Insert here your thesis' task.

Czech Technical University in Prague

Faculty of Information Technology

Department of Computer Systems

Master's thesis

Enhancing Availability of Services in Multi-Tenant Environment Using Software-Defined Networking

Bc. Martin Klepáč

Supervisor: Ing. Tomáš Hégr

29th April 2015

Acknowledgements

I would like to thank my supervisor, Ing. Tomáš Hégr, and the entire SDNLabs.cz organisation for their support during the past 6 months.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 29th April 2015 . . . .

Citation of this thesis

Klepáč, Martin. *Enhancing Availability of Services in Multi-Tenant Environment Using Software-Defined Networking.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Rapídny nárast klientskych požiadavkov na poskytovateľov služieb dátového centra prípadne cloudu sa ocitá v rozpore s tradičnými, málo pružnými konceptami fungovania sietí. V záujme zachovania flexibility, ktoré dátové centrá svojim zákazníkom sľubujú, je potrebné odstránit' tento rozpor, napríklad s využitím inovatívneho konceptu *softvérovo definovaných sietí.* Táto diplomová práca využíva tento koncept pri snahe zlepšit' proces živej migrácie virtuálnych strojov tak, aby nedostupnost' služieb behom migrácie bola minimalizovaná. Práca je zameraná na oblast' malých až stredne veľkých dátových centier. Z tohto dôvodu sú výstupy práce založené na reálnych modeloch komunikácie v danom prostredí. Výsledkom tejto práce a implementácie príslušného programu je zistenie, že latencia a stratovost' paketov sa vd'aka proaktívnemu prístupu k riadeniu siete dokáže mierne znížit' v porovnaní s klasickým prístupom založeným na L2 prepínaní, čím dochádza k zvýšeniu dostupnosti služieb bežiacich na virtuálnom stroji, ktorý podstupuje živú migráciu.

Klúčové slová SDN, softvérovo definované siete, živá migrácia, ONOS, SDN kontrolér, virtuálny switch, Open vSwitch, overlay siete, orchestrátor, OpenNebula

Abstract

The immense growth of client requirements imposed on data center and cloud providers results in a conflict with traditional networking concepts lacking the required agility. In order to promote flexibility, which data center providers promise to their clients, this discrepancy needs to be resolved, for instance by employing the novel concept of *Software Defined Networking* (SDN). This diploma thesis utilises this concept in order to minimise service downtime while performing live virtual machine migration. The work is aimed at small/medium-sized data centers and hence the findings are based on real communication patterns found within such environments. The results of the thesis and implemented application show that latency and packet loss is slightly diminished thanks to the proactive approach taken during network topology changes when compared to the traditional approach based on L2 forwarding. Hence, the overall service availability within the virtual machine undergoing live migration is enhanced.

Keywords SDN, software defined networking, live VM migration, ONOS, SDN controller, virtual switch, Open vSwitch, overlay networks, orchestrator system, OpenNebula

Contents

List of Figures

List of Tables

Introduction

In the last couple of years business has largely focused on improving its efficiency. In terms of IT and its infrastructure, a transition from maintaining a dedicated server room with a number of physical hosts to running virtualised servers hosted by a cloud provider has taken place. Besides higher efficiency and thus financial savings there are other reasons for this change, such as accountability in terms of service level agreement between the two parties (i.e. the customer and the cloud provider), possible lack of in-depth knowledge of local IT personnel on the one hand and a degree of trust in the cloud provider with multiple positive references from large companies on the other hand.

The set of requirements held by clients affect how cloud providers organise their internal data center networking topology. Naturally, the cloud providers need to take *multi-tenancy* into account, which means that they allow for sharing of their infrastructure by multiple clients simultaneously without exposing any tenant to information disclosure.

The requirement of flexibility and scalability (for instance, expressed by building new virtual machines or connecting VM instances to a new network virtually instantaneously) must also be considered when employing a particular network design.

Networking has been relatively stable in the past few years, as it is still based on TCP/IP stack. Core principles of traffic forwarding (switching and routing at layers 2 and 3 respectively) are well documented and have not changed whatsoever. However, networking has long been known for burden of configuring every device separately. Each device, be it a switch or router, represents an independent instance with its own logic. At some time, unlimited scalability as proposed by some data center providers may eventually discover its limits due to the nature of networking device operation as described above [1].

The scalability problem defined in the previous paragraph is aimed to be solved by an emerging *Software Defined Networking* (SDN) paradigm. The purpose of the thesis is to improve service availability in SDN-based data centers mostly in conjunction with an orchestrator system, which is responsible for providing and keeping track of virtual machines on a hypervisor level. The work is aimed to fit into small or medium-sized data centers. The thesis will be divided into four main chapters, which will now be briefly described.

Chapter 1 will tackle the principles of SDN, evolution of virtual switches operating within the hypervisor host and networking protocols which cope with the multi-tenancy requirement in a more scalable fashion when compared to traditional VLAN-based approaches.

Chapter 2 will include an analysis of the subsequent implementation part. Based on the conducted research in the first chapter regarding SDN controllers, a combination of SDN controller and orchestrator system will be designated. A set of functional and non-functional requirements will be raised.

Chapter 3 will focus on the implementation itself on the basis of choice made in the previous chapter. Some pieces of the code will be presented in this chapter.

Chapter 4 will include a description of the evaluation methodology. Quantitative assessment of the resulting application in data center-like lab environment will be carried out. Its results will be compared to the traditional non-SDN-based approach.

Last but not least, based on the evaluation in chapter 4, conclusion will attempt to provide definitive results and will aim to define the extent to which SDN-based networks are a feasible alternative to currently employed networks.

Chapter 1

State-of-the-art

This chapter will provide an introduction into the concept of software-defined networks. Since the implementation of such a concept requires changes throughout the entire networking infrastructure, multiple related areas including but not limited to overlay networking protocols and virtual switches will be tackled.

1.1 SDN

The concept of software-defined networking attempts to fundamentally change the world of networking, predominantly in data centers. SDN as advocated by Open Networking Foundation is characterised by a separation of control and data planes within networking devices [2]. Every single networking device has to perform at least the following activities:

- process traffic
- interact with the neighboring environment
- provide management access

More formally, every device consists of three *planes* – *data* plane is responsible for traffic forwarding (be it switching at L2 or routing at L3), *control* plane runs a variety of protocols such as dynamic routing protocols (OSPF, BGP), neighbor discovery protocols (CDP, LLDP) or link aggregation protocols (LACP, PAgP). Last but not least, *management* plane enables administrators to configure the device via CLI or monitor its health via SNMP, for instance [3].

Nowadays, most enterprise-level devices perform data plane functionality within dedicated hardware, whereas control plane protocols are handled in software by CPU. In other words, moving control plane away from the networking device should relieve the processor of switch/router in question and allow the device to focus on traffic forwarding only. Furthermore, the transition of control plane functionality to an external entity should provide a unified view on the entire networking topology instead of every device possessing a partial view. On the other hand, the existence of a centralised *controller*, which is in charge of providing control plane functionality to all connected devices, brings about the problem of *single point of failure* (SPOF).

However, the most valuable advantage of SDN concept is the ability to programatically change the behaviour of multiple devices irrespective of their vendor at once, thus providing a degree of abstraction above the underlying networking infrastructure. This is especially beneficial in large data centers with multi-tenancy in mind, as business requirements should no longer be impeded by their lengthy realisation.

Figure 1.1 shows a high-level architecture of SDN environment. The bottom layer consists of network devices, which perform actual traffic forwarding within their local data planes. The forwarding rules are, however, downloaded from the SDN controller via controller's *southbound API*. This action is usually performed by *OpenFlow* protocol. The SDN controller may communicate with other applications such as an *orchestrator system* (OpenStack, VMware vCloud Director), which keeps track of currently active virtual machines within the tenant's environment. The orchestrator may thus provide a list of IP and MAC addresses for which the controller creates appropriate forwarding rules (for instance, forward all traffic among VMs within a given subnet) and finally downloads them to the physical infrastructure. From the perspective of the controller, applications consume its *northbound API*.

The ideas behind SDN are far from being novel. In fact, twenty years ago Open Signaling group believed that "a separation between the communication hardware and control software was necessary, but challenging to realise" [4]. Their attempt led to the specification of General Switch Management Protocol (RFC 1987) [5], but the work has been discontinued since 2002. Therefore, it is worth contemplating why a similar concept has failed in the past and what has changed ever since.

Since a single SDN controller represents a SPOF, multi-controller deployment is more preferrable. Single controller solution may also result in a significant latency between edge networking devices and the controller and hence slower deployment of forwarding rules. In the current era of horizontal scaling, *scale-out* architecture in which every controller is responsible for its zone or region appears to be more favourable when compared to vertical scaling. The drawback of scale-out architecture is state transition – when one controller becomes unavailable, another controller must take his place.

Another obstacle to SDN deployment regards capabilities of the physical hardware. As OpenFlow forwarding entries are not necessarily limited to traditional destination L2/L3 addresses only (in fact, OpenFlow entries may match almost any field from layers 2 to 4), this implies greater TCAM space consumption within physical devices. Excessive *packet punting* (effectively,

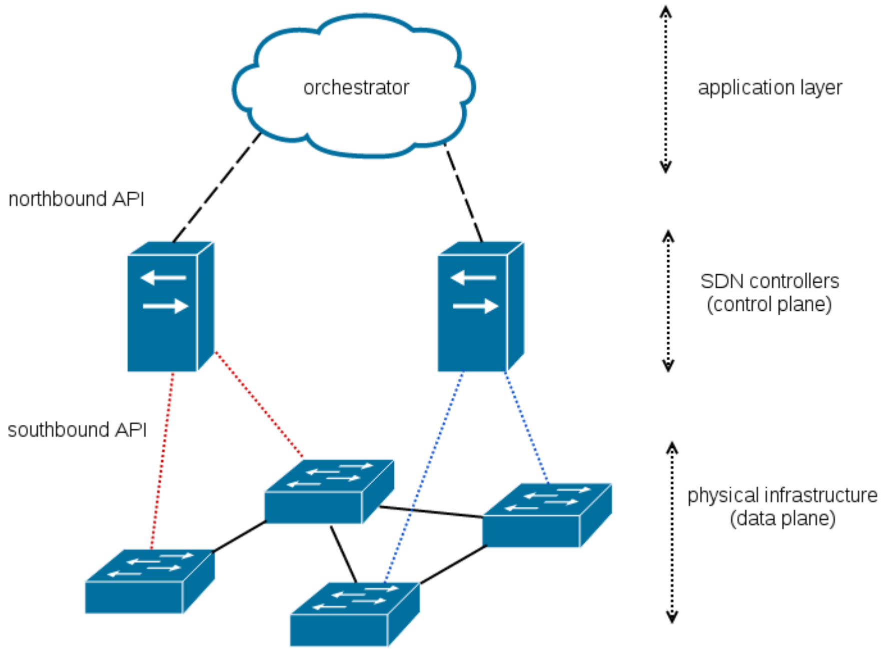


Figure 1.1: SDN high-level architecture

processing of unknown packets by CPU) and subsequent propagation of such traffic to the controller may bring network operation to a halt as well. More information regarding SDN adoption in physical networks is provided in chapter 1.4.

1.1.1 Comparison of selected SDN controllers

As the entire SDN industry is rapidly growing, no standard regarding the functionality of an SDN controller has yet been proposed. Although *OpenFlow* is considered as the de facto standard of communication between control and data planes, northbound API among SDN controllers differs significantly. Therefore, the following section will attempt to compare several SDN controllers from multiple perspectives.

Floodlight

Project Floodlight is an open source SDN controller sponsored by Big Switch Networks. It has originally been created as a fork of another SDN controller called Beacon. Floodlight is written in Java with support of both physical and virtual switches in mind. In terms of southbound API, its latest 1.0 release supports both major OpenFlow versions – 1.0 and 1.3. Northbound API is exposed via REST. Floodlight supports both *proactive* and *reactive* flow learning. While in reactive mode OpenFlow-enabled switch contacts the controller when the first packet within a flow arrives on its port and then waits for the controller's response, in proactive mode the controller downloads relevant flow entries to the device in advance, thus minimising delay.

Although Floodlight is capable of providing multiple networks, it lacks support of overlay protocols (notably VXLAN) as described in chapter 1.2. Distributed scale-out architecture is not supported either, so Floodlight controller represents a SPOF. On the other hand, the project documentation is clear and contains rich examples of both northbound and additional modules (such as stateless firewall) API.

The approach to Floodlight controller is well characterised by a former CEO of Big Switch Networks Guido Appenzeller who claims that "the enterprise version that is the Big Network Controller adds a lot of functionality that is wrapped around Floodlight" [6]. Therefore, Floodlight controller is rather suited for a proof-of-concept solution instead of production environment.

OpenDaylight

OpenDaylight is a complex SDN controller supported by virtually all renowned vendors within the networking industry. The project is currently managed by The Linux Foundation, which attempts to make OpenDaylight "a core component within any SDN architecture" [7]. The current version of OpenDaylight is codenamed Helium.

Since OpenDaylight acts as a framework for SDN deployment, it supports a large variety of protocols in southbound direction (OpenFlow 1.0 and 1.3, XMPP, Netconf among others). In terms of northbound API, applications are allowed to consume REST and OSGi interfaces.

HA-wise, several instances of OpenDaylight controller may act as a single logical controller due to the east-west state synchronisation among the controllers. OpenFlow-enabled switches in such an environment are required to connect to multiple controller instances. OpenFlow 1.2 specifies two modes of multi-controller operation: equal (all controllers have read-write access to the device) or master-slave, in which the access is limited to a single controller at any time [8].

OpenDaylight gives network administrators an opportunity to include numerous modules. Virtual Tenant Network, for instance, is an application that allows users to define a virtual network and subsequently automatically maps the virtual network into the underlying infrastructure using SDN control plane [9].

Unlike Floodlight, which lacks overlay network support, OpenDaylight provides an optional OpenDOVE plugin, which realises the multi-tenancy requirement via VXLAN encapsulation method [10]. It includes a separate control plane called DOVE Connectivity Server, so it resembles unicast VXLAN as described in chapter 1.2.3.

In summary, OpenDaylight provides a truly modular implementation of an SDN controller. In its first Hydrogen release OpenDaylight was offered in three editions (base, virtualisation, service provider), which emphasised the magnitude of its possible deployments.

ONOS

ONOS, SDN operating system founded by ON.Lab and released under Apache 2.0 license, is specifically built for carrier and service providers, thus focusing on HA, distributed scale-out architecture and performance [11, p. 1].

ONOS, similarly to OpenDaylight, provides its core services in a distributed fashion. ONOS may be deployed on a cluster of servers together forming a single platform. Hence, a failure of a single instance of ONOS controller should be transparent to the underlying networking device, which for the time being utilises another controller instance.

In terms of northbound API, ONOS supplies two levels of abstraction. The *Intent Framework* allows an application to request a service without knowing how the actual service will be provisioned. An intent such as connecting two hosts is translated into instructions compiled and downloaded to the underlying network devices. The other abstraction layer – *Global Network View* provides an insight into the operational status of the network [11, p. 5-6].

Southbound API enables ONOS to control underlying devices using multiple protocols, including OpenFlow 1.0, 1.3 and Netconf. ONOS is in southbound direction protocol-agnostic, as the details of protocol through which communication between the controller and the device takes place, are abstracted using adapter API.

The high-level architecture of ONOS is depicted in Figure 1.2. Since OpenDaylight provides similar abstraction layers as ONOS, the figure can be viewed as a general architecture of a robust, scalable SDN controller.

Since ONOS is not primarily focused on data center needs, it is hardly surprising that it does not support overlay networking protocols, specifically VXLAN. Instead, a suitable use case for ONOS as proposed by ON.Lab is *network functions virtualisation* (NFV), which can be characterised as decoupling specific hardware boxes from the software functionality. The purpose of NFV is to deploy services on generic x86 boxes within virtual machines. NFV differs from traditional virtualisation by connecting multiple VMs into a single building block, thus minimising management overhead. ONOS thanks to its northbound Intent Framework should be considered an ideal candidate for creating such blocks and mapping them to the underlying hardware [12, p. 5].

To sum up, ONOS in many respects resembles OpenDaylight (architecture, scale-out approach), but due to its recent public release it may lack some features, which OpenDaylight has managed to integrate within the framework.

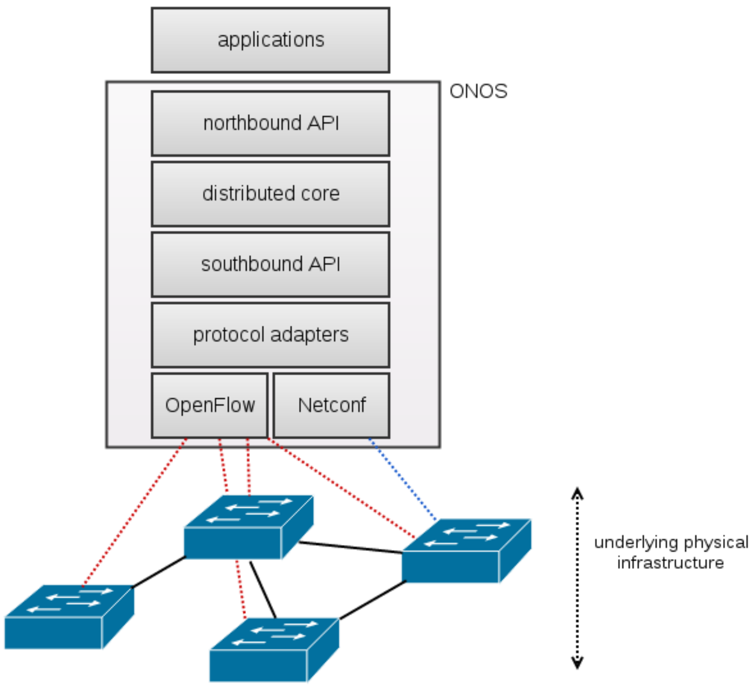


Figure 1.2: ONOS architecture

	Floodlight	**Open Daylight**	**Onos**
Current version	1.0	Helium-SR2 (2nd release)	1.0.1
Release date	December 2014	January 2015	January 2015
Implemented in	Java	Java	Java
Documentation	precise and concise	difficult to follow	satisfactory
Southbound API	OF 1.0, OF 1.3	OF 1.0, OF 1.3, XMPP, Netconf etc.	OF 1.0, OF 1.3, Netconf
Nortbound API	REST	OSGi, REST	REST
Primary use case	proof-of-concept, small environment	versatile (mostly data center)	service/carrier providers

Table 1.1: Comparison of open source SDN controllers (as of February 2015)

Summary of SDN controllers

Naturally, the list of SDN controllers as provided above is not exhaustive. There are other open source alternative solutions such as Ryu (Python-based), Maestro (Java-based) or Trema (C/Ruby-based) [13]. Not to mention the fact that virtually every major vendor has implemented an own proprietary controller such as VAN (HP), ProgrammableFlow (NEC), ONE (Cisco) and many others. Based on the lengthy list of controllers, SDN business has developed rapidly in the past few years, but no standard focusing on SDN controllers has yet been proposed and hence every vendor/community approaches the problem from a different perspective.

Table 1.1 compares and contrasts three analysed solutions. While Floodlight is a relatively light product in terms of available features, OpenDaylight and ONOS are more robust, scalable and thanks to their levels of abstraction more modular. All three products provide RESTful northbound API, support of OpenFlow up to version 1.3 in the southbound direction, but only OpenDaylight contains a truly scalable overlay networking module (VXLAN).

1.2 Overlay networking protocols

An *overlay network* can be characterised as a virtual network consisting of nodes and logical links built on top of an existing network [14]. VoIP can be considered as an example of overlay network, as it connects end-devices (VoIP phones and gateways) using a cheap, commonplace IP network (i.e. the Internet) as the *underlay* technology.

