial of Dervice ... 23
DoS – Denial of Service ... 29
EDA – Equipment Distribution Area ... 52
FC – Fibre Channel ... 55
FCoE – Fibre Channel over Ethernet ... 87
FS – File System ... 46
FUSE – Filesystem in Userspace ... 56
GPU ... 44
GUI – Graphical User Interface ... 36
HA – High Availability ... 40
HAD – Horizontal Distribution Area ... 52
HBA – Host Bus Adapter ... 55
HDD – Hard Disk Drive ... 46
HPC – High Performance Computing ... 107
IaaS – Infrastructure as a Service ... 44
iDRAC – integrated Dell Remote Access Controller ... 44
IDS – I +ntrusion Detection System ... 33
iLO – HP Integrated Lights Out ... 44
Intel ME – Intel Management Engine ... 44
IOPS – Input/Output Operations Per Second ... 36
iSCSI – Internet Small Computer System Interface ... 45
JSON – JavaScript Object Notation- ... 66
KVM – Keyboard Video Mouse ... 39
LAG - Link Aggregation Group ... 22
LV – Low Voltage ... 14
MDA – Main Distribution Area ... 52
MDS – Metadata Server ... 57

NAS- Network-attached storage 45
NFV – Network Function Virtualization 63
NIC – Network Interface Card 44
NMS – Networking Management System 15
NPUs – Network Programmable Units – 14
NUMA – Non-uniform memory access 119
ONF - Open Networking Foundation 16
OPEX - Operating expense 11
OSD – Object Storage Daemon 56
OVS – Open vSwitch 27
PaaS – Platform as a service 44
PCB – Printed Circuit Board 14
PCI – Peripheral Component Interconnect 43
PCIe – Peripheral Component Interconnect Express 46
PG – Placement Group 101
POC – Proof of Concept 24
PSSR 108
RADOS – Reliable, Autonomous, Distributed Object Store 56
RAID – Redundant Array of Independent Disks 46
RBD – Rados Block Device 56
RCP - Routing Control Platform 23
REST – REpresentational State Transfer 36
RGW – Rados Gateway 57
RP – Reaction Point 93
RSPAN – Remote Switched Port Analyzer 61
SaaS – Software as a service 45
SAN – Storage Area Network 38
SDDC – Software Defined Data Center 50
SDE – Software Defined Everything 11
SDN – Software Defined Networking 12
SD-WAN – Software Defined Wide Area Networking 48
SPAN – Switched Port Analyzer 61
SSD – Solid-state drive 46
TLV – Type Length Value datastructure 78
ToR – Top of Rack 52
ULV – Ultra Low Voltage 14
URL – Uniform Resource Locator 67
UUID – Universally unique identifier 98
VGA – Video Graphics Array 45
VM – Virtual Machine 30
WAN – Wide Area Network 48
ZDA – Zone Distribution Area 52

List of Publications

Publications connected to this thesis:

[PONC, 2016/1] Poncea, Ovidiu Mihai, Andrei Pistirica, Florica Moldoveanu, Victor Asavei. "Design and Implementation of an Openflow SDN Controller in NS-3 Discrete-Event Network Simulator*", International Journal of High Performance Computing and Networking (IJHPCN)*, accepted for publication.

[PIST, 2016/1] Pistirica, Sorin Andrei, Ovidiu Poncea, and Mihai Claudiu Caraman. "QCN Based Dynamically Load Balancing: QCN Weighted Flow Queue Ranking." In *2015 20th International Conference on Control Systems and Computer Science*, pp. 197-204. IEEE, 2015. (ISI)

[PONC, 2016/2] Poncea, Ovidiu Mihai, Florica Moldoveanu, Victor Asavei. "VLAN-PSSR: Port-Switched based Source Routing using VLAN tags in SDN Data Centers." University POLITEHNICA of Bucharest Scientific Bulletin, C series: Electrical Engineering and Computer Science, accepted for publication.

