Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Virtual Routers in cloud infrastructure

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΝΤΩΝΙΟΣ ΜΑΝΟΥΣΗΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Μάρτιος 2015

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Virtual Routers in cloud infrastructure

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΝΤΩΝΙΟΣ ΜΑΝΟΥΣΗΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30η Μαρτίου 2015.

Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Βασίλειος Μάγκλαρης Καθηγητής Ε.Μ.Π.	Ευσταθιος Συκάς Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2015

Αντώνιος Μανούσης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Contents

List of Figures

List of Tables

Περίληψη

Σκοπός της παρούσας εργασίας ειναι να διερευνήσουμε διαφορετικούς τρόπους ώστε να υλοποιήσουμε εικονικούς δρομολογητές και να τους ενσωματώσουμε στην δικτυακή υποδομή του Synnefo, παρέχοντας έτσι μια επιπλέον λειτουργικότητα στους χρήστες της πλατφόρμας.

Σύμφωνα με την ορολογία του OpenStack ένας εικονικός δρομολογήτης είναι η οντότητα που επιτρέπει την προώθηση δικτυακών πακέτων αφενός μεταξύ εσωτερικών απομονωμένων υποδικτύων και αφετερου προς το διαδίκτυο χρησιμοποιώντας Network Address Translation (NAT). Στη παρούσα διπλωματική εργασία σχεδιάσαμε και υλοποιήσαμε δυο αρχιτεκτονικές και ενσωματώσαμε μια από αυτές στην υποδομή του Synnefo.

Η πρώτη υλοποίηση επιτρέπει στο χρήστη τη δημιουργία δρομολογητών που χαρακτηρίζονται από υψηλή διαθεσιμότητα χρησιμοποιώντας ένα σύνολο εργαλείων και πρωτοκόλλων, όπως το Virtual Router Redundancy Protocol και τα conntrack tools. Το VRRP είναι ένα πρωτόκολλο το οποίο δίνει τη δυνατότητα σε δυο διεπαφές που βρίσκονται σε διαφορετικά φυσικά μηχανήματα να μοιραστούν την ίδια διεύθυνση IP με σκοπό η IP να είναι προσβάσιμη ακόμα κι αν το μηχάνημα που την εξυπηρετούσε αρχικά καταστεί μη διαθέσιμο. Τα conntrack tools είναι ένα σύνολο από εργαλεία που επιτρέπουν το συγχρονισμο των δυο μηχανημάτων που μοιράζονται την ίδια IP ώστε σε περίπτωση που ένα μηχάνημα κληθεί αιφνιδιαστικά να εξυπηρετήσει κίνηση που μέχρι πρότινως εξυπηρετούνταν από κάποιο άλλο μηχάνημα να γνωρίζει τις ήδη εγκατεστημένες δικτυακές συνδέσεις ώστε να συνεχίσει να τις εξυπηρετεί.

Η δεύτερη υλοποίηση είναι μια αρχιτεκτόνική βασισμένη στις αρχές του Software Defined Networking (SDN) και του Network Functions Virtualization (NFV) και δίνει στο χρήστη τη δυνατότητα να δημιουργήσει εικονικους δρομολογητές ενσωματώνοντας την απαραίτητη δικτυακή λογική που πρέπει να υλοποιεί ο δρομολογητής σε έναν SDN έλεγκτη. Στα πλαίσια αυτής της εργασίας υλοποίησαμε τη λογική για NAT, host tracking καθώς και έναν messenger ο οποίος δινει τη δυνατότητα αποκεντρωμένης επικοινωνίας μεταξύ των Κυκλάδων (δικτυακή υποδομή Synnefo) και των εικονικών δρομολογητών.

Λέξεις κλειδιά

Δικτυα Υπολογιστών, Δρομολογητές, Υψηλη Διαθεσιμότητα, OpenFlow

Abstract

The purpose of this diploma dissertation is to research ways to implement virtual routers and and integrate them into Synnefo, thus adding a new network feature to the platform.

A virtual router as a term according to OpenStack standards is a logical entity that forwards packets across internal subnets and NATs them on external networks through an appropriate external gateway. It has an interface for each subnet with which is associated and the ip address of these interfaces is the subnet's gateway IP. The need to create one such feature stems from the fact that many providers can only afford to give one public routable IP to every user. We have designed two solutions, evaluated them and then integrated into Synnefo one of the two.