In terms of SDN, we will be using virtual layer 2 segments built on top of existing IP-based (layer 3) networks. This gives users the impression of being connected using a simple L2 switch, although the communicating devices may actually be several hops from each other. Overlay networks are built to provide a scalable alternative to regular VLANs. However, the transition from VLAN-based approach to overlay networks requires a large part of logic being moved towards the network edge, where hypervisors (and hence virtual switches) reside – subchapter 1.3 contains more information about this transition.

This subsection will start by exploring the drawbacks of traditional L2 switching represented by VLAN-based solutions. Later, three novel overlay protocols, namely multicast VXLAN, unicast VXLAN and VMware NSX will be briefly described.

1.2.1 L2 switching using VLANs

VLANs have been the most prevalent overlay technology for the past 15 years, but their scalability issues in large data centers act as an incentive to create more robust solutions.

Firstly, VLAN ID within an encapsulated Ethernet frame is limited to 12 bits, so no more than 4096 VLANs can be provided within a data center. Furthermore, due to the lack of TTL-like feature, L2 switching is reliant on STP, which may prevent from an effective link usage in exchange for a loop-less environment – particularly when devices are interconnected using multiple physical links.

Additionally, flooding of broadcast, unknown unicast and multicast traffic (abbreviated as *BUM*) is an inherent part of L2 switching. This effectively means that every single broadcast generated by a device within a VLAN must be processed by all other devices within the same VLAN. The same applies to multicast before the switch manages to build a table of subscribers for a given multicast address using IGMP snooping, for instance. Unknown unicasts are handled by ARP, which yet again results in querying every device within the VLAN. Thus, every L2 segment represents a single failure domain because a failure of one host (be it a virus or a malicious user) may impact other hosts within the same segment.

Last but not least, unlike L3, L2 does not provide any addressing hierarchy and thus switch must note down MAC addresses of all devices for which forwarding should be performed in order to prevent from unnecessary ARP queries.

All in all, VLANs do not scale adequately for large data center needs. Not only their number is limited, but number of hosts within a VLAN is finite as well. However, they represent a traditional and well-supported traffic isolation mechanism befitting most other environments.

1.2.2 Multicast VXLAN

VXLAN in general is an overlay encapsulation method, which wraps original Ethernet frame with VXLAN and UDP headers. VXLAN header defines a 24-bit VXLAN ID, which is used for traffic separation similarly to VLAN tags.

While the original IP and Ethernet headers give VMs the impression of being connected to a regular L2 switch, VXLAN and UDP headers initially provided by a VXLAN module in a virtual switch logically connect virtual machines residing on two or more hypervisors. All VXLAN traffic is subsequently encapsulated so that TCP/IP stack on the underlay is capable of transporting VXLAN frames between multiple hypervisors. A sample network consisting of two hypervisors connected by an overlay network is depicted in Figure 1.3.

In order to send a packet from VM 1 hosted by server A to remote VM 2 hosted by server B, several questions must be answered. These are [15]:

- How does VM 1 find out the MAC address of VM 2? In other words, how is ARP request propagated to other virtual machines within the same VXLAN subnet?

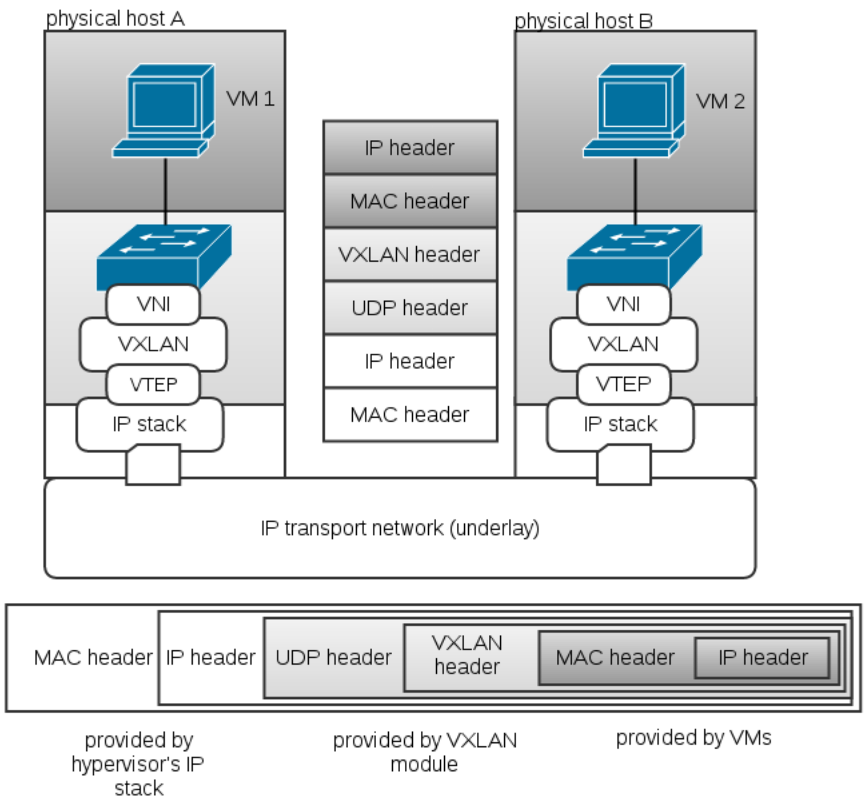


Figure 1.3: Sample overlay network using VXLAN

- How does hypervisor A determine the IP address of target hypervisor B to which encapsulated VXLAN frames should be sent? How do VM destination MAC addresses map to virtual tunnel endpoints (VTEPs)?

Multicast VXLAN as the name suggests uses IP multicast as a response to both of these questions. It maps a VXLAN segment to an IP multicast address. Instead of flooding BUM frames using L2, it performs flooding using L3. Upon the start of a VM, hypervisor sends an IGMP join to subscribe to a given multicast address. Hence, BUM traffic is sent to all hypervisors listening to a given multicast address. Using dynamic learning mechanisms, hypervisors gradually build their MAC-to-VTEP mappings to reduce further ARP queries.

Multicast VXLAN implies that IP multicast routing must be enabled throughout the underlay network. This requirement together with flooding of BUM traffic similarly to traditional bridged networks means that a more scalable solution will be required.

1.2.3 Unicast VXLAN

Unicast VXLAN unlike its multicast-based predecessor contains a control plane, which significantly reduces BUM flooding and no longer requires an IP multicast routing in the underlay – a major obstacle to multicast-based VXLAN deployment. However, unlike its competitor, which has been standardised since August 2014 [16], unicast VXLAN is a proprietary solution delivered by VMware and Cisco in new releases of Nexus 1000V distributed switch [17].

Nexus 1000V consists of two subcomponents – distributed forwarding plane within every hypervisor (*VEM*) and a centralised control plane (*VSM*). More information regarding Nexus 1000V will be provided in subchapter 1.3.3. Nevertheless, diagram 1.4 presents both above-mentioned subcomponents within a unicast-based overlay virtual network. In contrast to the generic VXLAN scheme in Figure 1.3, control plane is an inherent part of unicast VXLAN.

When a VM becomes active, local VEM reports this change to VSM, which in turn notifies other VEMs within the same VXLAN segment. If MAC address of the newly activated VM is propagated from VEM to VSM and subsequently to all other VEMs, unicast VXLAN operates in *MAC distribution mode* [17]. By reporting MAC addresses of respective VMs, VSM delivers a complete MAC-to-VTEP mappings, so unknown unicasts are no longer present within the network (i.e. no ARP requests). When it comes to multicast and broadcast traffic, local VEM is supposed to provide packet replication to every VEM with an active VM within the given VXLAN segment [18].

In summary, unicast VXLAN as implemented in Nexus 1000V provides a more scalable solution compared to multicast VXLAN mostly due to the fact that IP multicast routing is no longer a necessity and unknown unicast flooding is eradicated via the means of control plane. Thus, the only scalability limitation is the number of VEMs per VSM (128 as of February 2015 [17]). Even greater degree of scalability could be achieved via BGP-based communication between multiple instances of VSM (scale-out architecture) [19].

1.2.4 VMware NSX

VMware NSX or formerly known as Nicira NVP represents yet another overlay networking solution, which provides L2 forwarding. It is actually offered in two distinct variants – *NSX for multi-hypervisor environment* and *NSX for vSphere* [20]. While the former heavily relies on *Open vSwitch* (OVS) (later described in subchapter 1.3.4), NSX for vSphere utilises the functionality of virtual distributed switch (as tackled in section 1.3.2) already present within vSphere environment. NSX similarly to unicast VXLAN possesses a control plane. Throughout the rest of the chapter NSX for multiple hypervisors will be discussed and will simply be referred as NSX.

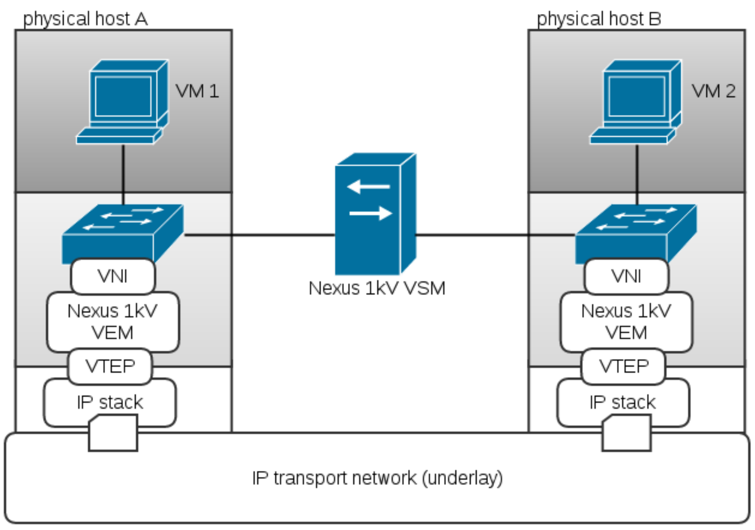


Figure 1.4: Unicast VXLAN using Nexus 1000V

Every hypervisor within NSX environment runs OVS with OpenFlow support. Every two hypervisor virtual switches sharing a common VXLAN segment are interconnected using a VXLAN tunnel among other possible encapsulation methods such as GRE or STT. Hence, a full mesh of tunnels results in the fact that every two hypervisors within the same VXLAN segment appear to be directly connected. Virtual switches also establish a connection to the NSX controller, which determines the behaviour of Open vSwitches via OpenFlow (flow control) and ovsdb-proto (tunnel configuration) protocols [21]. A simplified picture of NSX components is provided in Figure 1.5.

OpenFlow entries downloaded by the NSX controller into respective Open vSwitches bind VM destination MAC address with the tunnel endpoint. For instance, if VM 1 attempts to communicate with VM 2, NSX controller provides OVS within hypervisor A with a forwarding rule towards A-B tunnel. This corresponds to MAC-to-VTEP mappings defined in unicast-based VXLAN. Similarly, flooding of multicast and broadcast traffic may be handled by packet replication by the source hypervisor. Unlike unicast VXLAN in MAC distribution mode, NSX still performs unknown unicast flooding. For more complex environments NSX supports BUM flooding through a dedicated *service node*. In such a deployment, a single packet representing BUM traffic is sent by the source hypervisor towards the service node, which in turn sends replicated packets to all required OVS tunnel endpoints.

Besides L2 switching capability, NSX supports routing between multiple VXLAN segments (i.e. L3 forwarding). Routing may be performed by individual Open vSwitches which together form a distributed router or via a dedicated gateway node [22]. When a VM wants to communicate with an-

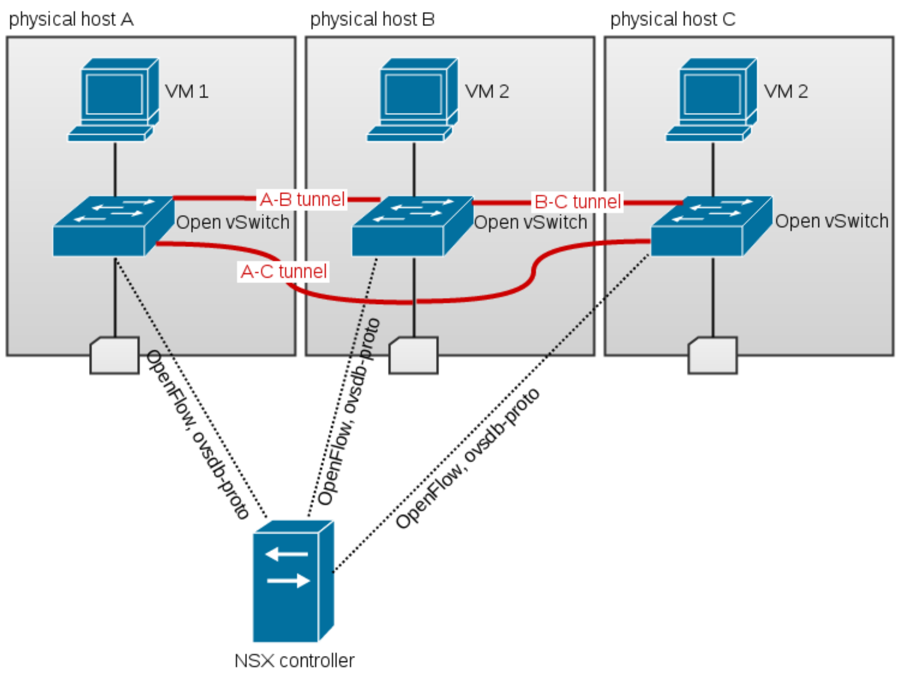


Figure 1.5: VMware NSX for multiple hypervisors architecture

other VM from a different subnet, it sends traffic to its default gateway, which is the local OVS. Local Open vSwitch then forwards the packet towards underlay IP address corresponding to the remote OVS. Remote OVS then delivers the packet to its directly connected VM and sends the response back to the originating OVS.

Figure 1.6 summarises both L2 and L3 traffc forwarding. While A-C tunnel is employed for intra-segment forwarding between virtual machines 1 and 3, hosts A and B are not connected via a tunnel, as they do not share a common VXLAN segment. Hence, for successful communication between virtual machines 1 and 2 local Open vSwitches acting as default gateways for directly connected VMs must know full topology information, including ARP entries and MAC-to-underlay-IP mappings for all tenant's VMs so that OVS within host A is capable of routing the packet towards OVS within host B and vice versa.

In summary, NSX provides a richer set of features in comparison to unicast VXLAN (distributed L3 routing, gateways to the external non-VXLAN world, service nodes), not to mention the fact that NSX can be employed in both open source OVS and proprietary vSphere environment alike.

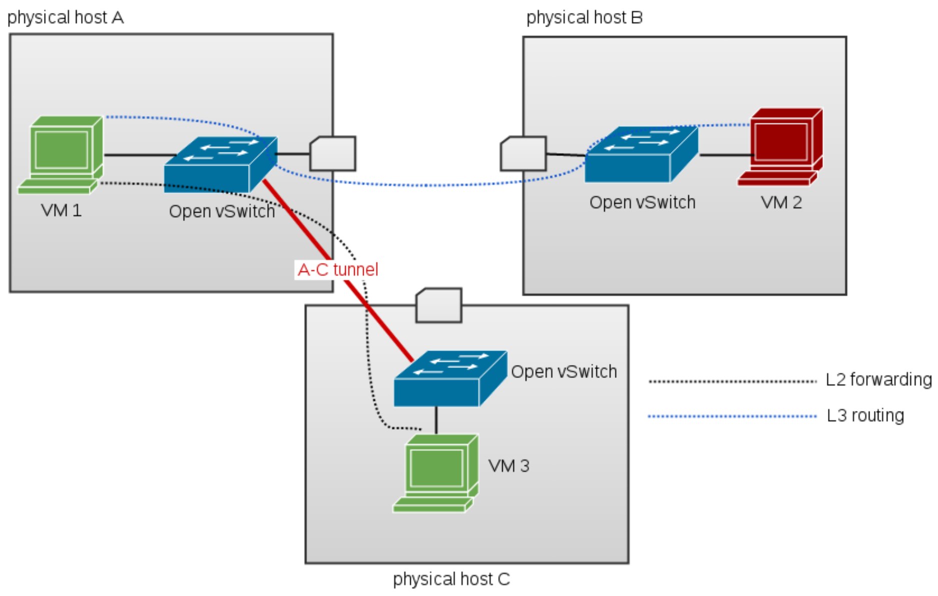


Figure 1.6: Comparison of L2 forwarding and L3 routing within NSX

	VLANs	**Multicast VXLAN**	**Unicast VXLAN**	**NSX**
Standard	IEEE 802.1Q	IETF RFC 7348	no (VMware + Cisco proprietary)	no (VMware proprietary)
Forwarding options	L2	L2	L2	L2 + L3
Control plane separation	none	none	VSM module (part of Nexus 1k V)	NSX controller
Scalability issues	limited number of VLANs; BUM flooding; STP; CAM size for core switches	multicast routing within underlay network; BUM flooding	maximum 128 VEMs per VSM, multiple VSMs (scale-out architecture) possible	distributed L3 forwarding (maintaining full topology information on a tenant basis)

Table 1.2: Comparison of overlay networking protocols

1.2.5 Overlay networking protocols summary

Table 1.2 compares and contrasts all of the above-mentioned solutions. While VLANs are by far the most widely spread technology for traffic separation, they are not suitable for large multi-tenant data center environment. Multicast VXLAN relies on IP multicast routing in the underlay network and suffers from BUM flooding. On the plus side, as a standardised solution it can work with VXLAN gateways of multiple vendors. Unicast VXLAN and NSX removes broadcast/multicast flooding thanks to a dedicated control plane. NSX for multiple hypervisors additionally provides L3 forwarding and northbound API, which can be consumed by an orchestrator system such as OpenStack [21].

1.3 Virtual switches

The purpose of a *virtual switch* is to allow virtual machines residing on a single physical host communicate with each other using the same set of protocols which would be used on a physical network without the need for additional networking hardware [23, p. 3]. Moreover, virtual switches act as a bridge between the physical and virtualised world, as they also provide virtual machines with connectivity to the external environment. In terms of SDN, we need to determine the extent to which virtual switches can be managed via an SDN controller, while providing support for a variety of networking protocols (including overlay protocols which themselves guarantee the multi-tenancy requirement). The schematic diagram of a virtual switch and its location is depicted in Figure 1.7.

In the following sections several virtual switches will be described. While most of them can be found in a set of proprietary products by VMware, Open vSwitch acts as an open source alternative.

1.3.1 vSwitch

Virtual switch (abbreviated as *vSwitch*) is the simplest virtual L2 switch provided in VMware products, notably in ESXi servers. Unlike physical switches, vSwitch does not learn MAC addresses associated with given ports – this piece of information is already provided by the hypervisor, which knows MAC address distribution to respective VMs and ports on vSwitch to which vNICs (virtual NICs; virtual Ethernet adapters) are attached. Similarly to physical switches, vSwitch maintains MAC-to-port mappings and performs lookup of each frame's destination MAC address [23, p. 4].

VM isolation is provided by means of VLANs similarly to regular physical networks. vSwitch implements the concept of *port groups*, which are associated with VLANs (usually in a 1:1 manner, although multiple port groups may share a single VLAN). A port group can be viewed as a configuration template common to all attached VM NICs. An example of configuration is the type

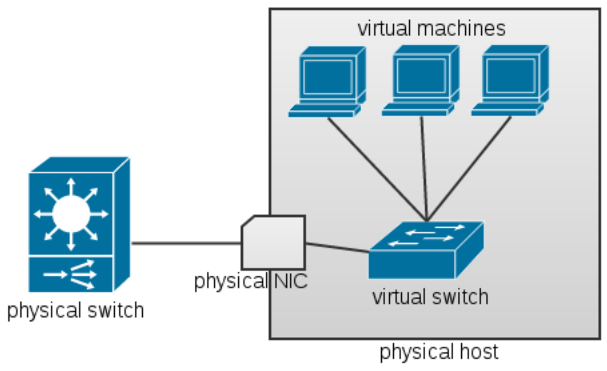


Figure 1.7: Virtual switch schematic diagram

of load balancing towards the external world - active/active or active/standby options allow administrators to specify which physical ports (called *uplinks* in VMware terminology) are to be used for a particular port group. As of vSphere 5, a single ESXi server may host 127 vSwitches [24]. Uplink is dedicated to a single vSwitch, whereas multiple port groups may share a common uplink.