[ANDR, 2015] Andrus, Bogdan, Ovidiu Mihai Poncea, JJ Vegas Olmos, and I. Tafur Monroy. "Performance evaluation of two highly interconnected Data Center networks." In *2015 17th International Conference on Transparent Optical Networks (ICTON)*, pp. 1-4. IEEE, 2015 (ISI)

[PIST, 2014] Pistirica, Sorin Andrei, Ovidiu Mihai Poncea, Victor Asavei, and Alexandru Egner. "IMPACT OF DISTRIBUTED FILE SYSTEMS AND COMPUTER NETWORK TECHNOLOGIES IN ELEARNING ENVIRONMENTS." In *The International Scientific Conference eLearning and Software for Education*, vol. 4, p. 85. " Carol I" National Defence University, 2014 (ISI)

Other publications:

[PONC, 2011] Ovidiu Mihai Poncea. "Integrarea Multinivel a Informațiilor de Rutare si Tactice in Rețele Ad-hoc de Capacitate Redusă". Sesiunea de comunicări științifice cu tema Cercetarea științifică militară în sprijinul interoperabilității*,* ACTTM 2011

[SIM, 2015] Simion, Andrei, Victor Asavei, Sorin Andrei Pistirica, and Ovidiu Poncea. "Practical GPU and Voxel-Based Indirect Illumination for Real Time Computer Games." *In 2015 20th International Conference on* Control *Systems and Computer Science*, pp. 379-384. IEEE, 2015 (ISI)

[GREU, 2012] Greu, Victor, Petrica Ciotirnae, Constantin Vizitiu, Sorin Cernea, and Ovidiu Poncea. "A secure routing algorithm with additional cognitive information scalable features for the design approach of the tactical frequency hopping radios Ad-hoc networks (TAFHNET)." In *2012 9th International Conference on Communications (COMM)*. pp. 181 – 184, IEEE 2012 (ISI)

REFERENCES

[1] "Internet Users," [Online]. Available: http://www.internetlivestats.com/internet-users/. [Accessed 28 February 2016].

[2] Cisco, "Visual Networking Index," 2015. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html. [Accessed 15 February 2016].

[3] I. Pepelnjak, "OPEN VSWITCH PERFORMANCE REVISITED," 21 11 2014. [Online]. Available: http://blog.ipspace.net/2014/11/open-vswitch-performance-revisited.html. [Accessed 23 02 2016].

[4] B. P. E. J. Justin Pettit, "Accelerating Open vSwitch to “Ludicrous Speed”," 13 11 2014. [Online]. Available: http://networkheresy.com/2014/11/13/accelerating-open-vswitch-to-ludicrous-speed/. [Accessed 23 03 2016].

[5] Mellanox Technologies, "NP/NPS family of NPUs," Mellanox, [Online]. Available: http://www.mellanox.com/page/npu_overview. [Accessed 23 02 2016].

[6] NXP, "Network Services Switching Platform," Freescale/NXP, [Online]. Available: http://www.nxp.com/products/microcontrollers-and-processors/power-architecture-processors/qoriq-processors-power-architecture-t-series/network-services-switching-platform:NETWORK-SERVICES-SWITCH. [Accessed 23 02 2016].

[7] "Wikipedia: Packet Processing#Network Processors," [Online]. Available: https://www.wikiwand.com/en/Packet_processing#Network_processors. [Accessed 23 02 2016].

[8] G. Hedfors, "Owning the data centre, Cisco NX-OS," 18 03 2011. [Online]. Available: https://media.blackhat.com/bh-eu-11/George_Hedfors/BlackHat_EU_2011_Hedfors_Owning_the_datacenter-Slides.pdf. [Accessed 22 02 2016].

[9] S. M. Kerner, "Juniper Embraces Linux for Junos and Open Compute for New Switch," Enterprise Networking Planet, 03 12 2014. [Online]. Available: http://www.enterprisenetworkingplanet.com/nethub/juniper-embraces-linux-and-open-compute-project-for-new-switch.html. [Accessed 22 02 2016].

[10] Wind River Systems Inc., "VxWorks 6.4 BSP for Broadcom BCM95331X Network Switch Reference Designs," [Online]. Available: https://bsp.windriver.com/index.php?bsp&on=details&bsp=5900. [Accessed 22 02 2016].