The first one allows us to create a highly available virtual router using a set of tools and protocols such as the Virtual Router Redundancy Protocol and conntrack tools. VRRP is the protocol which allows two interfaces on two different machines to share the same IP in an ACTIVE/ACTIVE or ACTIVE/-PASSIVE configuration so that the IP is reachable should one of the machines fail, thus creating small HA clusters. Conntrack tools is a set of tools that provide synchronization of the connection tracking kernel tables of two machines that are supposed to share the same IP using a protocol like VRRP so that when a transition occurs the new machine will have information about the incoming established connections and hence will not drop them.

The second implementation is a SDN based solution that allows us to create a virtual router by incorporating the required network logic in a set of controller components. For the purposes of this project we used the POX controller and developed components that allow for NAT, host tracking as well as a messenger that provides decentralised communication over TCP between the virtual router and the Network Service of the cloud stack, namely Cyclades. This implementation provides a versatile base to easily incorporate more features onto the virtual routers such as custom firewalling per subnet or network discovery.

Key words

Computer Networks, Network Address Translation, Software Defined Networks, Openflow.

Ευχαριστίες

Με την παρούσα διπλωματική κλείνει ένας πενταετής κύκλος σπουδών στο ΕΜΠ και ανοίγει ο δρόμος για τα επόμενα ακαδημαικά μου βήματα. Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα της εργασίας αυτής, κ. Νεκτάριο Κοζύρη, για τη συνεχή καθοδήγηση του ήδη από το τρίτο έτος των σπουδών μου αλλά και για την εμπιστοσύνη που μου έδειξε δίνοντας μου τη δυνατότητα να εκπονήσω τη διπλωματική μου εργασία στο Synnefo. Επίσης θα ήθελα να ευχαριστήσω τον Δρα. Βαγγέλη Κούκη καθώς και τον Δρα. Χρήστο Αργυρόπουλο, οι οποίοι μεσα από τις εποικοδομητικές συζητήσεις μας, μου έδωσαν την έμπνευση να ασχοληθώ με το παρόν θέμα αλλά και τα εφόδια για να το αντιμετωπίσω επιτυχώς. Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένεια και τους φίλους μου, οι οποίοι στάθηκαν δίπλα μου και μου έδωσαν τη πολύτιμη υποστήριξη τους σε κάθε βήμα αυτής της προσπάθειας.

Αντώνιος Μανούσης,

Αθήνα, 30η Μαρτίου 2015

Chapter 1

Introduction

Over the past years, many IT professionals, business managers, and researchers have started to talk about a new phenomenon called cloud computing [14]. Each of these groups defined cloud computing differently according to their understanding of its offerings. Although there was no agreement about what precisely constituted cloud computing, it still offered a promising paradigm that could enable businesses to face market volatility in an agile and cost-efficient manner.

However, a concise definition of cloud computing has emerged recently: To outsource IT activities to one or more third parties that have rich pools of resources to meet organization needs easily and efficiently. These needs may include hardware components, networking, storage, and software systems and applications. In addition, they may include infrastructure items such as physical space, cooling equipments, electricity, fire fighting systems, and human resources to maintain all those items. In this model, users are billed for their usage of remote IT infrastructures rather than buying, installing, and managing them inside their own datacenters. This structure gives users the flexibility to scale up and down in response to market fluctuations. For instance, a business enters the market with a new website that is initially unknown to customers, but eventually becomes popular with hundreds of thousands of requests per day. With cloud computing, businesses may start with a minimum set of IT resources and allocate additional services during peak times. Moreover, website owners can easily dispose unused IT resources during non-peak/recession times enabling them to reduce overall costs.

Cloud computing providers offer their services according to several fundamental models,one of which is known as IaaS (Infrastructure as a Service) [24]. Specifically an IaaS cloud is the most basic cloud-service model according to the IETF (Internet Engineering Task Force). Providers of IaaS offer computers – physical or (more often) virtual machines – and other resources. IaaS clouds often offer additional resources such as a virtual-machine disk image library, raw block storage, and file or object storage, firewalls, load balancers, IP addresses, virtual local area networks (VLANs), and software bundles. IaaS-cloud providers supply these resources on-demand from their large pools installed in datacenters. For wide-area connectivity, customers can use either the Internet or carrier clouds (dedicated virtual private networks).

The National Technical University of Athens (NTUA) along with the Greek Research and Development Network (GRNET) have taken a leading role in research relating with cloud computing by building an IaaS cloud stack, namely Synnefo[16]. The purpose of this thesis is to contribute to the development of Synnefo by proposing and implementing novel features for its Network service.