The concept of VLANs results in a tighter coupling between the networking and virtualisation world. Not only a given VLAN must be included in the list of allowed VLANs on the trunk port from the physical switch towards the server, but the same VLAN must be propagated in the port group configuration within the ESXi host. The requirement of live migration (aka *vMotion* in VMware terms) between hypervisors effectively means that all VLANs must be propagated to all physical switches connected to the virtualised environment and to all switches in between (core and aggregation layers). This largely extends VLAN broadcast domain and hence the spread of broadcast traffic. Cisco would consider this to be in a breach of their best practices design [25, p. 60]. Thankfully, there are some techniques which eliminate excessive broadcast for VLANs which do not have any active ports configured within a switch – Cisco for this purpose implements VTP pruning.

The VLAN sprawl is displayed in Figure 1.8. Each color represents a separate VLAN. Connection between two switches from a different layer (core, distributed/aggregation, access/top of rack) is accomplished via a trunk link, which is capable of transporting multiple VLANs over a single physical cable. This architecture therefore enables virtual machines to be migrated from physical host A to physical host B and vice versa as long the same port group (and hence, VLAN) is configured in both vSwitches.

Another disadvantage of vSwitch is a lack of supported protocols. vSwitch does not support *Spanning Tree Protocol*, which is used to prevent loops at layer 2. More importantly, vSwitch does not provide *LACP* capabilities used for link aggregation. The fact that vSwitch does not support STP means that

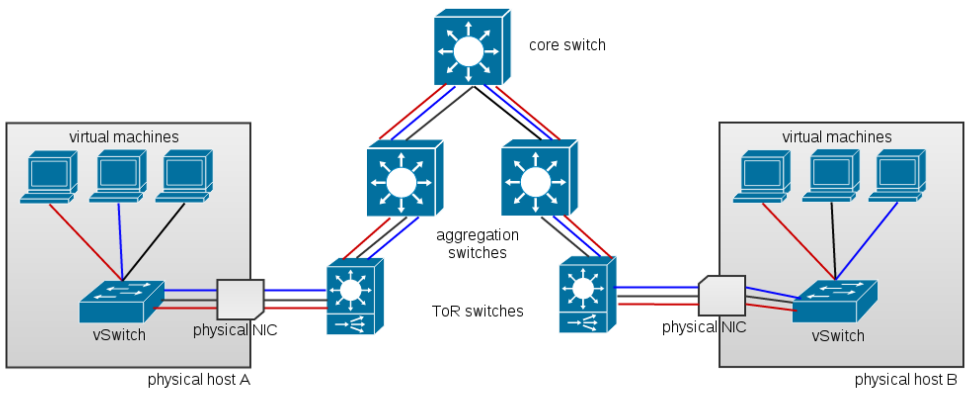


Figure 1.8: VLAN sprawl due to vMotion requirement

there must be other mechanisms used for loop prevention and failure detection, in particular when considering deployment with multiple uplinks.

Indirect link failure (a situation when, for instance, a link between aggregation and ToR switch fails) is impossible to be detected by vSwitch itself. Instead, *link state tracking*, which specifies two types of links – connections towards the core layer on the one hand and connections towards ESXi servers on the other hand – must be applied in the ToR switch. If a link towards the core fails, all links leading to downstream ESXi hosts are shut down as a result. Consequently, vSwitch detects direct link failure and uses other functional uplinks instead.

In terms of loop prevention, vSwitch implements several rules, which compensate for the lack of STP support, which would traditionally be responsible for detecting loops at L2. These are as follows [26]:

- Dropping incoming BPDUs – vSwitch ignores incoming BPDUs from physical ToR switches which results in all uplinks being active.
- Split horizon switching – frames received on one uplink are never forwarded to other uplinks.
- Verifying source MAC address of incoming frame using RPF check – if broadcast/multicast is generated by one of VMs running within the host, it is sent out using a single uplink. The same frame, however, arrives on other uplinks as well. By checking source MAC address using reverse path forwarding, vSwitch determines whether the frame comes from a VM hosted by the server itself. In such a case, the frame is silently dropped. This behaviour is depicted in Figure 1.9. The red arrows show the direction of broadcast traffic as it is generated by a VM, sent out of one uplink, transported throughout the entire broadcast domain and finally received and consequently dropped by another uplink.

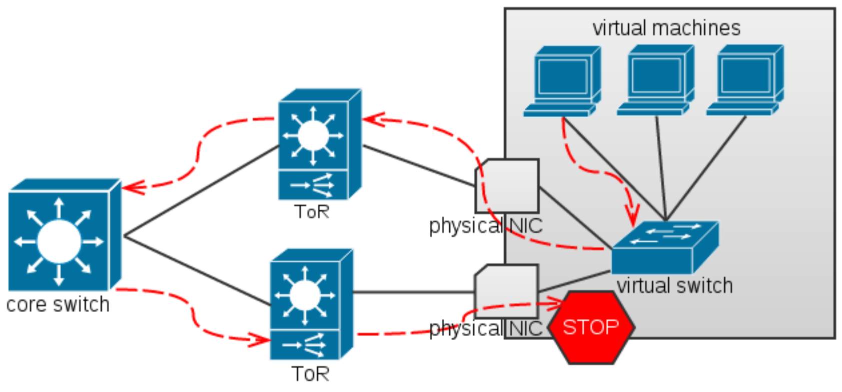


Figure 1.9: vSwitch loop prevention due to broadcast generated by a VM

The fact that vSwitch ignores 802.3ad (LACP) means that the only viable manner in which a ToR switch and vSwitch can be interconnected using multiple links at the same time involves a statically configured port channel. This tends to result in a lack of external connectivity for some VMs if configured improperly. By default, vSwitch pins VM NIC to a specific uplink and if traffic for the given VM NIC is received on a different uplink than expected, it is silently discarded. The countermeasure is to assign traffic to an uplink based on a hash function of (src_IP, dst_IP) on the vSwitch and physical ToR switch alike [27]. Moreover, static port channelling is unreliable even if configured properly because unlike LACP, which performs negotiation between the two sides before establishing the connection and throughout its duration, static port channel does not perform any validation whatsoever and may result in a bridging loop [28].

In summary, vSwitch represents a very basic VLAN-capable switch, which suffers from a lack of support of major networking protocols, notably LACP. Additionally, it induces a close cooperation between the networking and virtualisation team. Last but not least, vSwitch represents an independent entity with both control and data planes hidden within the hypervisor.

1.3.2 vDS

VNetwork Distributed Switch (*vDS*) overcomes the requirement of having a separate vSwitch for every ESXi host and instead provides a separation of control and data planes. While the data plane functionality (i.e. frame forwarding) remains hidden within the hypervisor, the control plane is now centralised and resides in vCenter Server. The schematic diagram is attached as Figure 1.10. vDS is conceptually similar to stackable switches, which represent a single entity from the management perspective, but every switch within a stack still performs the forwarding itself. vDS thus brings several advantages

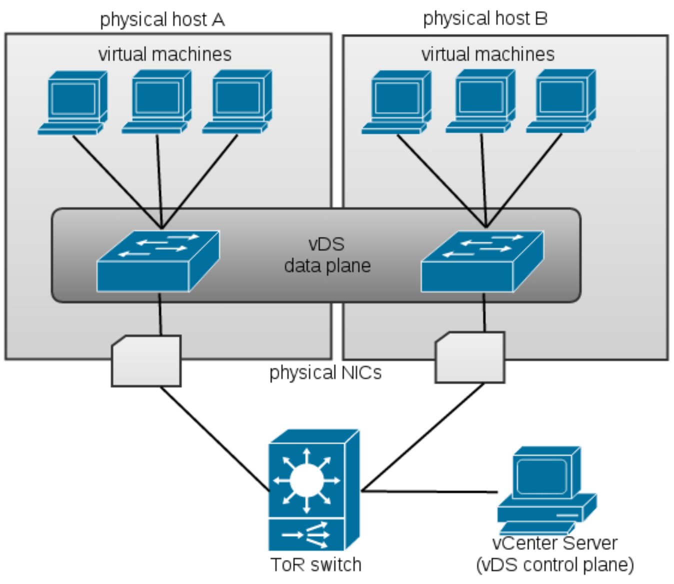


Figure 1.10: vDS schematic diagram

over original vSwitch, namely:

- increased flexibility due to simplified configuration in large-scale environment
- port group configuration consistency – vDS makes sure that port groups configured on physical hosts are identical, thus allowing vMotion
- LACP support since vSphere 5.1 (currently available version – 5.5 as of January 2015) [29]

All in all, vDS offers the same set of functionality as vSwitch plus it provides simplified configuration and LACP support. However, vDS still does solve the problem of limited functionality compared to enteprise physical switches.

1.3.3 Cisco Nexus 1000V

Nexus 1000V represents an implementation of pluggable distributed virtual switch architecture delivered by Cisco and VMware. Nexus 1000V is a replacement of vDS described in the previous chapter 1.3.2. In fact, Nexus 1000V heavily uses vDS-provided API and therefore its internal architecture resembles vDS. It consists of two major parts, namely *VEM* and *VSM*. VEM (Virtual Ethernet module) represents the virtual switch running within the

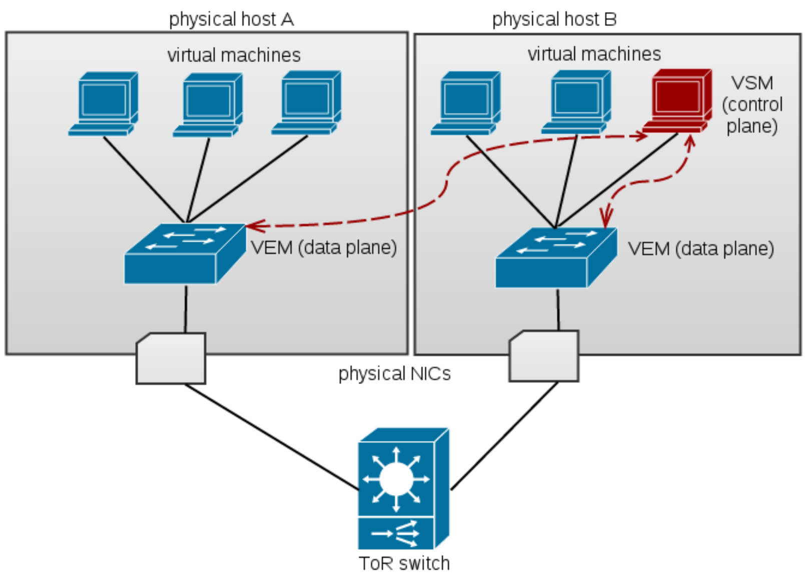


Figure 1.11: Nexus 1000V schematic diagram

hypervisor, whereas VSM (virtual supervisor module) acts a central element in terms of management, configuration and logic. Figure 1.11 displays the location of both components within vSphere environment. The communication depicted between the highlighted components will be described later.

VEM basically downloads configuration from VSM and forwards traffic at L2 accordingly. Unlike above-mentioned virtual switches, VEM has always provided a larger set of features, including but not limited to port security, L2/L3 access control lists or private VLANs [30]. In later releases additional security features such as IP source guard, DHCP snooping or dynamic ARP inspection have been added making VEM and hence the entire Nexus 1000V almost feature-wise comparable to enterprise-level physical switches [31].

VSM is conceptually similar to a supervisor module in high-end physical switches such as Cisco Catalyst 6500. Modular supervisor in a chassis-based switch acts a processor (i.e. central piece of logic) to the entire switch. VSM operates as a virtual machine within a cluster of ESXi hosts. As it represents a central point of configuration, which is then propagated to respective VEMs, running two VSMs in an active-standby mode is available and recommended as part of HA design.

For the the purpose of communication between VSM(s) and VEMs, two additional VLANs are formed. *Packet VLAN* carries CDP/LLDP/IGMP packets received by a VEM and propagated to the VSM for further processing. Similarly, VSM sends periodic CDP/LLDP packets to VEMs, which in turn forward them to physical uplinks. *Control VLAN* is used to download configuration from VSM to individual VEMs [32].

Nexus 1000V provides multi-tenancy not only via VLANs similarly to vSwitch/vDS, but also via unicast VXLAN as described in chapter 1.2.3. Additionally, Nexus 1000V is capable of bridging the world of VLANs and VXLANs by running a VXLAN gateway as a virtual machine.

To sum up, Cisco Nexus 1000V provides the most advanced virtual switch in the VMware world. Range of supported protocols (including unicast VXLAN), well-known IOS command line interface and prevention from a single point of failure make Nexus 1000V an ideal virtual switch in a large data center.

1.3.4 Open vSwitch

Open vSwitch (OVS) represents an open source implementation of a virtual switch. Unlike vDS or Cisco Nexus 1000V, OVS is not a distributed switch that would be able to manage multiple virtual switches running within different hypervisors. Hence, in terms of manageability, OVS rather resembles a simple vSwitch.

What distinguishes OVS from a regular Linux bridge (and from other virtual switches as well) is the concept of *flows*. Unlike regular switches, which make forwarding decisions based on destination MAC address only, OVS allows for using multiple parameters at the same time. These may include L2 to L4 options such as source/destination MAC/IP addresses, source/destination TCP/UDP ports, VLAN tags, DSCP marks etc. This is where SDN comes into place by allowing OpenFlow protocol to set up forwarding entries within the virtual switch based on the requirements of a central authority, the SDN controller. OVS is considered a de-facto Openflow (virtual) switch reference implementation [33].

OVS consists of three major parts, two of which operate in user space (vSwitch daemon also known as *ovs-vswitchd* and database server labelled *OVSDB server*). The last component performs the actual packet forwarding within kernel space. When the first packet arrives to OVS, the kernel module determines whether it has any rules associated with the packet. If not, the kernel module asynchronously sends the packet to user-space ovs-vswitchd, which determines action to be taken (drop, forward, flood etc.). User-space daemon notifies the kernel module and sends the packet to it. The kernel module is then able to perform the actual forwarding. Subsequent packets within a flow are forwarded directly by kernel without querying ovs-vswitchd [34].

Figure 1.12 displays all three components comprising OVS as well as packet handling as described in the previous paragraph. SDN controllers as defined in chapter 1.1 may represent an external flow controller by providing a set of permit/deny rules for inter-VM communication. The creation of switch interfaces/ports/QoS settings can be performed via built-in *ovs-vsctl* tools, a

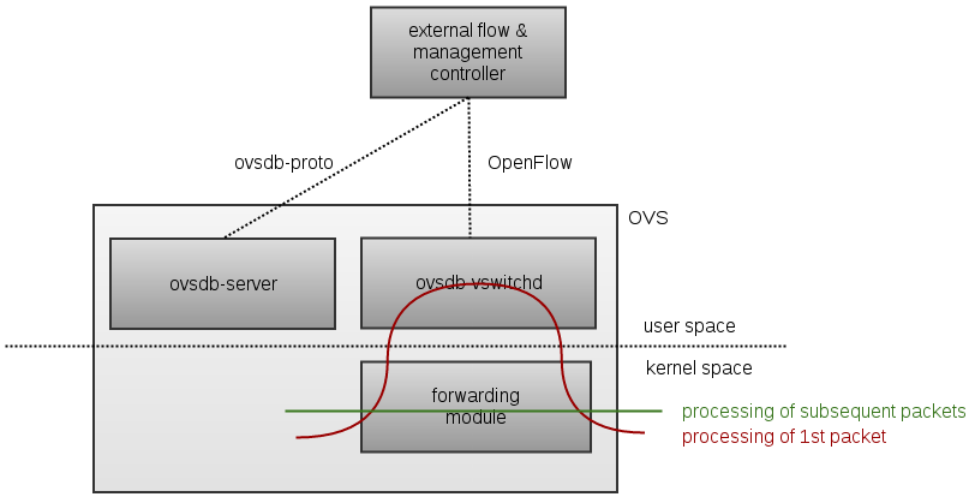


Figure 1.12: Open vSwitch architecture and packet flow diagram

high-level API working above OVSDB. The database then acts as a persistent storage for OVS configuration.

OVS scalability used to be limited prior to 1.11 release due to the *microflow* concept. OVS would require exact flow match in order to forward a packet, otherwise the packet would be punted to user-space ovs-vswitch daemon. Since release 1.11, OVS has supported *megaflows*, which no longer require an exact match and instead, provide the kernel forwarding module with an option to perform wildcarding [35].

In terms of supported protocols, OVS lags behind state-of-the-art Nexus 1000V, but at the same time it provides support for STP, LACP and even unicast-based VXLAN. NSX for multiple hypervisors heavily depends on OVS, as it creates tunnels between every two Open vSwitches within hypervisor hosts - for more information see chapter 1.2.4.

In summary, OVS is able to compete with proprietary, mostly VMware-based virtual switches. Although it lacks the distributed nature when compared to vDS or Nexus 1000V, its OpenFlow capabilities mean that the control plane functionality can easily be ”outsourced” by means of the external controller.

1.3.5 Virtual switches summary

Table 1.3 concludes the discussion related to virtual switches by comparing each solution in terms of supported features and deployment model. Based on the findings, it is obvious that Cisco Nexus 1000V and Open vSwitch are the most advanced virtual switches. While Nexus 1000V provides well-known Cisco IOS CLI for managing distributed virtual switch, its deployment

	vSwitch	**vDS**	**Nexus 1000V**	**Open vSwitch**
Vendor	VMware	VMware	Cisco + VMware	open source community
Deployment	per host	per vCenter	per vCenter	per host
Separation of control and data planes	none	data plane within every ESXi host, centralised control plane within vCenter Server	data plane (VEM) within every ESXi host, control plane (VSM) centralised within a VM	data plane within every hypervisor, control plane possibly ”outsourced” by the SDN controller
STP support	no	no	no	yes (802.1D)
LACP support	no	yes	yes	yes
OpenFlow support	no	no	no	yes
Multi-tenancy	via VLANs	via VLANs	via VLANs and unicast VXLANs	via VLANS and unicast VXLANs

Table 1.3: Comparison of virtual switches

is limited to the highest vSphere edition called Enterprise Plus [36] [37]. On the contrary, Open vSwitch is supplied free of charge, but for more a complex deployment it must be combined with an external controller, otherwise it behaves as a more sophisticated Linux bridge.

1.4 Physical switches

The state-of-the-art chapter will be finished by a brief discussion of physical switches and their problems with regard to SDN adoption. Virtual switches, which are basically software constructs often representing a plugin or a loadable module above the existing hypervisor, may be fairly easily changed. However, physical switches may have been operating for years within a data center and the entire SDN adoption in such a case depends on the respective vendor – whether OpenFlow-enabled firmware will be released or not.

Besides the actual support of OpenFlow or other SDN-related protocols, scalability of the physical switch should be scrutinised as well. It has already been implied that upon receipt of a packet for which a switch does not have any associated flow, the switch may either drop the packet or more likely send the packet for analysis to the external controller. The actual processing or punting is realised via CPU and may negatively impact overall performance.

Another factor which could potentially impede switch performance is the number of flows the switch is able to install per second. In a dynamic environment the SDN controller may create thousands of flows per second, but the switch may limit the number of FIB updates to several hundreds - at least, HP Procurve 5406zl switch is capable of installing roughly 275 new flows per second [38].

The last, yet the most obvious scalability issue when installing new flows into the physical switch is the limited size of TCAM. While the switch may hold tens of thousands destination MAC addresses used for classical L2 forwarding, the same TCAM table will hold few thousands of flows when considering the 12-tuple based on which forwarding decisions are made in the OpenFlow world [39].

All in all, SDN adoption should not be restricted to the virtualised world only. On the contrary, there are several issues specific to physical switches which are worth contemplating during (data center) design phase.