[11] "Wikipedia: VxWorks - Networking and communication infrastructure," [Online]. Available: https://www.wikiwand.com/en/VxWorks#Networking_and_communication_infrastructure. [Accessed 22 02 2016].

[12] Leroux, Paul, QNX Software Systems, "25 Ways QNX Touches Your Life," [Online]. Available: http://www.qnx.com/news/pr_1329_3.html. [Accessed 22 02 2016].

[13] Open Networking Foundation, "Software-Defined Networking: The new Norm for Networks," Open Networking Foundation, 13 04 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf. [Accessed 23 02 2016].

[14] N. McKeown, "How SDN will shape the world," 18-19 10 2011. [Online]. Available: https://www.youtube.com/watch?v=c9-K5O_qYgA. [Accessed 22 02 2016].

[15] C. Black, "What is 'real SDN'?," October 2014. [Online]. Available: http://searchsdn.techtarget.com/answer/What-is-real-SDN. [Accessed 23 02 2016].

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker and J. Turner, "OpenFlow: Enabling Innovation," *ACM SIGCOMM Computer Communication Review,* vol. 38, no. 2, pp. 69-74, 2008.

[17] R. Basinger, M. Berger, E. Prell, V. Ransom and J. Williams, "Stored program controlled network: Calling card service—overall description and operational characteristics," *Bell System Technical Journal,* vol. 61, no. 7, pp. 1655-1674, 1982.

[18] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," in *DARPA Active Networks Conference and Exposition, 2002. Proceedings*, San Francisco, US, 2002.

[19] J. E. Van der Merwe, S. Rooney, L. Leslie and S. Crosby, "The Tempest: A Practical Framework for Network Programmability," *IEEE network,* vol. 12, pp. 20-28, 1998.

[20] L. Yang, R. Dantu, T. Anderson and R. Gopal, *RFC3746: Forwarding and Control Element Separation (ForCES) Framework,* IETF, 2004.

[21] A. Bavier, N. Feamster, M. Huang, L. Peterson and J. Rexford, "In VINI Veritas: Realistic and Controlled Network Experimentation," *ACM SIGCOMM Computer Communication Review,* vol. 36, no. 4, pp. 3-14, 2006.

[22] XORP, Inc, "XORP Extensible Opensource Routig Platform," [Online]. Available: http://www.xorp.org/. [Accessed 22 02 2016].

[23] "Linux VServers project," [Online]. Available: http://linux-vserver.org. [Accessed 22 02 2016].

[24] "uml_switch Manual page," [Online]. Available: http://manpages.ubuntu.com/manpages/dapper/man1/uml_switch.1.html. [Accessed 22 02 2016].

[25] The PlanetLab Consortium, "Planet Lab Web Site," [Online]. Available: https://www.planet-lab.org/. [Accessed 22 02 2016].

[26] "Operating-system-level virtualization," [Online]. Available: https://www.wikiwand.com/en/Operating-system-level_virtualization. [Accessed 22 02 2016].

[27] "The Click Modular Router Project," [Online]. Available: http://www.read.cs.ucla.edu/click/. [Accessed 23 02 2016].

[28] Alexrk2, Artist, *based on "Europe map" (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons.* [Art].

[29] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown and S. Shenker, "Ethane: Taking Control of the Enterprise," *ACM SIGCOMM Computer Communication Review,* vol. 37, no. 4, pp. 1-12, 2007.

[30] 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,* vol. 38, no. 3, pp. 105-110, 2008.

[31] The Linux Foundation, "Open vSwitch Web Page," [Online]. Available: http://openvswitch.org. [Accessed 23 02 2016].

[32] "Mininet: An Instant Virtual Network on your Laptop (or other PC)," Mininet Team, [Online]. Available: http://mininet.org. [Accessed 23 02 2016].

[33] Open Networking Foundation, "ONF Overview," [Online]. Available: https://www.opennetworking.org/about/onf-overview. [Accessed 23 02 2016].

[34] Hoelzle, Urs @ Open Networking Summit 2012, "OpenFlow @ Google," ONF, [Online]. Available: https://www.youtube.com/watch?v=VLHJUfgxEO4. [Accessed 23 02 2016].