1.1 Thesis motivation

The motivation for this thesis was the result of the effort to improve the Network Service of the Synnefo cloud software which powers the okeanos [15] public cloud service. A brief description of Synnefo and okeanos can be found below.

Synnefo is an open source cloud stack written in Python using the Django framework. It lies on top of Google's open source cluster management tool, namely Ganeti and provides services similar to the ones offered by Amazon Web Services (AWS). In order to boost third party compatibility, Synnefo exposes Openstack APIs, such as Swift, Nova and Neutron.

Synnefo implements the following services:

- *Compute Service*, which enables the creation and management of the Virtual Machines.
- *Network Service*, which is the service that provides network management, creation and transparent support of various network configurations.
- *Image Service*, which handles the customization and the deployment of OS images.
- *Volume Service*, which provides the necessary operations for volume management
- *Storage Service*,which is the service responsible for provisioning the VM volumes and storing user data.

okeanos is an IaaS that provides Virtual Machines, Virtual Networks and Storage services to the Greek Academic and Research community. It is an open-source service that has been running in production servers since 2011 by GRNET S.A.

Synnefo allows each user to create and manage multiple Virtual Machines(VM) and Virtual Private Networks. However, given the shortage of public routable IPv4 addresses it is not always guaranteed that a provider running Synnefo will have the resources to provide one public IP per VM. Consequently, this has created the need to design and implement a logical entity that forwards packets across internal private networks and also performs Network Address Translation (NAT) to external public networks through an appropriate external gateway. This entity, in accordance with Openstack Neutron standards is called router, even though it does not implement dynamic routing protocols, such as OSPF or BGP.

In this thesis, we explore various tools and architectures that could be used in order to implement virtual routers. Eventually, we design and implement an architecture based on the principles of Software Defined Networks using the Openflow protocol. The novelty of this thesis consists in the development of the required network logic and its integration into the Synnefo infrastructure.

1.2 Thesis structure

This thesis is structured as follows:

Chapter 2

In this chapter we provide the necessary background so that the reader can familiarize himself with the technical and theoretical concepts concerning the thesis. We present current approaches to the problem, their advantages and disadvantages and how they have affected our implementation of virtual routers.

Chapter 3

This chapter proposes an architecture for Highly Available virtual routers using legacy protocols and tools that are offered by two different operating systems, namely OpenBSD and Linux. We present the architecture, some implementation specifics as well as shortcomings of this approach with regards to Synnefo.

Chapter 4

In this chapter which is the core of this thesis, we explore the possibility of implementing virtual routers using the concept of Software Defined Networks. We show that if we integrate SDN and Openflow in the Synnefo infrastructure, we can design and implement a versatile architecture that seamlessly integrates virtual routers with minimal modifications in the existing networking service. This architecture is in accordance with the requirements set by the respective RFCs and follows the standards set by Openstack Neutron.

Chapter 5

The scope of this chapter is to present an evaluation of the architectures proposed so far in this thesis and provide a demonstration of the capabilities of the SDN based router.

Chapter 6

In this chapter, we present our concluding remarks as well as propositions for future work on this project.

Chapter 2

Theoretical Background

In this chapter we provide the necessary background to familiarize the reader with the main concepts used later in the document. We begin by providing an overview of the Network Address Translation mechanism and continue by describing briefly the tools that have been used in each of the designs and architectures that have been examined. The approach provided is rudimentary and only intends to familiarize the reader with the required tools for the creation of virtual routers.

2.1 Cyclades Networking

Cyclades is the component that is responsible for the Compute, Network and Image Services of Synnefo. It is the action orchestrator and the API layer on top of multiple Ganeti clusters. As far as Networking is concerned, Cyclades allows for a plethora of different deployments and configurations [6]. More details about how Cyclades handle Virtual Networks are presented below.

- *PHYSICAL_VLAN* implementation allows for L2 isolation which is ensured with one dedicated physical VLAN per network. Currently, the administrator must pre-provision one physical VLAN tag and a dedicated bridge per network.
- *MAC_FILTERED*. Since it is practically impossible to provide a single VLAN for each private network across the whole data center, L2 isolation can also be achieved via MAC fitlering on a single bridge and physical VLAN using ebtables. The infrastructure provides a physical VLAN or separate interface shared among hosts in the cluster. All virtual interfaces are bridged on a common bridge and the concept is that filtering will be done with ebtables and MAC prefix. Therefore, all interfaces within a L2 network have the same MAC prefix. MAC prefix uniqueness among private networks is guaranteed by Synnefo.
- *IP_LESS_ROUTED*. In this setup each host does not have an IP inside the routed network and does Proxy Arp for the host machines belonging in the virtual networks with IP_LESS_-ROUTED flavor.