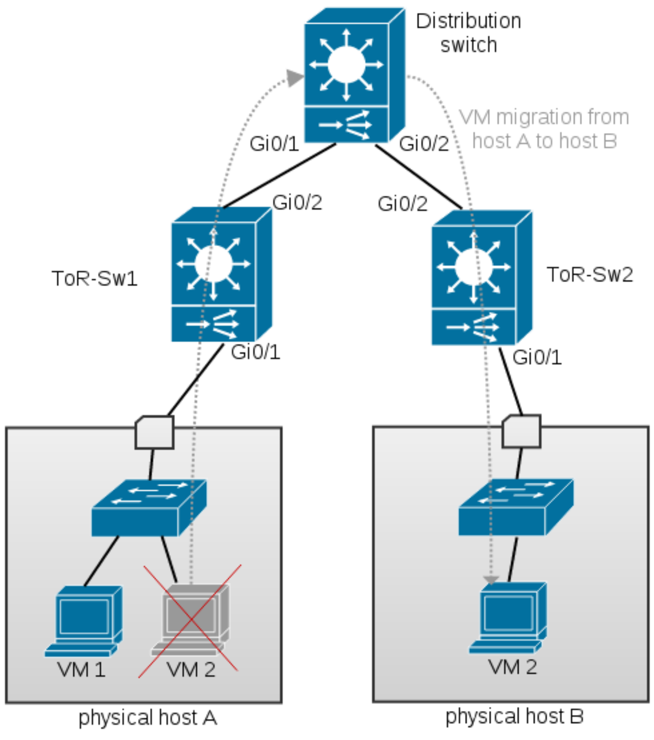
Chapter 2

Analysis and design

The purpose of the following chapter is to conduct an analysis of an application, which will be implemented as part of the thesis. The application will attempt to enhance the availability of services using the principles of SDN as discussed in chapter 1. Specifically, the application will focus on reducing the downtime during a *live VM migration* between multiple physical hosts. The actual principles and techniques of VM migration (with the exception of networking) shall not be discussed in the thesis, as they have been well described elsewhere, for instance in [40].

The downtime during live VM migration stems from the fact that upstream switches continue forwarding traffic destined to the VM in question towards the original hypervisor, while the VM has already been migrated to a different host. Therefore, upstream switches have to be notified of a change in VM location so that they can alter entries in their CAM tables. A sample topology highlighting this problem is depicted in Figure 2.1. In this case, VM 2 is being migrated from physical host A to host B. Consequently, CAM entries for VM 2 MAC address have to be updated throughout the entire broadcast domain (consisting of 3 switches) right after VM 2 has been migrated.

Security concerns should also be taken into consideration when discussing VM migration. The switchport to which VM is connected may impose restrictions on a list of source MAC addresses, thus preventing from MAC address spoofing (in Cisco terminology this is referred to as *port security*). Such rules must be reapplied at the destination switch – the switch to which the VM will be connected after the migration. This is difficult to automatically achieve using traditional networking concepts and therefore, port security is often neglected in flexible data center environment, where vMotion and other types of VM migration are commonplace.



	Original destination for VM2 MAC address	New destination for VM2 MAC address
Distribution switch	Gi0/1	Gi0/2
ToR-Sw1	Gi0/1	Gi0/2
ToR-Sw2	Gi0/2	Gi0/1

Figure 2.1: Changes in CAM tables due to live VM migration within a single tenant environment

2.1 Requirements

This section contains a list of requirements imposed on the application. These are:

1. decrease in service downtime period during live VM migration
2. cooperation between the orchestrator system and the SDN controller
3. savings on network bandwidth and CPU processing by minimising flooding of unknown unicasts (from the perspective of switches)
4. adoption of OpenFlow protocol to manage forwarding rules
5. multi-tenancy via the means of VLANs

The application should ideally perform faster (i.e. diminished service downtime) when compared to traditional live VM approaches (see the following section 2.2). The idea behind this assumption is the fact that the application will communicate with the orchestrator, which already possesses the latest information regarding the status and transition of hosted VMs during the migration. Additionally, these pieces of information will be propagated to the SDN controller and appropriate forwarding rules recreated via the means of its north-bound RESTful API. By directly injecting flow entries into the switching topology savings in terms of network bandwidth consumption and CPU processing will be achieved because switches will no longer rely on the traditional MAC learning mechanism, which results in flooding unknown unicast frames. Furthermore, OpenFlow will be be utilised for propagation of flow rules. Last but not least, multi-tenancy requirement will be handled via traditional VLANs, which should be sufficient for a small data center environment.

2.2 Live VM migration discussion

There are multiple approaches to the problem of live VM migration which vary in the degree of involvement of the virtual machine itself or the hypervisor. Ideally, since VM live migration should be transparent to the VM itself, hypervisor in charge should notify upstream switches of any changes.

The most simple method is based on MAC address table aging and does not induce any additional overhead. It is assumed that an entry in the switch CAM, which associates a port number, VM MAC address and VLAN ID, will age out after a period of inactivity (5 minutes in Cisco Catalyst series switches [41]). Similarly, a CAM table entry corresponding to a MAC address will be flushed if the same MAC address is detected on a different port. In other words, the most recent arrival port is always associated with a given

MAC address within the switch CAM table – at least in the Cisco world, but the same concept is expected to be utilised universally [41]. In both of these situations, CAM entries are updated *reactively* – when the newly migrated VM attempts to communicate. Hence, this approach is imperfect, as it potentially creates a blackhole because upstream switches are unaware of any changes and will send the traffic to the original host until the VM starts communicating from a different host. UDP traffic is in this respect more susceptible compared to TCP, since UDP does not acknowledge successful packet receipt.

The *proactive* approach usually relies on sending *gratuitous ARP* either by the destination hypervisor or the VM itself. Gratuitous ARP is used to announce the ownership of an IP address on a network segment, so it is obvious that such a solution is limited to a single LAN. However, this is not a scalability issue, as the VM in question is supposed to retain both its IP and MAC addresses when undergoing live migration. Gratuitous ARP sent by the hypervisor does not require additional action to be taken by the VM itself, which is considered advantageous. On the other hand, hypervisor actually forges VM identity when sending ARP packets on its behalf. Gratuitous ARP sent by the VM itself to some extent contradicts the transparency requirement (VM should be completely unaware of its migration). This approach is employed by open source Xen hypervisor [42], whereas OpenNebula orchestrator system (and the underlying KVM hypervisor) utilises gratuitous ARP sent by the physical host [43].

VMware uses a different approach and instead of gratuitous ARP ESXi hosts send *reverse ARP* (RARP) packets on behalf of the VM undergoing live migration. Since the protocol is now considered obsolete, the broadcast message sent by the hypervisor is ignored by hosts within the broadcast domain with the exception of switches, which update their CAM entries as a result of receipt of such packets [44]. VMware states that the service downtime due to VM live migration takes up to 2 seconds on a Gigabit Ethernet network in their vSphere product set [45].

2.3 Application design

The following subsection includes several UML diagrams. Figure 2.2 shows a simplistic VM lifecycle within an orchestrator system. It does not take into account actions such as VM suspending, its subsequent resuming and deleting, to name a few. Instead, the gray triangle focuses on live migration only. VM live migration may succeed, resulting in VM being hosted by a different hypervisor. On the other hand, due to a lack of computing resources within the potentially new hypervisor or networking issues, live migration may fail and the application must be capable of handling both cases.

Figure 2.3 represents a use case diagram with a list of actions the application should be able to perform. Firstly, the application will need to obtain information from the orchestrator system regarding VM live migration – once it has been initiated and once it has been completed along with the migration status.

In case of a failed migration, for instance due to a lack of resources on the destination host, it is assumed that the orchestrator system will continue hosting the VM on the original hypervisor. However, this behaviour is platform-dependent. In this case, VM location and corresponding flow entries will not be modified by the SDN controller or the application.

If a migration is to succeed, the application will find all relevant flow entries by querying the SDN controller. Since the currently active sessions have to persist once the migration has been completed, the application in conjunction with the SDN controller will change ingress/egress ports in the flow entries as highlighted in Figure 2.1. The new port numbers will be defined by the SDN controller based on recalculation of a shortest path algorithm between the two talkers – this would include the new location of the VM. Generally, when redefining the relevant flow entries, we assume that the VM migration has already succeeded – albeit at the time of the flow recalculation, the migration result is unknown. Once the application has been informed of the migration success, it will then ask the SDN controller to install the above-mentioned flows.

The logic described in the previous paragraph is portrayed in Figure 2.4 – sequence diagram of communication among the orchestrator system, the application and the SDN controller.

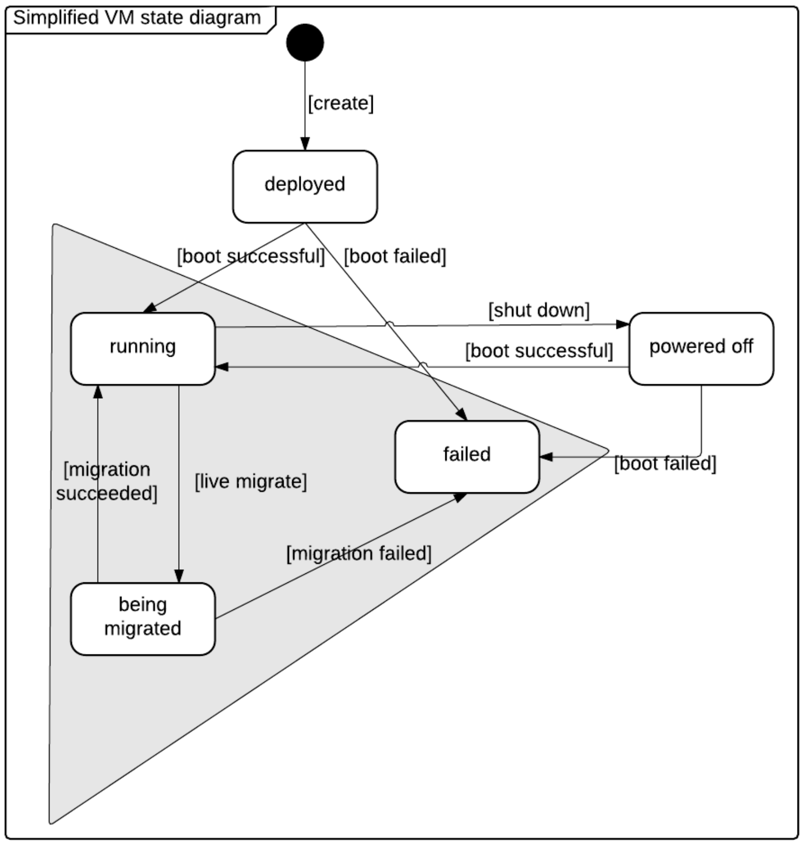


Figure 2.2: VM state diagram

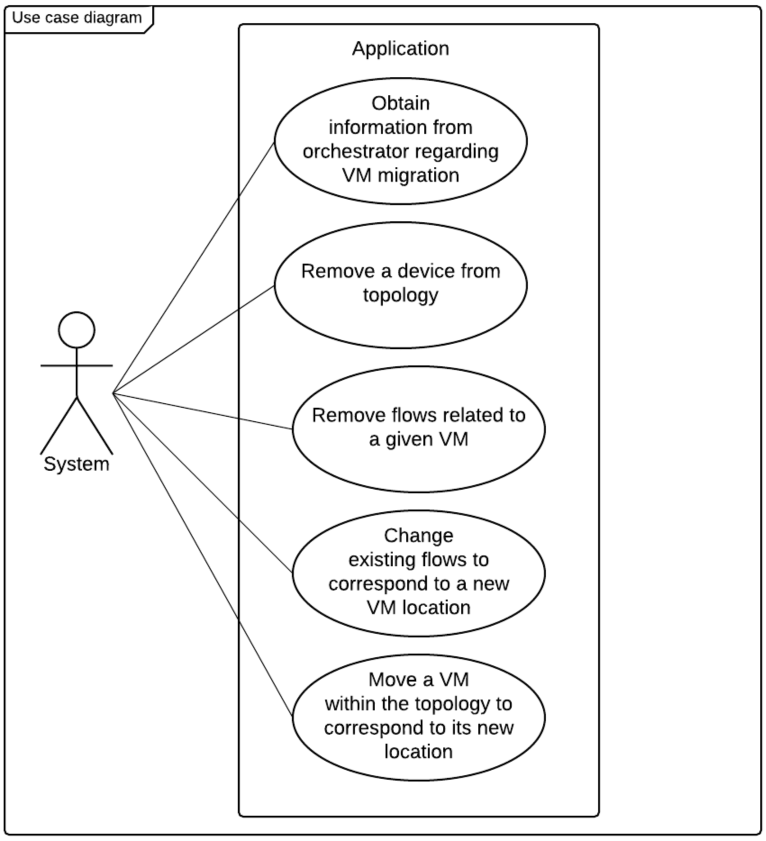

Use case diagram
Application
Obtain information from orchestrator regarding VM migration
Remove a device from topology
Remove flows related to a given VM
Change existing flows to correspond to a new VM location
Move a VM within the topology to correspond to its new location
System

Figure 2.3: Use case diagram

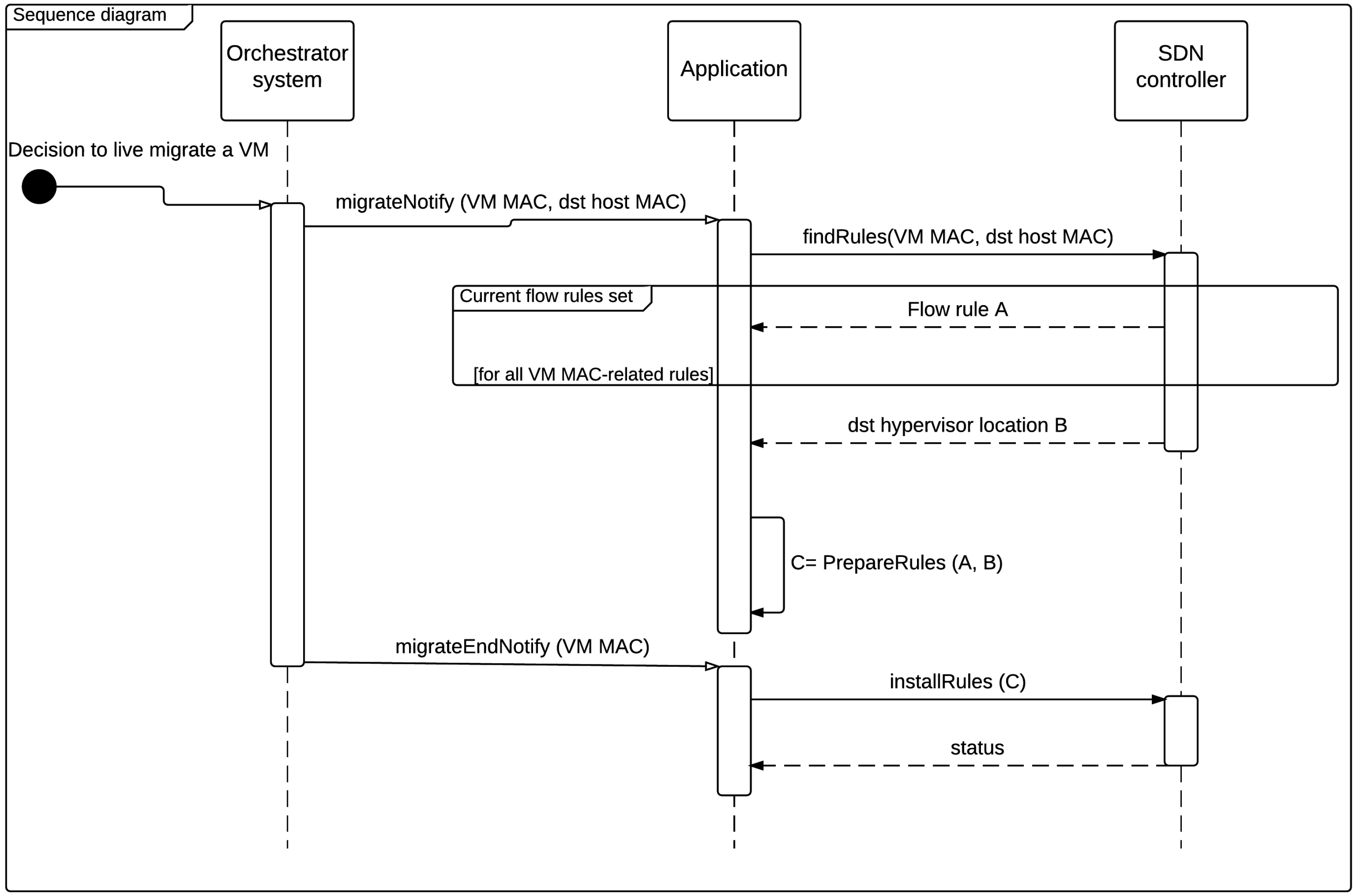

Sequence diagram
Orchestrator system
Application
SDN controller
Decision to live migrate a VM
migrateNotify (VM MAC, dst host MAC)
findRules(VM MAC, dst host MAC)
Current flow rules set
Flow rule A
[for all VM MAC-related rules]
dst hypervisor location B
C= PrepareRules (A, B)
migrateEndNotify (VM MAC)
installRules (C)
status

Figure 2.4: Sequence diagram

2.4 Technology choice

This subsection discusses the choice of appropriate technologies, which will be used to implement the application. Based on the sequence diagram depicted in Figure 2.4, both the SDN controller and the orchestrator system will be designated. Additionally, the choice of SDN controller also significantly affects the range of possible virtual switches.

Since the project should utilise open source tools, proprietary VMware products are thus disregarded. In terms of virtual switches as described in section 1.3, Open vSwitch remains the only viable option because the remaining products – be it a simple vSwitch, distributed VDS or Cisco Nexus 1000V are employed solely in VMware environment. Open vSwitch, on the other hand, supports OpenFlow, which complies with the list of requirements as stated in 2.1.

The discussion of orchestrator systems has not been part of this thesis. Instead, a comparison of private IaaS cloud systems has been performed as part of my bachelor's thesis [46]. Based on the two-year-old research, OpenNebula has been selected as the orchestrator platform due to a couple of reasons. Firstly, OpenNebula is capable of performing so called contextualisation – ability to programatically change the behaviour of virtual machines, while providing a fairly simple installation and subsequent configuration procedure. Secondly, OpenNebula is a suitable orchestrator system for a small or medium-sized data center, which fits the overall picture as presented in the introductory section.

In terms of SDN controllers, OpenDaylight and ONOS have been thoroughly contemplated, as neither of them represents a SPOF and thus they may potentially be employed in production environment. In the end, ONOS has been chosen mostly thanks to its Intent Framework, which allows the programmer to concentrate on his task without the need for calculating the shortest path within the underlying topology – this is instead performed by ONOS and its Intent Framework. On the other hand, ONOS is a novel product and as such does not support some features like VXLAN and therefore, traditional VLANs will be used as L2 over L3 overlay mechanism for separating respective tenants.

The actual application will be implemented in Java programming language due to the author's previous experience and the fact that ONOS is written in Java as well.

Chapter 3

Implementation

3.1 Inter-component communication

This subchapter attempts to describe the actual implementation of the communication between all entities as displayed in Figure 2.4. Based on the sequence diagram the *control application* needs to obtain data in terms of VM location from OpenNebula, perform some computation and propagate the results to ONOS SDN controller, which updates resultant flow entries. Since some features which will be discussed in the following section are version-specific, Table 3.1 shows versions of components on top of which the entire implementation is built.

3.1.1 Communication with OpenNebula

OpenNebula supports XML-RPC, which provides programmers with the ability to query the state of virtual machine being migrated (hostname, source hypervisor, IP address, MAC address, VLAN etc.)[47]. However, when it comes to the migration process, OpenNebula itself does not have sufficient amount of information regarding the destination hypervisor. The actual request to perform live migration for a given VM is internally transformed into a single command as listed in Code snippet 3.1. The code comes from */var/lib/one/remotes/vmm/kvm/migrate* bash script, which is part of OpenNebula's master node.