[35] R. Merritt, "Google describes its OpenFlow network," 17 04 2012. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1261562. [Accessed 23 02 2016].

[36] The Linux Foundation, "The OpenDaylight Platform," [Online]. Available: https://www.opendaylight.org. [Accessed 23 02 2016].

[37] Open Networking Lab (ON.Lab), "ONOS: Open Networking Operating System," [Online]. Available: http://onosproject.org. [Accessed 23 02 2016].

[38] "The Linux Foundation," [Online]. Available: https://www.linuxfoundation.org. [Accessed 23 02 2016].

[39] OpenHUB, "The OpenDaylight Open Source Project on Open Hub," [Online]. Available: https://www.openhub.net/p/opendaylight. [Accessed 23 02 2016].

[40] "Open Networking Lab (ON.Lab)," [Online]. Available: http://onlab.us. [Accessed 23 02 2016].

[41] D. Pitt, "What's Ahead For SDN In 2016," 14 12 2015. [Online]. Available: http://www.networkcomputing.com/networking/whats-ahead-sdn-2016/778103327. [Accessed 23 02 2016].

[42] T. P. Morgan, "A Rare Peek Into The Massive Scale of AWS," 14 11 2014. [Online]. Available: http://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/. [Accessed 23 02 2016].

[43] The Apache Software Foundation, "Apache Karaf," [Online]. Available: http://karaf.apache.org. [Accessed 23 02 2016].

[44] ONF, "OpenFlow Switch Specification - Version 1.5.1," 26 03 2015. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf. [Accessed 23 02 2016].

[45] P. Mell and T. Grance, "The NIST definition of cloud computing," Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 09 2011. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf. [Accessed 23 02 2016].

[46] J. Hamilton, "Overall Data Center Costs," 2010. [Online]. Available: http://perspectives.mvdirona.com/2010/09/overall-data-center-costs/. [Accessed 23 02 2016].

[47] Emerson Liebert Corporation, "Lifecycle Costing for Data Centers: Determining the True Costs of Data Center Cooling," [Online]. Available: goo.gl/x0Tunu.

[48] Telecommunication Industry Association and others, "TIA-942: Data Center Standards Overview," 2006. [Online]. Available: https://manuais.iessanclemente.net/images/9/9f/Tia942.pdf. [Accessed 23 02 2016].

[49] Open Compute Project Foundation, "Open Compute Project," [Online]. Available: http://www.opencompute.org. [Accessed 23 02 2016].

[50] C. Williams, "AMD's 64-bit ARM server chip Seattle finally flies the coop ... but where will it call home? Long-awaited CPU in build systems, SDN gear for now," The Register, 14 01 2016. [Online]. Available: http://www.theregister.co.uk/2016/01/14/amd_arm_seattle_launch/. [Accessed 23 02 2016].

[51] A. Shah, "Processor delays hurt ARM server adoption, Dell exec says," PC World, 04 08 2014. [Online]. Available: http://www.pcworld.com/article/2461180/processor-delays-hurt-arm-server-adoption-dell-exec-says.html. [Accessed 23 02 2016].

[52] T. P. Morgan, "Applied Micro Chases Xeons With X-Gene 3 And NUMA," Next Platform, 18 11 2015. [Online]. Available: http://www.nextplatform.com/2015/11/18/applied-micro-chases-xeons-with-x-gene-3-and-numa/. [Accessed 23 02 2016].

[53] A. Shah, "Server vendors tap ARM chips to give users alternative to Intel," NetworkWorld, 17 11 2015. [Online]. Available: http://www.networkworld.com/article/3005744/server-vendors-tap-arm-chips-to-give-users-alternative-to-intel.html. [Accessed 23 02 2016].

[54] HP, "Server remote management with HPE Integrated Lights Out (iLO)," [Online]. Available: http://www8.hp.com/us/en/products/servers/ilo/. [Accessed 23 02 2016].

[55] Dell, "iDRAC with Lifecycle Controller," [Online]. Available: http://www.dell.com/learn/us/en/16/solutions/integrated-dell-remote-access-controller-idrac. [Accessed 23 02 2016].