2.2 Network Address Translation

Network Address Translation (NAT), was originally presented in RFC 1631[10] in May 1994. NAT was designed as an interim approach to the problem of IPv4 address depletion by defining a set of IP addresses that could be shared or reused by many hosts. These addresses are allocated for private use within a home or office network and they have a meaning only for the devices within that subnet, thus creating a realm of private addresses that are transparent to the larger global Internet. Communication with the globally routable Internet relies on a NAT-enabled router which uses (at least) one global IP

address and a translation table to map private addresses to globally routable addresses and forward the modified packets to the Internet.

For the purposes of this thesis only the case of Network Address Port Translation (NAPT) will be analyzed and the terms NAT and NAPT will be used interchangeably. Further analysis of the technical characteristics of NAT can be found in RFC 2663 [23] and RFC 3022 [11]. In the case of NAPT, instead of using multiple public IP addresses, NAPT uses one global address and distinguishes the sessions being mapped by it based upon a source and destination port number. When a packet from the private realm reaches the NAT, the router replaces the private source address and port number with a global address and a port number that is not currently in its translation table and then forwards the packet to the Internet. Since port numbers are 16 bits long, this implies that the NAT can handle more than 60000 simultaneous connections for a given transport protocol (UDP, TCP). According to RFC 5382 [13] a device that implements Network Address Translation must follow specific guidelines for TCP/UDP and ICMP:

Regarding the main transport protocols TCP and UDP as well as ICMP the requirements are described below:

- *A NAT MUST have an "Endpoint-Independent Mapping" behavior for TCP.*
  Endpoint-Independent Mapping behavior permits the consistent work of peer-to-peer applications without jeopardizing the security features and benefits of Network Address Translation. It allows applications to learn and advertise the external IP address and port allocated to an internal endpoint so that external peers can contact it.

- *A NAT MUST support all valid sequences of TCP packets (defined in [RFC0793]) for connections initiated both internally as well as externally when the connection is permitted by the NAT. In particular in addition to handling the TCP 3-way handshake mode of connection initiation, A NAT MUST handle the TCP simultaneous- open mode of connection initiation.*
  The purpose of this requirement is to allow TCP stacks that are compliant with the standards to traverse NATs no matter which end initiates the connection as long as the connection is permitted by the NAT policies. Specifically, in addition, to TCP packets for 3-way handshake, a NAT must be prepared to accept an inbound SYN and an outbound SYN-ACK for an internally initiated connection in order to support simultaneous-open.

- *If application transparency is most important, it is RECOMMENDED that a NAT have an "Endpoint-Independent Filtering" behavior for TCP. If a more stringent filtering behavior is most important, it is RECOMMENDED that a NAT have an "Address-Dependent Filtering" behavior. a) The filtering behavior MAY be an option configurable by the administrator of the NAT. b) The filtering behavior for TCP MAY be independent of the filtering behavior for UDP.*
  Let us consider an internal IP address and port (X:x) that is assigned (or reuses) a mapping (X1':x1') when it initiates a connection to an external (Y1:y1). An external endpoint (Y2:y2) attempts to initiate a connection with the internal endpoint by sending a SYN to (X1':x1'). If a NAT allows the connection initiation from all (Y2:y2), then it is defined to have "Endpoint-Independent Filtering" behavior. If the NAT allows connection initiations only when Y2 equals Y1, then the NAT is defined to have "Address- Dependent Filtering" behavior.

- *A NAT MUST NOT respond to an unsolicited inbound SYN packet for at least 6 seconds after the packet is received. If during this interval the NAT receives and translates an outbound SYN for the connection the NAT MUST silently drop the original unsolicited inbound SYN packet. Otherwise, the NAT SHOULD send an ICMP Port Unreachable error (Type 3, Code 3) for the original SYN, unless REQ-4a applies. a) The NAT MUST silently drop the original SYN packet if sending a response violates the security policy of the NAT.*

This requirement is to allows simultaneous-open to work reliably in the presence of NATs as well as to quickly signal an error in case the unsolicited SYN is in error.