Type of SW	**Name**	**Version**
Orchestrator system	OpenNebula	4.12
Hypervisor management	Libivrt	1.2.9-9
SDN controller	ONOS	1.0.1

Table 3.1: Versions of SW components

This suggests that OpenNebula is heavily dependent on the underlying hypervisor (KVM in this case), which hosts virtual machines and performs VM migration requests on behalf of the orchestrator system. Therefore, in order to obtain all the required information regarding the source and destination hypervisors in time, libvirt is used as a source of the information. Libvirt itself is not a hypervisor – instead, it provides a unified management and API for all major hypervisors (KVM, Xen, ESXi).

```
virsh --connect qemu:///system migrate --live $VM_ID qemu+ssh://
    $DESTINATION/system
```

Code snippet 3.1: VM migration request transformed into libvirt call

Unlike OpenNebula, which performs regular polling to determine the state of its virtual machines, libvirt uses the concept of *hooks*, which are custom scripts triggered by specific system events [48]. These events according to the official website occur when "the libvirt daemon starts, stops, or reloads its configuration" or "a QEMU guest is started or stopped" [49]. The latter use case will be utilised to send asynchronous messages to the control application.

Due to the asynchronous nature of hooks in general, it makes sense to implement the hook as a simple script, which is able to respond to all required events by sending messages to the control application. For this purpose, bash was designated as the scripting language of the *qemu* hook. The bash script asynchronously contacts the control application using network sockets. Since the qemu hook is triggered on multiple occasions during VM migration, the control application receives information regarding VM location gradually. By sending messages asynchronously, the qemu hook remains stateless and instead, the entire state is maintained within the control application.

The standard input of the qemu hook is represented by the libvirt XML configuration file (which corresponds to the output of *virsh dumpxml* command) so the bash script is mostly in charge of parsing XML. External program named *xmlstarlet* is used to perform simple XPath query to obtain the appropriate pieces of information (VM hostname, MAC address, bridge name, bridge port).

Not only is the qemu hook invoked repeatedly during a single VM migration, but it is triggered by source and destination hypervisors alike. The order of the messages along with their timestamps is displayed in the following Log snippet 3.2:

```
1 src hypervisor                       dst hypervisor
2                                     [10:27:42.224] one-16 migrate begin
3                                     [10:27:42.253] one-16 prepare begin
4                                     [10:27:42.472] one-16 start begin
5                                     [10:27:42.890] one-16 started begin
6                     < actual VM migration >
7 [10:27:51.847] one-16 stopped end
8 [10:27:51.882] one-16 release end
```

Log snippet 3.2: Qemu hooks generated during a sample migration

The qemu hook informs the control application in steps corresponding to line numbers 2 and 5 on the destination side and 7 on the source side. *Migrate* message (line no. 2) is invoked at the very beginning of the migration when resources are yet to be allocated on the destination side. Therefore, the control application awaits another message from the destination hypervisor corresponding to line number 5.

The *started* message informs the control application of the actual resource reservation. This basically means that at this point the destination hypervisor has already allocated a bridge port to which the VM will be connected after the migration. The bridge port thus acts as the VM destination location.

The *stopped* message corresponding to line number 7 occurs after the actual VM migration and represents the turning point after which the source hypervisor no longer hosts the VM.

All three above-described message types are handled by a single qemu hook script, which upon the message receipt from libvirt and subsequent parsing contacts the control application.

Libvirt, however, does not possess all the required pieces of information. Specifically, when it comes to networking, libvirt is L3-unaware and hence does not know VM IP address and VLAN tag (if used). Among others, these are required to fully specify host location in ONOS. Therefore, the qemu hook also queries OpenNebula using built-in XML-RPC. The actual XML-RPC query is constructed using *curl*. IP address and VLAN tag are appended to other pieces of information in the *started* message.

The control application and OpenNebula XML-RPC endpoints are not hard coded within the qemu hook. Instead, a configuration file *controller.conf* contains all the required data, including credentials of a user with sufficient privileges to query OpenNebula. Note that storing user credentials in an appropriately secured plain text file is the default method employed by OpenNebula. A sample configuration file is listed in Configuration snippet 3.3.

```
DESTINATION_HOST=PRGNEB001
DESTINATION_PORT=9001
RPC_ENDPOINT=http://PRGNEB001:2633/RPC2
RPC_CREDENTIALS=oneadmin:DaFrojli3wif
```

Configuration snippet 3.3: Sample configuration file for the qemu hook

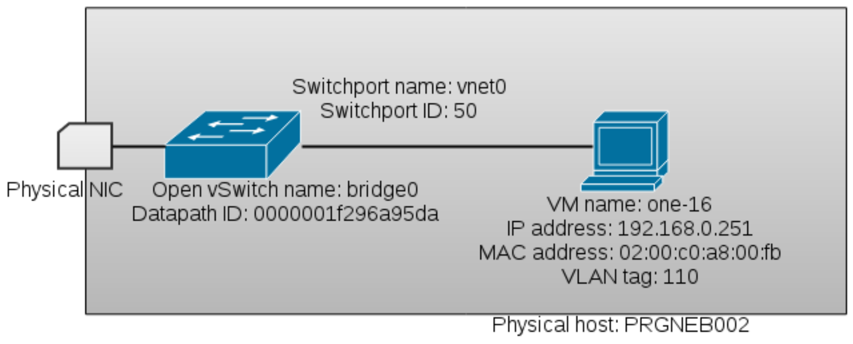

Switchport name: vnet0
Switchport ID: 50
Physical NIC
Open vSwitch name: bridge0
Datapath ID: 0000001f296a95da
VM name: one-16
IP address: 192.168.0.251
MAC address: 02:00:c0:a8:00:fb
VLAN tag: 110
Physical host: PRGNEB002

Figure 3.1: Visual representation of data passed in *started* message

Both the *qemu* hook and configuration file *controller.conf* are supposed to be located in */etc/libvirt/hooks* folder. Configuration file should be secured by 600 umask, whereas the qemu script must be executable.

An example of actual *started* message as sent by the hook script to the control application is displayed in Log snippet 3.4. Parameters are separated by commas. The control application determines the message type based on the very first argument, which is followed by the destination hypervisor, VM name and MAC address. Next, human readable bridge name (*bridge0*) is followed by its datapath ID (i.e. unique identifier of the switch in the network). Similarly, bridge port (*vnet0*) is followed by its identifier within Open vSwitch. The last trio defines whether VLAN tagging is in use, VLAN ID and IP address respectively. These are obtained from OpenNebula, whereas the previous pieces of information come from libvirt. All bits of information as received from the *started* message in Log snippet 3.4 are visually displayed in Figure 3.1.

```
STARTED,PRGNEB002,one-16,02:00:c0:a8:00:fb,bridge0,0000001
    f296a95da,vnet0,50,YES,110,192.168.0.251
```

Log snippet 3.4: Sample *started* message sent by the qemu hook

3.1.2 Communication with ONOS

In terms of communication with ONOS SDN controller, REST API was supposed to be utilised. Unfortunately, ONOS version 1.0.1 released at the end of January 2015, did not have full REST support, i.e. only a subset of resources was available [50].

Therefore, a different approach has been taken. Since ONOS is provided in terms of modules run within Apache Karaf application container, instead of implementing a standalone application communicating with ONOS using well-known API, the control application will be run as an ONOS module, thus having access to internal structures and methods. The obvious drawback of such a solution is the complexity compared to standardised API calls. However, until REST API has been fully implemented, this method represents the only viable opportunity to interact with the SDN controller.

Access to ONOS internal resources is divided into two areas. While any application may access existing topology information, only a subset of applications is allowed to communicate with the ONOS core layer responsible for performing topology changes (adding switch, end host etc.). More specifically, if any application wishes to obtain current topology information, it must register instances of **Service* classes (for instance, *TopologyService*, *HostService*, *DeviceService* etc.). Each service type is responsible for providing a particular set of information. Code snippet 3.5 shows how instances of respective services are requested in the control application. The *@Reference* notation signifies a dependency on another ONOS module providing a particular service instance.

```
// end host information
@Reference(cardinality = ReferenceCardinality.MANDATORY_UNARY)
private HostService m_hostService;

// network devices within the topology
@Reference(cardinality = ReferenceCardinality.MANDATORY_UNARY)
private DeviceService m_deviceService;
```

Code snippet 3.5: Access to internal topology services

Having access to various topology services, the control application may now request specific topology information such as a list of detected end hosts. Code snippet 3.6 comes from a class dedicated to communication with ONOS – *ONOSInteraction*.

```
// return all end stations
public static List<Host> getHosts()
{
    List<Host> h_hosts = Lists.newArrayList(ONOSInteraction.
     t_hostService.getHosts());
    return h_hosts;
}
```

Code snippet 3.6: Sample method utilising access to topology information

ONOS as suggested in previous code snippets distinguishes between infrastructure devices and end hosts. Class *Device* is a representation of infrastructure device managed by OpenFlow protocol. On the other hand, class *Host* is an abstraction of end-station host on the network, essentially a NIC. ONOS defines class *Topology* as a union of all existing devices and links between them. Additionally, topology is able to recover from a link failure as long as alternate paths exist. This means that a *HostToHostIntent* between two hosts is able to recover from a failure condition as long as alternate paths exist, resulting in a rewrite of underlying flows.

However, ONOS concept of topology does not take end hosts into consideration. Instead, the location of host is defined in *HostLocation* class, which defines the infrastructure device and its port number to which the end host is connected. Moreover, the host is further defined by its MAC address, set of IP addresses and VLAN tag. These pieces of information are sent to the control application using the qemu hook as described in section 3.1.1. The fact that the end host is not part of the topology from ONOS perspective implies that a change in *HostLocation* is not transparently detected. Therefore, the control application needs to forcefully remove the original *HostLocation* and replace it with a new location corresponding to the destination hypervisor as received in the *started* message from the qemu hook – see Log snippet 3.4.

At this point, however, another obstacle has been detected. ONOS does not allow applications to modify topology information. Regular applications are thus able to perform read-only operations when it comes to the topology, but cannot modify it. For this purpose the concept of *providers* has been implemented. Providers are in charge of adding/removing end hosts, devices, links and flows – usually based on receipt of some packets (such as ARP/LLDP messages which are used to discover new end hosts).

Providers tend to act reactively, whereas the control application needs to change host location proactively. Therefore, the decision to enhance the functionality of *Host* subcomponent within OpenFlow provider has been made. The current functionality is left intact and the newly implemented logic allows the control application to interact with *HostProviderService* responsible for adding/removing end hosts via the provider. Registering new host takes place in *OpenFlowHostProvider* class using the following Code snippet 3.7.

```
// construct structured entities
HostId hostId = HostId.hostId (macAddress,vlanId);
HostLocation hostLocation = new HostLocation (deviceId,portNumber,
    System.currentTimeMillis());
HostDescription hostDescription = new DefaultHostDescription (
    macAddress,vlanId,hostLocation,ipAddress);

// add the host using new location into the topology
providerService.hostDetected (hostId,hostDescription);
```

Code snippet 3.7: Adding a new host into the topology

In terms of multi-tenancy as one of the functional requirements defined in section 2.1, VLANs are used to separate traffic between multiple tenants or between multiple subnets belonging to a single tenant. The qemu hook obtains VLAN tags from OpenNebula via XML-RPC query along with the VM IP address and passes these pieces of information to the control application. The control application changes host location of the VM undergoing migration and by modifying flow rules it makes sure that all existing communication persists after the migration. However, the control application does not enforce any security measures in terms of permitting communication from VM A to VM

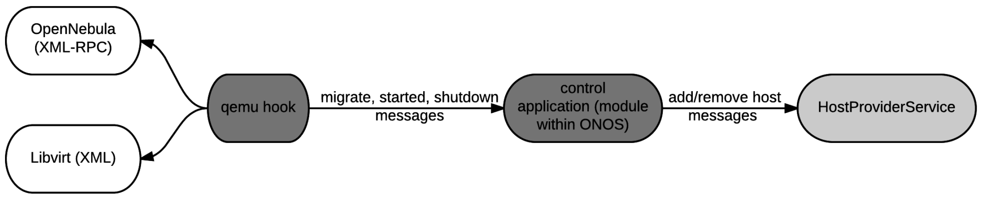


Figure 3.2: Flow of information between all components

B, probably in a different subnet. This should be performed by a firewall-like upper layer application, utilising ONOS north-bound API.

The overall flow of information between all components is depicted in Figure 3.2. The fill color signifies the amount of work implemented in this thesis. While the qemu hook and the control application are constructed from scratch, *HostProviderService* is merely enhanced to provide proactive topology changes.

The purpose of Figure 3.3 is to compare and contrast the original sequence diagram as depicted in Figure 2.4 to the actual implementation. While the original diagram assumed three entities, the actual implementation has resulted in an addition of the fourth – hypervisor embodied by libvirt. Moreover, both the control application and *HostProviderService* modules are actually part of ONOS, whereas the original diagram expected the control application to be a standalone application independent of ONOS.

3.2 Class diagram

Figure 3.4 displays a list of classes and their relationships defined within the control application. Note that the following diagram does not take the qemu hook or *HostProviderService* module into consideration.

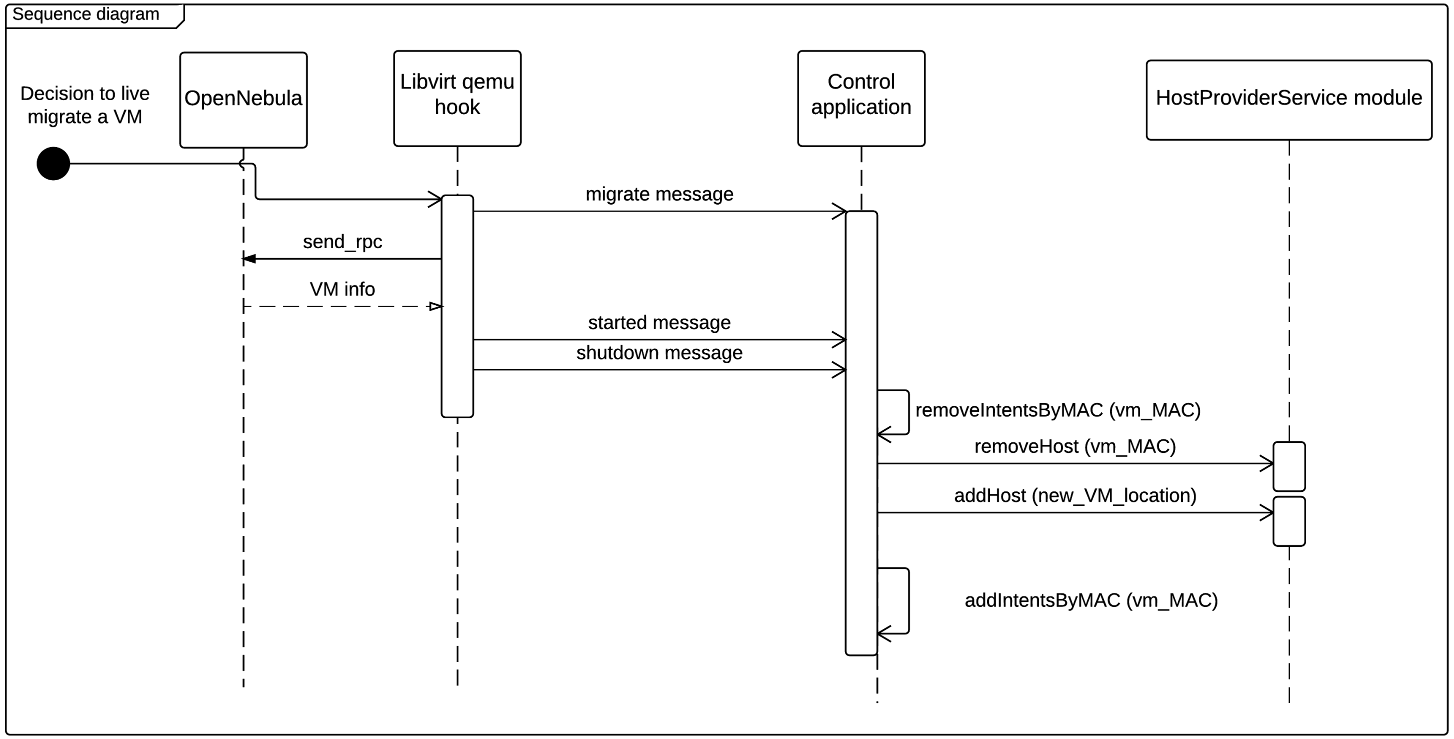


Figure 3.3: Sequence diagram corresponding to the implementation

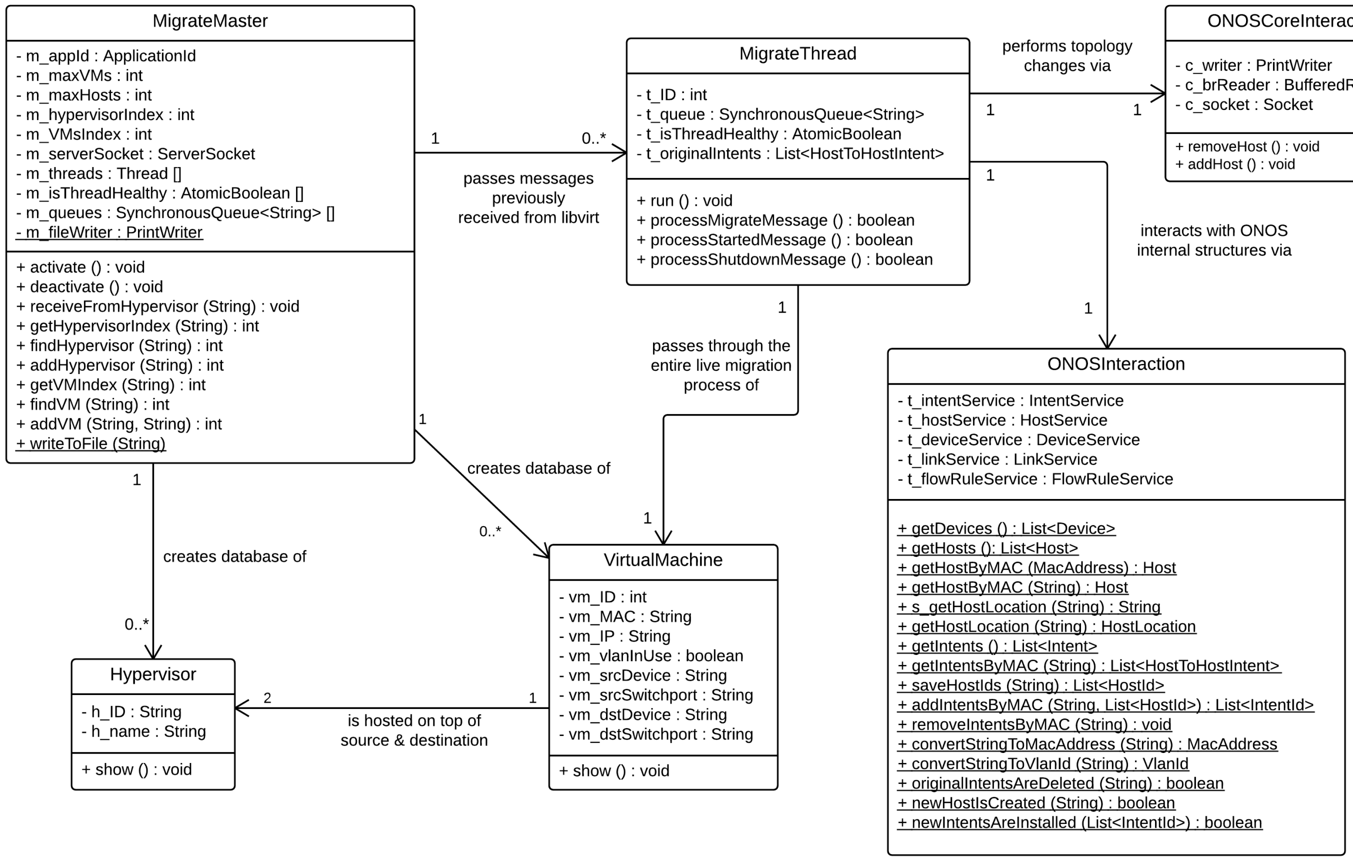

MigrateMaster
- m_appId : ApplicationId
- m_maxVMs : int
- m_maxHosts : int
- m_hypervisorIndex : int
- m_VMsIndex : int
- m_serverSocket : ServerSocket
- m_threads : Thread []
- m_isThreadHealthy : AtomicBoolean []
- m_queues : SynchronousQueue<String> []
- m_fileWriter : PrintWriter
+ activate () : void
+ deactivate () : void
+ receiveFromHypervisor (String) : void
+ getHypervisorIndex (String) : int
+ findHypervisor (String) : int
+ addHypervisor (String) : int
+ getVMIndex (String) : int
+ findVM (String) : int
+ addVM (String, String) : int
+ writeToFile (String)
MigrateThread
- t_ID : int
- t_queue : SynchronousQueue<String>
- t_isThreadHealthy : AtomicBoolean
- t_originalIntents : List<HostToHostIntent>
+ run () : void
+ processMigrateMessage () : boolean
+ processStartedMessage () : boolean
+ processShutdownMessage () : boolean
ONOSCoreInteraction
- c_writer : PrintWriter
- c_brReader : BufferedReader
- c_socket : Socket
+ removeHost () : void
+ addHost () : void
ONOSInteraction
- t_intentService : IntentService
- t_hostService : HostService
- t_deviceService : DeviceService
- t_linkService : LinkService
- t_flowRuleService : FlowRuleService
+ getDevices () : List<Device>
+ getHosts (): List<Host>
+ getHostByMAC (MacAddress) : Host
+ getHostByMAC (String) : Host
+ s_getHostLocation (String) : String
+ getHostLocation (String) : HostLocation
+ getIntents () : List<Intent>
+ getIntentsByMAC (String) : List<HostToHostIntent>
+ saveHostIds (String) : List<HostId>
+ addIntentsByMAC (String, List<HostId>) : List<IntentId>
+ removeIntentsByMAC (String) : void
+ convertStringToMacAddress (String) : MacAddress
+ convertStringToVlanId (String) : VlanId
+ originalIntentsAreDeleted (String) : boolean
+ newHostIsCreated (String) : boolean
+ newIntentsAreInstalled (List<IntentId>) : boolean
VirtualMachine
- vm_ID : int
- vm_MAC : String
- vm_IP : String
- vm_vlanInUse : boolean
- vm_srcDevice : String
- vm_srcSwitchport : String
- vm_dstDevice : String
- vm_dstSwitchport : String
+ show () : void
Hypervisor
- h_ID : String
- h_name : String
+ show () : void
1
0..*
passes messages previously received from libvirt
performs topology changes via
1
1
1
interacts with ONOS internal structures via
1
1
passes through the entire live migration process of
1
1
creates database of
0..*
1
creates database of
0..*
2
1
is hosted on top of source & destination

Figure 3.4: Class diagram

3.3 Description of respective classes

This subchapter defines the purpose of every implemented class within the control application as well as ONOS *HostProviderService* module, through which the control application interacts with the core layer when performing topology changes.