[56] Cisco, "Cisco Networking Academy Connecting Networks Companion Guide: Hierarchical Network Design," Cisco Networking Academy, [Online]. Available: http://www.ciscopress.com/articles/article.asp?p=2202410&seqNum=4. [Accessed 23 02 2016].

[57] S. Lowe, "IDF 2014: Open Source Storage Optimizations," 11 09 2014. [Online]. Available: http://blog.scottlowe.org/2014/09/11/idf-2014-open-source-storage-optimizations/. [Accessed 23 02 2016].

[58] "Qemu, Open Source Processor Emulator," [Online]. Available: http://wiki.qemu.org/Main_Page. [Accessed 23 02 2016].

[59] "Kernel Virtual Machine," [Online]. Available: http://www.linux-kvm.org/page/Main_Page. [Accessed 23 02 2016].

[60] "CephFS Client on Win32 based on Dokan 0.6.0," [Online]. Available: https://github.com/ketor/ceph-dokan. [Accessed 23 02 2016].

[61] B. Lincoln, "Experiences with Ceph at the US ATLAS Midwest Tier 2 Center," [Online]. Available: https://indico.cern.ch/event/247864/contributions/1570321/attachments/426681/592250/ceph-hepix-v2.pdf. [Accessed 23 02 2016].

[62] WintelGuy.com, "Disk Performance Basics," 06 04 2013. [Online]. Available: http://wintelguy.com/2013/20130406_disk_perf.html. [Accessed 26 02 2016].

[63] N. Brownlee and K. C. Claffy, "Understanding Internet traffic streams: dragonflies and tortoises," *IEEE Communications magazine,* vol. 40, no. 10, pp. 110-117, 2002.

[64] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems (TOCS),* vol. 21, no. 3, pp. 270-313, 2003.

[65] K. Papagiannaki, N. Taft, S. Bhattacharyya, P. Thiran, K. Salamatian and C. Diot, "A pragmatic definition of elephants in internet backbone traffic," *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment,* pp. 175-176, 6-8 11 2002.

[66] K. Lan and J. Heidemann, "On the correlation of internet flow characteristics," Technical Report ISI-TR-574, USC/ISI, 2003.

[67] Cisco, "Monitor Microbursts on Cisco Nexus 5600 Platform and Cisco Nexus 6000 Series Switches," 15 09 2014. [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/switches/nexus-5000-series-switches/white-paper-c11-733020.html. [Accessed 23 02 2016].

[68] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson and S. Seshan, "Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems," in *6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, US, 2008.

[69] Y. Chen, R. Griffith, J. Liu, R. H. Katz and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, Barcelona, Spain, 2009.

[70] E. Krevat, V. Vasudevan, A. Phanishayee, D. G. Andersen, G. R. Ganger, G. A. Gibson and S. Seshan, "On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems," in *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, New York, NY, US, 2007.

[71] W. John and S. Tafvelin, "Analysis of internet backbone traffic and header anomalies observed," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, New York, NY, US, 2007.

[72] D. a. K. T. Murray, "The state of enterprise network traffic in 2012," in *18th Asia-Pacific Conference on Communications (APCC)*, Jeju Island, Korea, 2012.

[73] T. Bray, *RFC7159: The JavaScript Object Notation (JSON) Data Interchange Format,* 2014.

[74] Ecma, Standard, "ECMA-404: The JSON Data Interchange Format," 2015. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. [Accessed 23 02 2016].

[75] B. Andrus, O. M. Poncea, J. V. Olmos and I. T. Monroy, "Performance evaluation of two highly interconnected Data Center networks," in *17th International Conference on Transparent Optical Networks (ICTON)*, Budapest, Hungary, 2015.

[76] B. Lantz, B. Heller and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Monterey, CA, 2010.

[77] Riverbed, "SteelCentral Riverbed Modeler," [Online]. Available: http://www.riverbed.com/gb/products/steelcentral/steelcentral-riverbed-modeler.html. [Accessed 23 02 2016].

[78] SDx Central, "EstiNet 9.0 OpenFlow Network Simulator and Emulator, EstiNet Technologies Inc.," [Online]. Available: https://www.sdxcentral.com/products/estinet-8-1-openflow-network-simulator-and-emulator/. [Accessed 23 02 2016].