- *If a NAT cannot determine whether the endpoints of a TCP connection are active, it MAY abandon the session if it has been idle for some time. In such cases, the value of the "established connection idle-timeout" MUST NOT be less than 2 hours 4 minutes. The value of the "transitory connection idle-timeout" MUST NOT be less than 4 minutes. a) The value of the NAT idle-timeouts MAY be configurable.*
  While some NATs may choose to abandon sessions reactively in response to new connection initiations (allowing idle connections to stay up indefinitely in the absence of new initiations), other NATs may choose to proactively reap idle sessions. This requirement provides the necessary conditions for such a policy.
- *If a NAT includes ALGs that affect TCP, it is RECOMMENDED that all of those ALGs (except for FTP [21]) be disabled by default.*
- *A NAT MUST NOT have a "Port assignment" behavior of "Port overloading" for TCP.*
  This requirement allows two applications on the internal side of the NAT to consistently communicate with the same destination.
- *A NAT MUST support "hairpinning" for TCP. a) A NAT's hairpinning behavior MUST be of type "External source IP address and port".*
- *If a NAT translates TCP, it SHOULD translate ICMP Destination Unreachable (Type 3) messages.*
  The translation of ICMP Destination Unreachable messages helps avoiding communication failures, namely 'black holes' [17]. Most importantly, it is important to translate the 'Fragmentation Needed and Don't Fragment was Set' (Type 3, Code 4) message.
- *Receipt of any sort of ICMP message MUST NOT terminate the NAT mapping or TCP connection for which the ICMP was generated.*
  This is necessary for reliably performing TCP simultaneous-open where a remote NAT may temporarily signal an ICMP error.

2.3 The Netfilter Framework

The Netfilter[22] framework consists of a set of hooks over the Linux network protocol stack. An administrator can register kernel modules upon these hooks that are responsible for performing some sort of network packet handling at various stages of the network stack.The Netfilter framework inserts five hooks in the network protocol stack in order to perform packet handling at different stages. These are the following:

- *PREROUTING*. This is the hook that all packets reach without exception. At this stage, the kernel performs sanity checks on the packet headers and performs specific functions such as Destination Network Address Translation (DNAT).
- *LOCAL INPUT*. The packets that are destined for the local machine reach this hook. It is the last hook in the chain of hooks that the traffic destined for the local machine encounters.
- *FORWARDING*. The packets that will reach this hook are those not destined for the local machine and need to be forwarded (if the machine has forwarding capabilities). Most firewalling operations are implemented at this stage.

- *LOCAL OUTPUT*. This is the first hook on the egress path of the local machine. All outgoing packets from processes in the local machine reach this hook.

- *POSTROUTING*. This hook is reached after the routing decision has been made and is responsible for implementing Source Network Address Translation (SNAT). All outgoing packets reach this hook.

Since the scope of this thesis is to investigate machines that are supposed to forward traffic we can categorize the traffic reaching them in three groups, depending on the destination.

- *Traffic destined for the router.*
  Given the definitions above for the netfilter hooks, this traffic hits the PREROUTING hook first and then continues to INPUT.

- *Traffic to be forwarded by the router.*
  This kind of traffic follows the PREROUTING, FORWARDING, POSTROUTING path.

- *Traffic created by the router.*
  This traffic first hits the OUTPUT and then the POSTROUTING hooks.

The netfilter module allows the administrator to register one or more callbacks to each of the hooks mentioned above. The callbacks allow for a broad set of actions such as letting a packet continue travelling through the stack, dropping or mangling the packet. In the following sections we present two common applications that sit on top of the Netfilter framework and have proven useful throughout the development of this thesis.

2.3.1 IPtables

IPtables[2] is a user-space application that allows a system administration to filter network traffic. IPtables is usually confused with the Netfilter framework itself as it utilizes the same name for its chains. However, it is merely a tool that sits on top of Netfilter and takes advantage of the abilities that the latter offers.

IPtables allows for the creation of tables containing chains of rules for treating packets and every kind of packet processing refers to a specific table. Every packet is processed by sequentially traversing the rules of the chains that handle this kind of packet and traverses at least one chain. There are five predefined chains, with every chain being a module registered on each of the five Netfilter hooks. A table may not have all chains.