Starting with the control application, class *Hypervisor* defines a physical host on top of which virtual machines may be hosted. Hypervisor information is collected from the qemu hook. *VirtualMachine* class specifies various parameters of a virtual machine detected within the control application. These mostly include networking information such as MAC address, IP address, VLAN tag etc. Besides, *VirtualMachine* is defined by its source and destination *Hypervisor*. These two classes mostly consist of getters and setters.

ONOSInteraction class simulates a static class present in many programming languages with the exception of Java. It consists of static methods handling invocations of internal ONOS functions and procedures. In order to perform this goal, *ONOSInteraction* possesses the access to all required ONOS **Services*, for instance *HostService*, *DeviceService* and *IntentService* among others. The goal of this class is to provide a single point of interaction with ONOS.

ONOSInteraction class consists of methods of three kinds. Firstly, there are some methods which merely act as wrappers around the existing ONOS internal calls without adding any additional functionality. The purpose of such methods is to provide encapsulation for other classes of the control application. An example has already been listed in Code snippet 3.6 – query about existing end hosts.

Secondly, *ONOSInteraction* class comprises of methods converting parameters (stored for instance as *String* objects) into internal ONOS structures. This is performed either via ONOS built-in *valueOf* methods or static constructors taking textual/numeric representation of the structure as the only parameter. The conversion is always associated with appropriate exception handling. Example is displayed in Code snippet 3.8.

```
// return MAC address based on its string representation
public static MacAddress convertStringToMacAddress (String
    s_macAddress)
{
    try
    {
        MacAddress macAddress = MacAddress.valueOf(s_macAddress);
        return macAddress;
    }
    catch (IllegalArgumentException iae)
    {
        MigrateMaster.writeToFile("ONOSInteraction -
    convertStringToMacAddress() - cannot convert String to
    MacAddress.");
        return null;
    }
}
```

Code snippet 3.8: Sample conversion of textual representation to *MacAddress*

Lastly, *ONOSInteraction* class contains methods which are used to implement the logic of control application, for example when regenerating original intents between the VM undergoing migration and all associated devices/end hosts. The original intents are saved during the VM migration and are passed as the function argument. Outline of this method is listed in Code snippet 3.9.

```
// add intents based on VM MAC address & already saved MAC
    addresses of other endpoints
public static List<IntentId> addIntentsByMAC (String vm_MAC, List
    <HostId> vm_otherHostIds)
{
    HostId i_first , i_second;
    List<IntentId> i_intentIds = new ArrayList();
    i_first = ONOSInteraction.getHostIdByMAC (vm_MAC);
    ...
    for (HostId i_otherHostId : vm_otherHostIds)
    {
        ...
        HostToHostIntent i_h2hIntent = new HostToHostIntent(t_appId,
      i_first , i_second);
        t_intentService.submit (i_h2hIntent);
        i_intentIds.add (i_h2hIntent.id());
    }
    return i_intentIds;
}
```

Code snippet 3.9: Regeneration of intents after VM migration

ONOSCoreInteraction class sends requests to add or remove a host to *HostProviderService* module in order to change the location of VM undergoing migration proactively. The communication is asynchronous and takes place using sockets.

Classes *MigrateMaster* and *MigrateThread* are the heart of the control application. While the former receives all messages from libvirt, the latter defines a lifecycle for a single VM migration. When a message from the qemu hook is received, *MigrateMaster* passes the message to the correct instance of *MigrateThread*. Every *MigrateThread* object is run as a separate thread handling a particular VM migration, thus passing through the following methods (assuming that no errors are detected):

1. *processMigrateMessage ()*
2. *processStartedMessage ()*
3. *processShutdownMessage ()*

These methods correspond to a receipt of messages sent by libvirt on behalf of the destination and source hypervisors as depicted in Log snippet 3.2.

MigrateThread and *MigrateMaster* objects must communicate on several occasions. Firstly, *MigrateMaster* passes messages received by libvirt to *MigrateThread* using a Java built-in *SynchronousQueue* object, which basically acts as a rendezvous point between the two threads, allowing the producer to pass a single message to the consumer. This is displayed in Code snippet 3.10 from the perspective of the consumer, i.e. an instance of *MigrateThread* class.

```
// method used for processing STARTED message following previous
    receipt of MIGRATE message
public boolean processStartedMessage ()
{
    // perform blocking read until master thread inserts a new
    STARTED message
    String t_message = this.t_queue.take();
    String [] tokens = t_message.split(",");
    ...
}
```

Code snippet 3.10: *MigrateThread* receipt of mesage from *MigrateMaster*

Furthermore, any error detected by a *MigrateThread* object is passed to the *MigrateMaster* instance using a shared *AtomicBoolean* variable. This basically prevents *MigrateMaster* from passing further messages to *MigrateThread* if an error has been detected while processing a previous message.

It has been implied that the only instance of *MigrateMaster* class contains references to every detected VM. Moreover, the *MigrateMaster* object references the *ONOSInteraction* object – singleton pattern is used for communication with ONOS [51].

Regarding *HostProviderService* – existing component performing topology changes – two additional classes have been implemented. Class *PrepareServer* sets up a *ServerSocket* object and accepts hello/remove requests for a particular VM undergoing migration. The concept of *ExecutorService* has been utilised – instead of creating a dedicated thread to every request, a pool of threads of fixed size has been set up. If a thread is idle and a new request comes in, the request is handled by the thread and upon the completion of the task, the thread is returned to the pool. This is displayed in Log snippet 3.11.

```
while (true)
{
   // listen on given socket accepting client connections
    this.ps_socket = this.ps_serverSocket.accept ();

    // create a new task
    ApplicationHandler aHandler = new ApplicationHandler (this.
    ps_socket);

    // invoke thread
    this.ps_executor.execute (aHandler);
}
```

Code snippet 3.11: *HostProviderService* task assignment

Last but not least, an instance of *ApplicationHandler* class is responsible for performing topology changes proactively based on request type (end host addition or removal). For this purpose, it asynchronously invokes two methods of *HostProviderService*, namely *hostDetected* and *hostVanished* through which the ONOS core layer is notified of a topology change. The asynchronous nature of the two methods implies that *MigrateThread* responsible for a particular VM migration must make sure that host or intent changes have been performed successfully after submitting them. Original class *InternalHostProvider*, which adds end hosts reactively based on receipt of ARP packets, has not been tampered with.

Chapter 4

Evaluation

The purpose of this chapter is to evaluate whether the application, which utilises the concepts of SDN, may be deployed in a small/medium-sized data center when performing live VM migration. The SDN approach utilising intents and underlying flows will be compared to the traditional non-SDN world, which heavily relies on L2 switching. Before the description of the actual methodology and subsequent evaluation the lab environment will be described in detail.

4.1 Lab environment

In order to create a DC-like architecture, *leaf and spine* design has been designated. Leaf and spine, which is the implementation of Clos networks, distinguishes between leaves (effectively represented by ToR switches) and spines corresponding to core switches. Each leaf is connected to all spine switches and the same principle applies vice versa. Additionally, there are no links between switches at the same level. This design promotes high availability and resiliency [52]. At the same time it provides an equal distance in terms of number of hops for every two endpoints in the fabric [53].

Leaf and spine architecture was initially implemented using six physical switches of multiple vendors – HP with their Procurve and Comware series and Cisco Catalyst series. All physical switches support OpenFlow 1.3 and hence were supposed to be managed by an instance of ONOS SDN controller. Furthermore, two physical servers – HP DL360 G7 and Dell R210 II – were connected to leaf switches at the opposite edge of the network. The physical servers run OpenNebula and host virtual machines, which are migrated from the original server across the entire network to the other host. Both servers run Open vSwitch and hence are also managed by ONOS via OpenFlow 1.3. ONOS SDN controller is run within a virtual machine hosted by one of the servers. The ONOS virtual machine is run independently of OpenNebula orchestrator system.

Vendor	**Series**	**Fimrware version**	**Number of pieces**
HP	ProCurve E3800	KA.15.16.0004	2
HP	ProCurve 5406zl	K.15.16.0004	1
HP	Comware A5500	5.20.99, release 5501P03	1
Cisco	Catalyst 3650	03.07.50.SFT	2
Open vSwitch	N/A	2.3.0	2

Table 4.1: Switches employed in the application evaluation

The overall picture of the topology is portrayed in Figure 4.1. The red links which belong to *controlled* VLAN are detected and managed by ONOS SDN controller, whereas the blue links corresponding to *controlling* VLAN are managed using the traditional L2 switching. In other words, *out-of-band* approach to SDN adoption has been selected. OpenNebula-managed virtual machines are migrated from host *URAN001* (the HP server) to *URAN002* (the Dell server) and vice versa. The actual migration takes place in the *controlling* VLAN, so it is not managed via OpenFlow. The communication among tenant VMs is, however, controlled via OpenFlow. Additional management VLAN is employed to provide access to the physical switches. In the network topology depicted in Figure 4.1 separate links are used for each traffic type (the management VLAN is not depicted for the sake of clarity). The network topology therefore follows generally accepted network design principles in which a separate VLAN is dedicated to user traffic (i.e. a tenant), management and vMotion (vMotion as an example of enterprise-level live VM migration technology is sent unencrypted and therefore should not be mixed with other traffic types [54]). The list of switches as employed in Figure 4.1 along with their firmware revisions is displayed in Table 4.1.

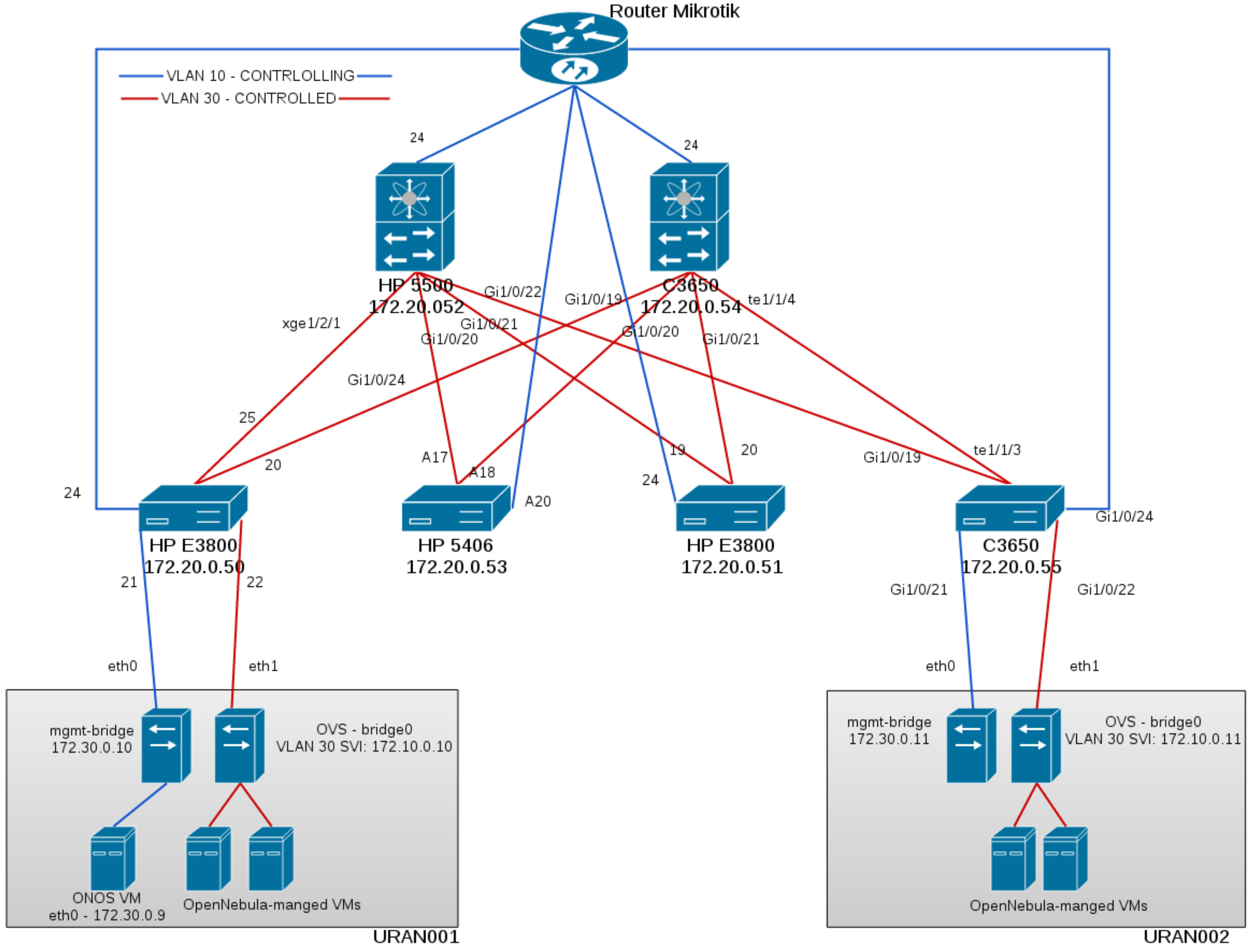


Figure 4.1: Leaf and spine topology prepared for the application evaluation

Once the application has been implemented and network topology set up according to Figure 4.1, a major obstacle was detected – ONOS as of version 1.0.1 suffered from severe problems in terms of managing the above-mentioned switches with the exception of Open vSwitch. This did not turn out during the application implementation itself because the application has initially been tested against Open vSwitches, which have worked satisfactorily.

In terms of specific malfunctions, switches of different vendors implement OpenFlow differently. For instance, HP switches use three OpenFlow tables in which they store respective flow rules. These are tables 0, 100 and 200. Table 0 is read-only and performs operation ”goto” table 100. Tables 100 and 200 have a default drop rule at the end. As it turned out after a discussion with one of ONOS developers, ONOS SDN controller ”sends layer 2 rules down to table 0 of the switch naively expecting a single table abstraction” [55]. Unsurprisingly, HP switches refuse to add flow rules to read-only table 0 resulting in HP switches being completely unmanageable by ONOS SDN controller.

Cisco Catalyst switches suffer from a less obvious, but equally serious flaw. Both switches are successfully connected to ONOS, but they tend to disappear soon afterwards and this cycle keeps repeating endlessly. ONOS reports a cryptic error as depicted in Log snippet 4.1. The ONOS developer this time could not detect the cause of the issue blaming ”poor implementation” of the vendor or ”some vendor extensions” [55]. This may be partly true, since the firmware revision 03.07.50.SFT is a nightly build used predominantly for development purposes. Cisco was not able to provide a more recent and stable build once the issues with connectivity persistence have been discovered.

```
2015-03-26 15:21:32,802 | ERROR | ew I/O worker #6 |
    OFChannelHandler | 74 - org.onosproject.onos-of-ctl - 1.0.1 |
    Disconnecting switch org.onosproject.openflow.drivers.
    DriverManager$1@57616091 due to message parse failure
org.projectfloodlight.openflow.exceptions.OFParseError: Unknown
    value for discriminator typeLen of class OFOxmVer13: 67588
         at org.projectfloodlight.openflow.protocol.ver13.
    OFOxmVer13$Reader.readFrom(OFOxmVer13.java:388)
         at org.projectfloodlight.openflow.protocol.ver13.
    OFOxmVer13$Reader.readFrom(OFOxmVer13.java:36)
         ...
         at java.util.concurrent.ThreadPoolExecutor$Worker.run(
    ThreadPoolExecutor.java:617) [:1.8.0_40]
         at java.lang.Thread.run(Thread.java:745) [:1.8.0_40]
```

Log snippet 4.1: Cryptic error reported by ONOS when managing Cisco switch

All in all, leaf and spine topology consisting of both physical and software switches as displayed in Figure 4.1, could not be used to perform the application evaluation.

Therefore, an alternative approach consisting of merely Open vSwitches had to be considered and implemented. In order to utilise leaf and spine design,

Hostname	URAN001	URAN002	URAN003	APU00*
Type	HP Proliant DL360 G7	Dell Poweredge R210 II	Cisco WAVE-274	PC Engines APU
CPU	Intel Xeon E5506 2.13GHz	Intel Xeon E3-1220 V2 3.10GHz	Intel Core2 Duo CPU E6400 2.13GHz	AMD G-T40E Processor 1000MHz
RAM	14 GB	8 GB	6 GB	4 GB
HDD	130 GB	443 GB	141 GB	3.6 GB
Distribution	Debian 7	Debian 7	Debian 7	Archlinux
Kernel	3.2.0-4-amd64	3.2.0-4-amd64	3.2.0-4-amd64	3.19.3-3-ARCH
Role within topology	OpenNebula master node	OpenNebula compute node	simulates external client (3rd test case)	leaf/spine switches

Table 4.2: Description of available servers

physical switches had to be replaced with dedicated physical servers running Open vSwitch. One physical server simulates the spine layer, while other two hosts replace leaf switches. The three servers simulating physical switches come from a family of APU architecture and are hence labelled *APU001* to *APU003*. Existing physical servers – *URAN001* and *URAN002* are connected to the leaf switches. These servers run OpenNebula services and host virtual machines. Additional dedicated server is connected directly to the spine switch – *URAN003*.