[79] NS-3 Consortium, "NS-3 Homepage," [Online]. Available: https://www.nsnam.org. [Accessed 23 02 2016].

[80] NS-3 Consortium, "NS-3 Tracing," [Online]. Available: https://www.nsnam.org/docs/manual/html/tracing.html. [Accessed 23 02 2016].

[81] NS-3 Consortium, "NS-3 Object Model - Aggregation," [Online]. Available: https://www.nsnam.org/docs/release/3.24/manual/html/object-model.html#aggregation. [Accessed 23 02 2016].

[82] NS-3 Consortium, "NS-3 Callbacks," [Online]. Available: https://www.nsnam.org/docs/release/3.24/manual/html/callbacks.html. [Accessed 23 02 2016].

[83] F. Pakzad, M. Portmann, W. L. Tan and J. Indulska, "Efficient topology discovery in software defined networks," in *Signal Processing and Communication Systems (ICSPCS), 2014 8th International Conference on*, Gold Coast, Australia, 2014.

[84] J. Tourrilhes, P. Sharma and S. a. P. J. Banerjee, "The Evolution of SDN and OpenFlow: A Standards Perspective," *IEEE Computer Society,* vol. 47, no. 11, pp. 22-29, 2014.

[85] NS-3 Consortium, "NetAnim Homepage," [Online]. Available: https://www.nsnam.org/wiki/NetAnim. [Accessed 23 02 2016].

[86] "Gnuplot Homepage," [Online]. Available: http://www.gnuplot.info. [Accessed 23 02 2016].

[87] A. Vishwanath, V. Sivaraman and M. Thottan, "Perspectives on router buffer sizing: recent results and open problems," *ACM SIGCOMM Computer Communication Review,* vol. 39, no. 2, pp. 24-29, 2009.

[88] J. Warner, "Size of switch buffers," [Online]. Available: https://people.ucsc.edu/~warner/buffer.html. [Accessed 28 02 2016].

[89] Arista Networks, "Deploying IP Storage," 2014. [Online]. Available: http://goo.gl/Z9w0kp. [Accessed 10 02 2016].

[90] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on networking,* vol. 1, no. 4, pp. 397-413, 1993.

[91] R. Edwards, Y. Ye, A. Kaul and J. Yu, "Characterization of SDN Switch Response Time in Proactive Mode," in *Proceedings of the Open Networking Summit (ONS)*, Santa Clara, CA, US, 2014.

[92] Y. Zhao, L. Iannone and M. Riguidel, "On the performance of SDN controllers: A reality check," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, 2015.

[93] Z. K. Khattak, M. Awais and A. Iqbal, "Performance evaluation of OpenDaylight SDN controller," in *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Hsinchu, Taiwan, 2014.

[94] "OpenDaylight Performance Stress Tests Report v1.0: Lithium vs Helium Comparison," Intracom Telecom, 2015.

[95] IEEE, "802.1Qau - Congestion Notification," [Online]. Available: http://www.ieee802.org/1/pages/802.1au.html. [Accessed 23 02 2016].

[96] K. J. Astrom and R. M. Murray, Feedback systems: an introduction for scientists and engineers, Princeton university press, 2010.

[97] M. Alizadeh, A. Kabbani, B. Atikoglu and B. Prabhakar, "Stability analysis of QCN: the averaging principle," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, San Jose, CA, US, 2011.

[98] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE,* vol. 103, no. 1, pp. 14-76, 2015.

[99] P. Geoffray and T. Hoefler, "Adaptive routing strategies for modern high performance networks," in *16th IEEE Symposium on High Performance Interconnects*, Stanford, CA, US, 2008.

[100] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll and E. Frachtenberg, "The quadrics network (qsnet): High-performance clustering technology," in *Hot Interconnects 9*, Stanford, CA, US, 2001.

[101] Intenet Standard, "RFC 791," 1981. [Online]. Available: https://tools.ietf.org/html/rfc791#page-1.

[102] S. A. Jyothi, M. Dong and P. Godfrey, "Towards a flexible data center fabric with source routing," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, Santa Clara, CA, US, 2015.