In conclusion, the final topology is depicted in Figure 4.2. VM migration is performed via the blue-colored *controlling* VLAN from *URAN001* to *URAN002* and vice versa, whereas the red-colored *controlled* VLAN is entirely managed via the means of ONOS SDN controller, but it may easily become a regular L2-switched VLAN if Open vSwitches stop being managed by the external controller.

Table 4.2 summarises the list of available servers, notably their hardware configuration and role within the final topology as depicted in Figure 4.2.

4.2 Methodology

This section attempts to describe the fashion in which the application will be evaluated. The methodology will be consistent with small/medium-sized data center requirements.

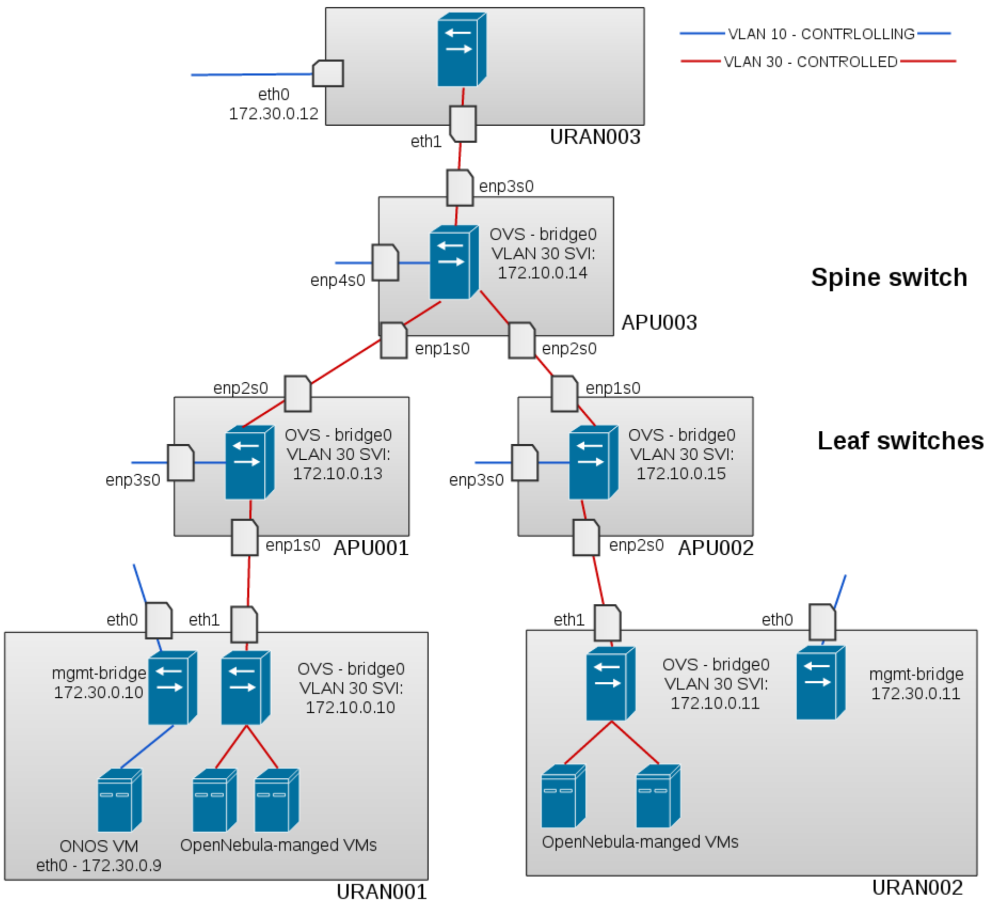


Figure 4.2: Actual networking topology used for the application evaluation

Cisco states that approximately 75 percent of the data center traffic is local and hence never leaves the premises of DC [56]. This is also labelled as east-west communication. Based on the report the trend is not supposed to change in the years to come. The campus network, on the other hand, follows 20/80 rule – most of the traffic is now passed across the core layer, which corresponds to north-south traffic pattern [57]. There are a few examples, which promote the east-west traffic pattern in data centers, namely:

- communication between web/application and database server – short SQL select may fetch a large portion of the database,
- parallel computing – a master node allocates tasks which are computed by slave nodes passing the complex results back to the master.

These examples not only emphasise the east-west traffic flow, but also non-uniform amount of traffic passing between the communicating entities.

Tool	iperf	flowping
Entity measured	throughput (TCP)	latency, packet loss (UDP)
SW version	2.0.5	1.2.6
Server invocation	iperf -s --interval 1	flowping -S -q
Client invocation	iperf -c SERVER -f m -t 50 --interval 1	flowping -h SERVER -c 50 -f FILE

Table 4.3: Description of tools employed during the application evaluation

These traffic flow patterns will be simulated using *iperf* and *flowping* utilities. Iperf will be used to measure throughput for TCP communication between an iperf client and server. The traffic is sent by the iperf client and received on the server side [58]. Flowping, on the other hand, is UDP-based application, which provides an alternative to regular *ping* utility [59]. Unlike ping, which measures latency at L3, flowping measures latency as detected by an application layer, since it operates at L7 of OSI model. Iperf and flowping measurements will be run simultaneously. SW versions and options employed on both client and server sides are listed in Table 4.3.

The below mentioned scenarios will be evaluated for both SDN and non-SDN cases. The SDN scenario utilises the application implemented in section 3, whereas the non-SDN scenario will employ the traditional L2 switching. Both will use the topology as displayed in Figure 4.2 – the only difference being the fact that in the non-SDN mode Open vSwitches handle forwarding themselves without being managed by an external SDN controller. The evaluated test cases are as follows:

1. VM being migrated (iperf client) vs. static VM (iperf server). The VM being migrated will act as a database server, while the static VM will represent an application/web server. This is schematically displayed in Figure 4.3.

2. VM being migrated (iperf server) vs. static VM (iperf client). The VM being migrated will act as a web/application server, whereas the static VM will represent a DB server. This is depicted in Figure 4.4.

3. VM being migrated (iperf client) vs. physical host (iperf server). The VM being migrated will act as a server for external client (e.g. web, FTP, application server). This use case simulates external traffic leaving the DC premises. The external client performs download operation. This is portrayed in Figure 4.5.

The grey dotted lines signify the flow of VM migration process using the *controlling* VLAN, while the light blue dotted lines represent the communication in terms of iperf/flowping using the *controlled* VLAN.

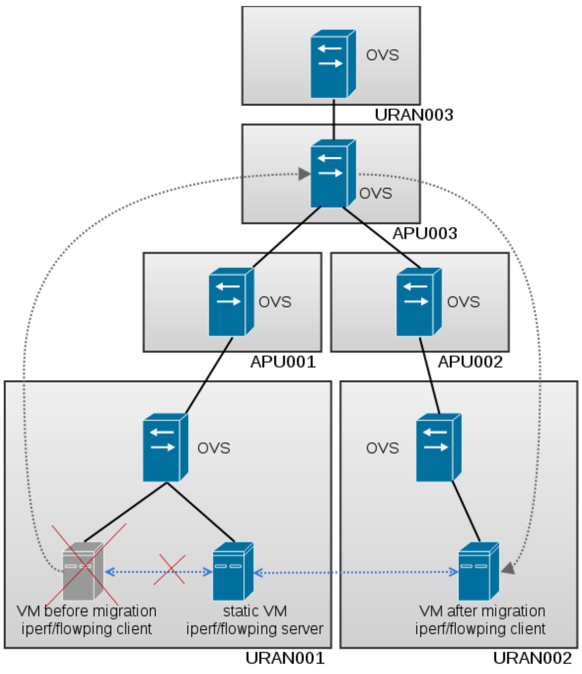


Figure 4.3: First test case

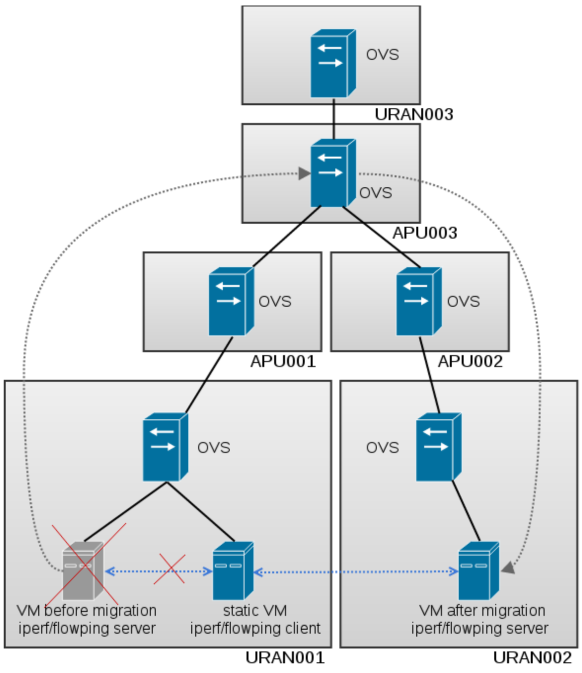


Figure 4.4: Second test case

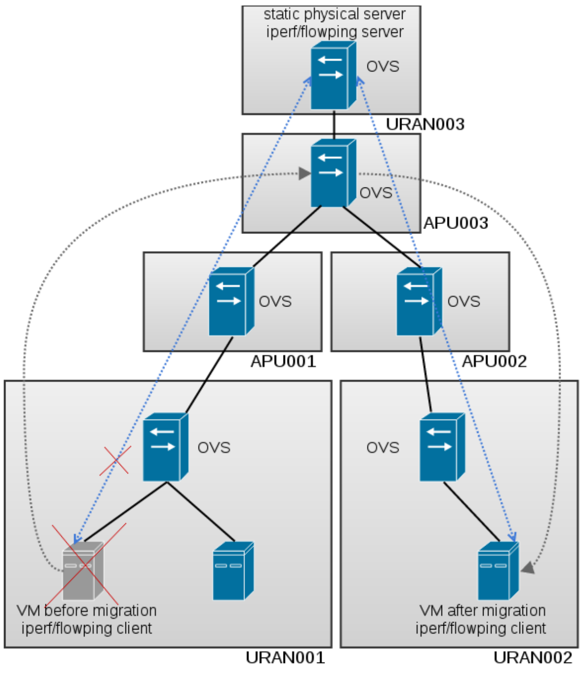


Figure 4.5: Third test case

The above-mentioned three test cases will be performed for both the SDN- and non-SDN world. In the SDN scenario, ONOS and the application implemented in section 3 will be invoked and required intents will be set up manually. Afterwards, each of the above-mentioned test cases consisting of live VM migration request, simultaneous iperf and flowping measurements will be run 20 times in the same direction, which altogether entails 40 live VM migrations per test case (20 live migrations will be the subject of the measurements and 20 live migrations will revert the VM in question back to the original hypervisor so that another iteration of iperf/flowping measurements can be invoked). Since a VM migration given the topology in use takes approximately 20 seconds, a 35-second time frame will be used for iperf and flowping measurements. This should be sufficient for the network and underlying communication to become stable after the VM migration. In the end, average values of throughput (based on iperf results) and latency (based on flowing output) throughout the 35 second period will be obtained on a per-second time frame. Additionally, average packet loss throughout all 20 migrations will be measured as well. These principles also apply to the non-SDN scenario, but this time switches will not be managed by ONOS SDN controller, but instead they will use legacy L2 forwarding.

4.3 Results

The following section points out the results of the evaluation based on the three test cases defined in the previous section 4.2. All test case measurements were performed using both the SDN-based and the non-SDN-based principles. The resultant figures 4.6 up to 4.11 corresponding to either iperf throughput or flowping latency measurements are an average of 20 VM migrations. What is common to all of the following measurements is the fact that VM migration finishes between 18th and 22nd second. During this critical time frame average packet loss and latency increases, whereas throughput drops. This applies to both the SDN-based and the non-SDN-based scenarios. The following paragraphs will discuss the extent to which SDN and the traditional L2 forwarding differ in terms of performance during the critical time period.

Figures 4.6 and 4.7 correspond to the first test case, in which the migrated VM acts as an an iperf/flowping client against a static VM running iperf and flowping server applications. Originally, both virtual machines reside on the same host, but once the migration has been completed, the traffic between the VMs spans across the entire network. The SDN approach performs slightly better compared to the traditional L2 forwarding in terms of packet loss and latency. SDN loses on average 2.25 % packets during the entire migration, while 2.40 % of packets is lost in the non-SDN approach. Similarly, flowping downtime during the critical part of VM migration is on average shorter in the SDN case, 0.75 second vs 0.80 second in the non-SDN case. SDN excels in terms of throughout during the critical phase of VM migration – 34 Mbps is more than twice as much as the traditional L2 forwarding can inject (almost 14 Mbps).

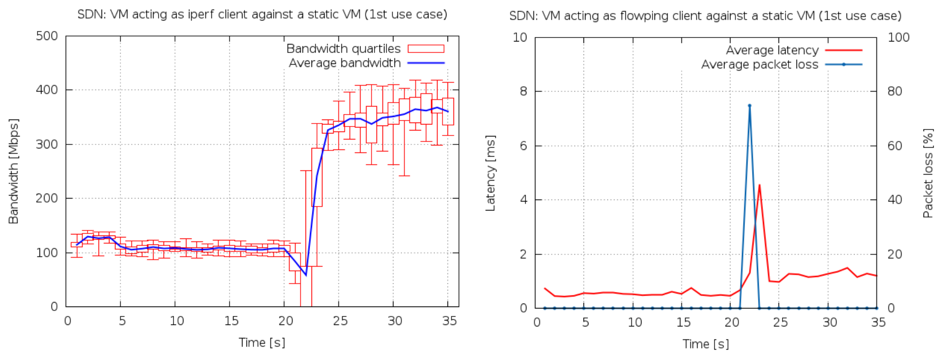


Figure 4.6: SDN: VM acting as iperf/flowping client against a static VM (1st use case)

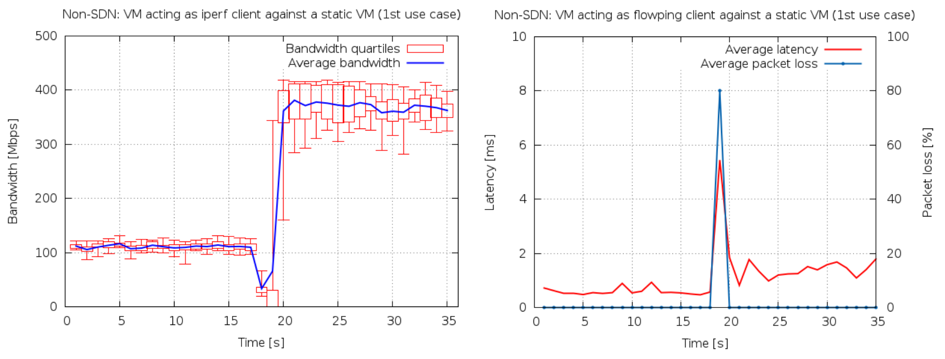


Figure 4.7: Non-SDN: VM acting as iperf/flowping client against a static VM (1st use case)

Figures 4.8 and 4.9 represent the second test case, in which the migrated VM acts as an iperf/flowping server against a static VM running iperf and flowping client applications. Similarly to the first use case, the traffic between the two VMs spans across the entire network after the VM migration. The SDN case is superior to its non-SDN counterpart in all major aspects – average packet loss during the entire migration (2.85 % compared to 3.75 % in the traditional L2 forwarding), flowping downtime during the critical part of VM migration (0.9 second vs 1.2 seconds) and last but not least, throughput during the transition (15.5 Mbps for the SDN case is almost double the figure obtained in the non-SDN case – 7.5 Mbps).

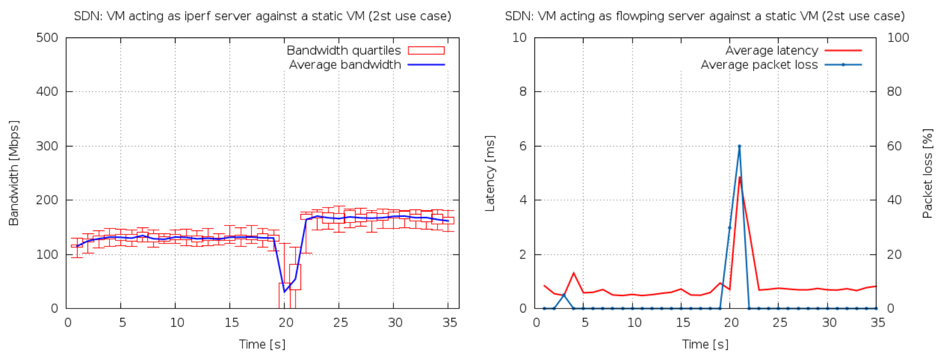


Figure 4.8: SDN: VM acting as iperf/flowping server against a static VM (2nd use case)

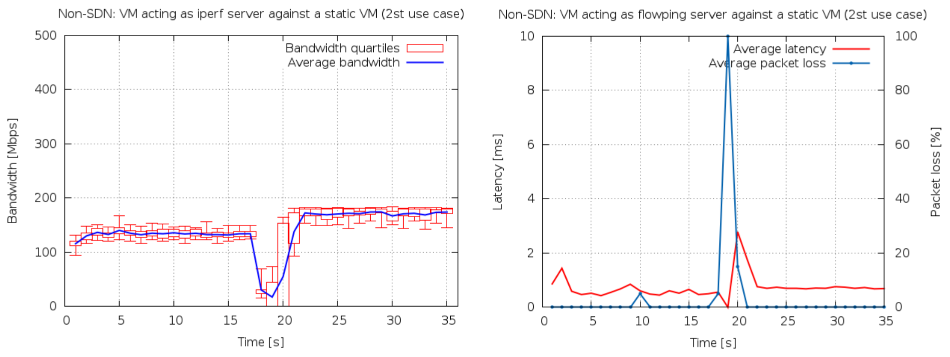


Figure 4.9: Non-SDN: VM acting as iperf/flowping server against a static VM (2nd use case)

Figures 4.10 and 4.11 represent the third test case, in which the migrated VM acts as an iperf/flowping client against an external host running iperf and flowping server applications. Unlike the two previous cases, which focus on intra-DC communication, this test case represents communication between the DC and the external world. The output of average throughput as recognised by iperf resembles the first use case for both the SDN and the non-SDN cases. This is analysed in the next paragraph. In terms of comparing the SDN-based and the traditional approach, SDN outperforms its competitor in all major respects. Similarly to the two previous test cases, iperf client can inject twice as much traffic into the network during the critical part of migration when compared to the non-SDN-based approach (31 Mbps vs. 15.7 Mbps).

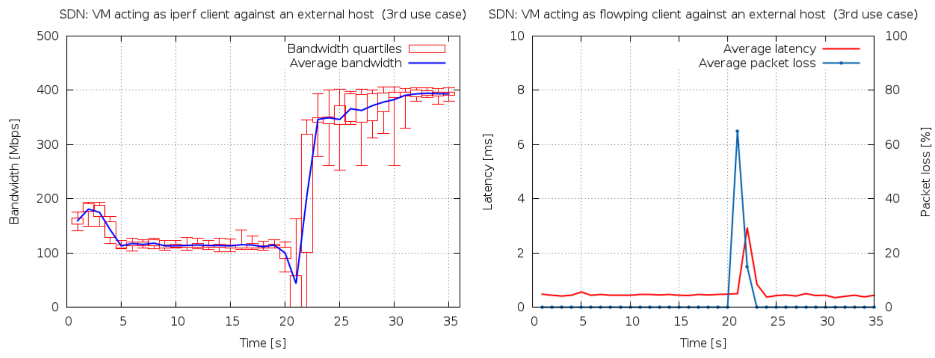


Figure 4.10: SDN: VM acting as iperf/flowping client against an external client (3rd use case)

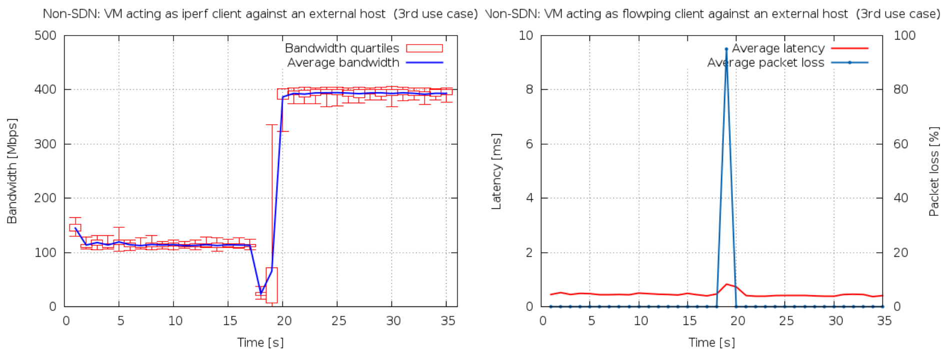


Figure 4.11: Non-SDN: VM acting as iperf/flowping client against an external client (3rd use case)

Aspect	**SDN**	**Non-SDN**
Total packet loss	2.25 ± 1.33 %	2.40 ± 1.23 %
Lack of connectivity during transition	0.75 ± 0.44 s	0.80 ± 0.41 s
Minimal throughput during transition	34.26 ± 34.05 Mbps	13.68 ± 15.33 Mbps
Maximal latency during transition	4.76 ± 0.74 s	3.41 ± 2.53 s

Table 4.4: First test case performance comparison

Aspect	**SDN**	**Non-SDN**
Total packet loss	2.85 ± 1.18 %	3.75 ± 1.33 %
Lack of connectivity during transition	0.90 ± 0.30 s	1.20 ± 0.41 s
Minimal throughput during transition	15.58 ± 14.93 Mbps	7.45 ± 11.54 Mbps
Maximal latency during transition	4.50 ± 0.49 s	2.94 ± 1.23 s

Table 4.5: Second test case performance comparison

Test cases 1 and 3 show a sharp increase in throughput after the end of VM migration for both the SDN and the non-SDN scenarios. This may be caused by a combination of the following two factors:

- different HW configuration of hosts *URAN001* and *URAN002* and
- utilisation of both servers differs since *URAN001* hosts an additional VM.

Therefore, *URAN001* may have fewer computational resources as required by the iperf client (i.e. VM undergoing migration), which attempts to inject a large amount of traffic into the network.

Tables 4.4, 4.5 and 4.6 compare and contrast the results of respective test cases for both the SDN and the non-SDN scenarios. Besides average packet loss throughout the entire migration, average downtime and minimal throughput during the critical phase of VM migration, the tables contain *expected values* and *corrected standard deviations* for maximum latency as detected by flowping during the 5-second time frame (18th-22nd second). These figures are of lesser importance because they comprise of traffic that was actually received by the flowping server and do not consider lost traffic.

The results also point out the fact that a large standard deviation is detected in all cases for minimal throughput measurements – it is approximately

Aspect	**SDN**	**Non-SDN**
Total packet loss	2.40 ± 1.23 %	2.85 ± 0.67 %
Lack of connectivity during transition	0.75 ± 0.44 s	0.95 ± 0.22 s
Minimal throughput during transition	31.20 ± 36.63 Mbps	15.68 ± 9.72 Mbps
Maximal latency during transition	2.89 ± 0.37 s	0.80 ± 0.60 s

Table 4.6: Third test case performance comparison

equal to the expected value. This is caused by the fact that VM migrations for a single test case tend to finish in a 1-3 second time frame and hence the resulting throughput varies from 0 Mbps to approximately 2x expected value. However, it is important to state that this behaviour is common to both the SDN-based and the non-SDN-based scenarios for all test cases. Hence, the expected value can still be considered a valid performance indicator.

In conclusion, proactive topology changes via the means of SDN result in better performance during the critical phase of VM migration for all relevant attributes (service downtime, throughput). Average packet loss throughout the entire migration is also lower in the SDN case. Effectively, if iperf and flowping were replaced by real-world services, VM undergoing migration would probably lose fewer TCP sessions (which corresponds to a decrease in flowping downtime) while serving more clients (which corresponds to an increase in iperf throughput).

Conclusion

The purpose of the thesis was to analyse the extent to which the novel concept of software-defined networking represents a feasible alternative to the traditional networking concepts in the small/medium-sized data center environment. The thesis specifically focused on a single use case – service availability enhancement during live VM migration.

Chapter 1 familiarises the reader with the state of the art in several related fields varying from the definition and requirements imposed on SDN controllers, currently available overlay networks providing multi-tenancy to a description of virtual switches as employed by hypervisors.

In chapters 2 and 3 the application maintaining flow rules among the communicating entities is outlined and subsequently implemented. The application is run as a module within ONOS SDN controller and gathers information from multiple sources, namely libvirt virtualisation API and OpenNebula orchestrator system. The application proactively performs topology changes (i.e. relocation of the VM undergoing migration) in accordance with the SDN principles.

Chapter 4 attempts to evaluate the application using leaf-and-spine network topology, which is commonplace in data centers. Since the original topology consisting of both physical and virtual switches cannot be fully managed by ONOS SDN controller due to particular ONOS limitations described in detail in page 54, an alternative topology comprised of virtual switches is created. Throughput, latency and packet loss are measured for three particular use cases, which represent both intra-DC and external communication. The measurements are carried out for both the SDN-based and the non-SDN-based scenarios.

Based on the measurement results it follows that the proactive approach taken by the the application implemented in chapter 3 decreases packet loss and provides larger throughput during the critical phase of VM migration. If iperf and flowping – tools performing the quantitative evaluation – were replaced by real-world services, more requests would probably be handled by the VM undergoing migration while losing fewer existing TCP sessions.

The implemented application meets the requirements defined in subchapter 2.1 with the exception of the third point – savings in terms of network bandwidth and CPU processing. While the proactive approach eliminates flooding of unknown unicast frames performed by switches in the traditional L2 forwarding, SDN inherently induces an additional layer of complexity – communication with the external controller. In terms of multi-tenancy as another requirement, the application gathers VLAN configuration corresponding to each of the communicating endpoints and changes the endpoint location within a given VLAN, but does not enforce any security measures which would prevent from inter-VLAN communication.

Although the SDN-based approach does not outperform the traditional L2 forwarding dramatically, one should keep in mind that SDN is more than a novel traffic forwarding mechanism – instead, port security and perhaps other security-related issues may potentially be solved in a programatic manner via SDN.

On the other hand, the problems encountered while trying to manage physical switches of multiple vendors by ONOS SDN controller emphasise the current immaturity of SDN implementations. However, I believe that the ability to program networks, which is the principal advantage of SDN over traditional networking, will prevail and SDN will eventually become commonplace in certain environments, data centers being one of them.

Bibliography

[1] Vxchange. Scalable, secure and energy-efficient data centers. http://www.vxchnge.com/data-centers/, 2015, accessed: 21 January 2015.

[2] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf, 2012, accessed: 06 February 2015.

[3] Salisbury, B. The Control Plane, Data Plane and Forwarding Plane in Networks. http://networkstatic.net/the-control-plane-data-plane-and-forwarding-plane-in-networks/, 2012, accessed: 06 February 2015.

[4] Bruno Nunes Astuto, Marc Mendonca, Xuan Nam Nguyen, Katia Obraczka, Thierry Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *Communications Surveys and Tutorials, IEEE*, volume 16, 2014, ISSN 1553-877X.

[5] IETF. Ipsilon's General Switch Management Protocol Specification Version 1.1. https://tools.ietf.org/html/rfc1987, 1996, accessed: 06 February 2015.

[6] Kerner, S. M. Big Switch Emerges with Commercial SDN Portfolio. http://www.enterprisenetworkingplanet.com/datacenter/big-switch-emerges-with-commercial-sdn-portfolio.html, 2012, accessed: 07 February 2015.

[7] The OpenDaylight Project. About Open Daylight. http://www.opendaylight.org/project/about, 2015, accessed: 07 February 2015.

[8] The OpenDaylight Project. OpenDaylight Controller: Architectural Framework. https://wiki.opendaylight.org/view/

OpenDaylight_Controller:Architectural_Framework, 2013, accessed: 07 February 2015.

[9] The OpenDaylight Project. OpenDaylight Virtual Tenant Network (VTN): Overview. https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_%28VTN%29:Overview, 2015, accessed: 07 February 2015.

[10] The OpenDaylight Project. OpenDOVE: Proposal. https://wiki.opendaylight.org/view/Open_DOVE:Proposal, 2015, accessed: 07 February 2015.

[11] ON.Lab. ONOS Overview. http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf, 2014, accessed: 07 February 2015.

[12] ON.Lab. Driving SDN Adoption in Service Provider Networks. http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-Service-Provider-SDN-final.pdf, 2014, accessed: 07 February 2015.

[13] Casado, M. List of OpenFlow Software Projects. http://yuba.stanford.edu/~casado/of-sw.html, accessed: 08 February 2015.

[14] University of Virginia: Jorg Liebeherr. Lecture nodes: Overlay Networks 1. http://www.cs.virginia.edu/~cs757/slidespdf/757-09-overlay.pdf, 2003, accessed: 02 February 2015.

[15] Pepelnjak, I. A Day in a Life of an Overlaid Virtual Packet. http://blog.ipspace.net/2013/08/a-day-in-life-of-overlaid-virtual-packet.html, 2013, accessed: 03 February 2015.

[16] IETF. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. http://tools.ietf.org/html/rfc7348, 2014, accessed: 03 February 2015.

[17] Pepelnjak, I. Unicast-only VXLAN finally shipping. http://blog.ipspace.net/2013/07/unicast-only-vxlan-finally-shipping.html, 2013, accessed: 03 February 2015.

[18] Raffe, A. Enhanced VXLAN: Who needs multicast? http://adamraffe.com/2013/06/24/enhanced-vxlan-who-needs-multicast/, 2013, accessed: 05 February 2015.

[19] Cisco. Cisco Nexus 1000V release notes, release 5.2(1)SV3(1.1). http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus1000/sw/5_2_1_s_v_3_1_1/release/notes/n1000v_rn.html#pgfId-213751, 2015, accessed: 03 February 2015.

[20] Bedem, R. V. D. Tech101 – VMware NSX. http://vcdx133.com/2015/01/06/tech101-vmware-nsx/, 2015, accessed: 05 February 2015.

[21] Lowe, S. Learning NVP, Part 1: High-Level Architecture. http://blog.scottlowe.org/2013/05/21/learning-nvp-part-1-high-level-architecture/, 2013, accessed: 05 February 2015.

[22] Pepelnjak, I. Layer-2 and Layer-3 Switching in VMware NSX. http://blog.ipspace.net/2013/11/layer-2-and-layer-3-switching-in-vmware.html, 2013, accessed: 05 February 2015.

[23] VMware. VMware Virtual Networking Concepts. http://www.vmware.com/files/pdf/virtual_networking_concepts.pdf, 2015, accessed: 25 January 2015.

[24] VMware. vSphere 5 Command-Line Documentation. http://pubs.vmware.com/vsphere-50/topic/com.vmware.vcli.examples.doc_50/cli_manage_networks.11.4.html, 2014, accessed: 25 January 2015.

[25] Cisco. Cisco SAFE Reference Guide. http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Security/SAFE_RG/SAFE_rg.pdf, 2010, accessed: 25 January 2015.

[26] Pepelnjak, I. VMware virtual switch: no need for STP. http://blog.ipspace.net/2010/11/vmware-virtual-switch-no-need-for-stp.html, 2010, accessed: 25 January 2015.

[27] Oswalt, M. ESXi switch load balancing woes. http://keepingitclassless.net/2013/05/esxi-vswitch-load-balancing-woes/, 2013, accessed: 25 January 2015.

[28] Stretch, J. EtherChannel considerations. http://packetlife.net/blog/2010/jan/18/etherchannel-considerations/, 2010, accessed: 25 January 2015.

[29] Wahl, C. Using LACP with a vSphere Distributed Switch 5.1. http://wahlnetwork.com/2012/10/15/using-lacp-with-a-vsphere-distributed-switch-5-1/, 2012, accessed: 25 April 2015.

[30] Cisco. Cisco Nexus 1000V virtual switch data sheet. https://ciscodatacenter.files.wordpress.com/2010/03/data_sheet_nexus-1k.pdf, 2008, accessed: 26 January 2015.

[31] Cisco. Cisco Nexus 1000V series switches data sheet. http://www.vmware.com/files/pdf/Cisco-Nexus-Network-Analysis-Module-DS-EN.pdf, 2012, accessed: 26 January 2015.

[32] Pepelnjak, I. VLANs used by Nexus 1000V. http://blog.ipspace.net/2011/08/vlans-used-by-nexus-1000v.html, 2011, accessed: 26 January 2015.

[33] Pepelnjak, I. What is OpenFlow (part 2)? http://blog.ipspace.net/2011/10/what-is-openflow-part-2.html, 2011, accessed: 27 January 2015.

[34] Oswalt, M. Introduction to Open vSwitch. http://keepingitclassless.net/2013/10/introduction-to-open-vswitch/, 2013, accessed: 27 January 2015.

[35] Pettit, J. Open vSwitch 1.11.0 Available. http://openvswitch.org/pipermail/announce/2013-August/000054.html, 2013, accessed: 27 January 2015.

[36] Cisco. Cisco Nexus 1000V Essential and Advanced Editions FAQ. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-1000v-switch-vmware-vsphere/qa_c67-717571.html, 2013, accessed: 28 January 2015.

[37] VMware. vSphere pricing. http://www.vmware.com/products/vsphere/pricing, 2015, accessed: 28 January 2015.

[38] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, Sujata Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. *Proceedings of the ACM SIGCOMM 2011 conference*, 2011.

[39] Ungerman, J. Cisco Connect: OpenFlow. http://www.cisco.com/web/CZ/ciscoconnect/2014/assets/tech_sdn2_sp_api_openflow_ungerman.pdf, 2014, accessed: 09 February 2015.

[40] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, Andrew Warfield. Live Migration of Virtual Machines. *Proceeding NSDI'05 Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation*, volume 2, 2005.

[41] Cisco. Catalyst 6500/6000 Switches ARP or CAM Table Issues Troubleshooting. http://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/71079-arp-cam-tableissues.html, 2009, accessed: 17 February 2015.

[42] Jalaparti, V. [Xen-users] Xen live migration: from where is the ARP sent? http://www.gossamer-threads.com/lists/xen/research/201826, 2011, accessed: 17 February 2015.

[43] Kooman, S. VM_HOOK scripts to send Gratuitous ARP replies on behalf of VMs in OpenNebula cloud (ONE). https://github.com/hydro-b/one-grarp, 2014, accessed: 17 February 2015.

[44] Nobel, R. The vSwitch "Notify Switches" setting. http://rickardnobel.se/vswitch-notify-switches-setting/, 2012, accessed: 17 February 2015.

[45] VMware. Virtual machine live migration with vSphere vMotion. http://www.vmware.com/cz/products/vsphere/features/vmotion, 2015, accessed: 17 February 2015.

[46] Klepáč, M. Private IaaS cloud comparison. https://dip.felk.cvut.cz/browse/details.php?f=F8&d=K104&y=2013&a=klepamar&t=bach, 2013, accessed: 27 February 2015.

[47] OpenNebula Project. XML-RPC API. http://docs.opennebula.org/4.12/integration/system_interfaces/api.html, 2015, accessed: 21 March 2015.

[48] Ruben S. Montero. How does opennebula monitor the vm's state. http://lists.opennebula.org/pipermail/users-opennebula.org/2012-November/038418.html, 2012, accessed: 21 March 2015.

[49] Libvirt virtualization API. Hooks for specific system management. https://www.libvirt.org/hooks.html, accessed: 21 March 2015.

[50] Koshibe, A. ONOS Developer's guide: REST API (Draft). https://wiki.onosproject.org/pages/viewpage.action?pageId=1048699, 2015, accessed: 21 March 2015.

[51] Gupta, L. Singleton design pattern in java. http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/, accessed: 02 April 2015.

[52] Hogg, S. Clos Networks: What's Old Is New Again. http://www.networkworld.com/article/2226122/cisco-subnet/clos-networks--what-s-old-is-new-again.html, 2014, accessed: 11 April 2015.

[53] Booth, R. Cisco Nexus - Part 2 - Design Basics. http://blog.movingonesandzeros.net/2013/05/cisco-nexus-part-2-design-basics.html, 2013, accessed: 11 April 2015.

[54] Brown, M. Isolating vMotion traffic. `https://communities.vmware.com/message/2347811`, 2010, accessed: 18 April 2015.

[55] Klepáč, M. onos-dev mailing list: ONOS vs. physical switches. `https://groups.google.com/a/onosproject.org/forum/#!topic/onos-dev/U_BcylyY_kw`, 2015, accessed: 11 April 2015.

[56] Cisco. Cisco Global Cloud Index: Forecast and Methodology 2013–2018 White Paper. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.html`, 2014, accessed: 11 April 2015.

[57] O'Brien, S. 80/20. `https://learningnetwork.cisco.com/thread/25487`, 2014, accessed: 11 April 2015.

[58] Sourceforge. Iperf. `http://sourceforge.net/projects/iperf/`, 2013, accessed: 11 April 2015.

[59] Vondrous, O. FlowPing - UDP based ping application. `http://flowping.comtel.cz/`, 2012, accessed: 11 April 2015.

Appendix A

Acronyms

AMD Advanced Micro Devices

API Application Programming Interface

APU Accelerated Processing Unit

ARP Address Resolution Protocol

BGP Border Gateway Protocol

BPDU Bridge Protocol Data Unit

BUM Broadcast, Unknown unicast and Multicast

CAM Content Addressable Memory

CDP Cisco Discovery Protocol

CLI Command Line Interface

CPU Central Processing Unit

DB Database

DC Data Center

DHCP Dynamic Host Configuration Protocol

DSCP Differentiated Services Code Point

ESX (VMware's hypervisor) Elastic Sky X

FIB Forwarding Information Base

FTP File Transfer Protocol

GRE Generic Routing Encapsulation

HA High Availability

HP Hewlett-Packard

HW Hardware

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IGMP Internet Group Management Protocol

IOS (Cisco's) Internetwork Operating System

IP Internet Protocol

IT Information Technology

KVM Kernel-based Virtual Machine

LACP Link Aggregation Control Protocol

LLDP Link Layer Discovery Protocol

MAC Media Access Control

NFV Network Functions Virtualisation

NIC Network Interface Card

ONOS Open Network Operating System

OSGi Open Services Gateway initiative

OSPF Open Shortest Path First

OVS Open vSwitch

OVSDB Open vSwitch Database

PAgP Port Aggregation Protocol

QEMU Quick Emulator

RARP Reverse Address Resolution Protocol

REST Representational State Transfer

RFC Request For Comments

RPF Reverse Path Forwarding

SDN Software Defined Networking

SNMP	Simple Network Management Protocol
SPoF	Single Point of Failure
SQL	Structured Query Language
STP	Spanning Tree Protocol
STT	Stateless Transport Tunnelling
SVI	Switch Virtual Interface
SW	Software
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
ToR	Top of Rack (switch)
TTL	Time To Live
UDP	User Datagram Protocol
vDS	(VMware's) VNetwork Distributed Switch
VEM	(Nexus 1000V's) Virtual Ethernet Module
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNI	VXLAN Network Identifier
vNIC	Virtual Network Interface Card
VoIP	Voice over Internet Protocol
VSM	(Nexus 1000V's) Virtual Supervisor Module
VTEP	VXLAN Tunnel End Point
VXLAN	Virtual Extensible Local Area Network
XML-RPC	Extensible Markup Language Remote Procedure Call
XMPP	Extensible Messaging and Presence Protocol

Appendix **B**

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
src ....................................... the directory of source codes
  bash ..................................... includes qemu hook script
  java ............................ includes all Java-based source code
    control-application ..... contains the control application source
    host-provider-service ..... contains HostProviderService source
  latex .............. the directory of LaTeX source codes of the thesis
text ........................................ the thesis text directory
  thesis.pdf .......................... the thesis text in PDF format
  thesis.ps ............................ the thesis text in PS format
```