[103] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav and G. a. o. Varghese, "CONGA: Distributed congestion-aware load balancing for datacenters," *ACM SIGCOMM Computer Communication Review,* vol. 44, no. 4, pp. 503-514, 2014.

[104] K. Agarwal, C. Dixon, E. Rozner and J. Carter, "Shadow macs: Scalable label-switching for commodity ethernet," in *Proceedings of the third workshop on Hot topics in software defined networking*, Chicago, IL, US, 2014.

[105] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao and C. Guo, "Explicit path control in commodity data centers: Design and applications," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Santa Clara, CA, US, 2015.

[106] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *ACM SIGCOMM Computer Communication Review,* vol. 44, no. 4, pp. 307-318, 2014.

[107] R. M. Ramos, M. Martinello and C. E. Rothenberg, "SlickFlow: Resilient source routing in data center networks unlocked by OpenFlow," in *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, Sydney, Australia, 2013.

[108] M. Soliman, B. Nandy, I. Lambadaris and P. Ashwood-Smith, "Source routed forwarding with software defined control, considerations and implications," in *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, Nice, France, 2012.

[109] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Co-NEXT '10 Proceedings of the 6th International COnference*, New York, NY, US, 2010.

[110] Wireshark Foundation, "Wireshark Home Page," [Online]. Available: https://www.wireshark.org. [Accessed 23 02 2016].

[111] Ryu SDN Framework Community, "Ryu SDN Framework," [Online]. Available: https://osrg.github.io/ryu/.

[112] "LinuxContainers," [Online]. Available: https://linuxcontainers.org.

[113] L. Deri, "nCap: Wire-speed packet capture and transmission," in *Workshop on End-to-End Monitoring Techniques and Services*, Nice, France.

[114] ntop, "PF_RING: High-speed packet capture, filtering and analysis," [Online]. Available: http://www.ntop.org/products/packet-capture/pf_ring/.

[115] Z. Majo and T. R. Gross, "Memory system performance in a NUMA multicore multiprocessor," in *Proceedings of the 4th Annual International Conference on Systems and Storage*, Haifa, Israel, 2011.

[116] Arista Networks, "Deploying IP Storage," 2014. [Online]. Available: https://www.arista.com/assets/data/pdf/Whitepapers/Deploying_Storage_Net_WhitePaper.pdf. [Accessed 10 02 2016].

[117] IBM, "First Commercial IBM Hot-Water Cooled Supercomputer to Consume 40% Less Energy," 18 06 2012. [Online]. Available: http://www-03.ibm.com/press/us/en/pressrelease/38065.wss. [Accessed 23 02 2016].

[118] L. Bell, "Iceotope fully immersed liquid cooled servers," The Inquirer, 20 05 2013. [Online]. Available: http://www.theinquirer.net/inquirer/news/2269056/iceotope-shows-off-fully-immersed-liquid-cooled-servers-video. [Accessed 23 03 2016].

[119] Wikipedia, "Thermal wheel," [Online]. Available: https://www.wikiwand.com/en/Thermal_wheel. [Accessed 23 02 2016].

[120] Wikipedia, "Evaporative cooler," [Online]. Available: https://www.wikiwand.com/en/Evaporative_cooler. [Accessed 23 02 2016].

[121] Google Inc., "Data Center Efficiency: How we do it," [Online]. Available: http://www.google.ro/about/datacenters/efficiency/internal/. [Accessed 23 02 2016].

[122] "Git Source Control Management," [Online]. Available: https://git-scm.com. [Accessed 23 02 2016].

[123] The Apache Software Foundation, "Apache Subversion," [Online]. Available: https://subversion.apache.org. [Accessed 23 02 2016].

[124] "Mercurial Source Control Management," [Online]. Available: https://www.mercurial-scm.org. [Accessed 23 02 2016].

[125] S. A. Pistirica, O. Poncea and M. C. Caraman, "QCN Based Dynamically Load Balancing: QCN Weighted Flow Queue Ranking," in *20th International Conference on Control Systems and Computer Science*, Bucharest, Romania, 2015.