Each rule within a chain specifies the characteristics of the packets it matches and the actions to be performed should one such packet is received. As a packet traverses a chain each rule is examined. If the rule does not match the packet, the packet is processed by the next rule. If a rule matches, the action described by that rule is taken and this may result to the packet being accepted and allowed to continue along the chain or dropped. The conditions upon which a rule matches a packet vary and correspond to almost every layer in the OSI model, e.g the –mac-source and –dport parameters. There are also protocol-independent parameters such as -m time.

2.3.2 The Connection Tracking system

IPtables provides a way to filter packets based on the contents of the packet header. However, filtering based only on this has proven to provide insufficient protection against various kinds of attacks such as DOS attacks. Therefore stateful filtering has been added and conntrack is practically one more application sitting on top of Netfilter.

The connection tracking system [3], in a nutshell, stores all the information that define a connection between two endpoints, such as source and destination IP addresses, port numbers and protocols. This allows for intelligent packet filtering based on the state of each connection that is saved in the connection table of the machine that forwards traffic (router or firewall). It is worth noting that this system does not filter packets and by default lets them pass without stopping them. Of course there are exceptions to this rule such as when there is memory overflow and the kernel table cannot hold any more connections.

The connection tracking system allows for four states when saving a connection.

- *NEW*. The connection is starting. The router or the firewall has seen traffic only in one direction, e.g a SYN packet.
- *ESTABLISHED*. The connection is established and there is two way communication.
- *RELATED*. This is an expected connection that is directly related to the protocol being used. For example in the case of FTP there is a connection establishing and controlling the communication between two endpoints and a second connection, related to the control one responsible for the file transfer.
- *INVALID*. Special state for packets that do not follow the normal or expected behavior of the connection they belong to.

Keeping state of a connection is independent of the protocol used. In this analysis for instance a connection between two endpoints, completed using the UDP protocol, a stateless protocol, is stateful.

2.4 Software Defined Networking - Openflow

In a nutshell, Software Defined Networking (SDN) [12] is a networking approach that allows the efficient decoupling of the control and data plane of a network, thus enabling enhanced control and wide visibility over the network. Openflow is the first standard communications interface defined between those two layers of the SDN architecture. More information regarding the SDN architecture and the Openflow protocol are described below.

2.4.1 Data plane

Typically, the data plane is locally implemented within each network device and operates based on the line-rate. The data plane refers to the underlying systems, which forward a packet to a selected destination. Said another way, the data plane is responsible for the process of packet forwarding based on forwarding rules and for the simultaneous check of Access Control Lists (ACLs). Moreover, queue

management and packet scheduling are implemented in the context of the data plane. All operations mentioned above are based on hardware components.

Despite the fact that data plane implementations vary among vendors, network devices communicate with each other via standardized data forwarding protocols, such as Ethernet.

2.4.2 Control plane

The control plane is mainly responsible for system configuration and for carrying signaling traffic and exchange routing table information. The control plane feeds the data plane and thus the data plane's functionality is defined by the rules imposed by the control plane. In legacy networks the signaling traffic is in-band and he control plane refers to the router component that focuses on the way that a given device interacts with its neighbours via state exchange. One of the main control plane operations is to combine routing information in order to populate the Forwarding Information Base (FIB), which is used by the data plane. The functionality of the control plane can be either centralized or distributed. In the case of a centralized control plane, decisions are taken from a central process whereas in the case of a distributed network the decisions are taken locally in the devices of the network that implement control plane algorithms.

2.4.3 SDN architecture

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

- *Directly programmable*: Network control is directly programmable because it is decoupled from forwarding functions.
- *Agile*: Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs.
- *Centrally managed*: Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.
- *Programmatically configured*: SDN lets network managers configure, manage, secure, and optimize network resources very quickly via dynamic, automated SDN programs, which they can write themselves because the programs do not depend on proprietary software.
- *Open standards-based and vendor-neutral*: When implemented through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

SDN addresses the fact that the static architecture of conventional networks is ill-suited to the dynamic computing and storage needs of today's data centers, campuses, and carrier environments. The key computing trends driving the need for a new network paradigm include:

- *Changing traffic patterns*: Applications that commonly access geographically distributed databases and servers through public and private clouds require extremely flexible traffic management and access to bandwidth on demand.
- *The "consumerization of IT"*: The Bring Your Own Device (BYOD) trend requires networks that are both flexible and secure.
- *The rise of cloud services*: Users expect on-demand access to applications, infrastructure, and other IT resources.
- *"Big data" means more bandwidth*: Handling today's mega datasets requires massive parallel processing that is fueling a constant demand for additional capacity and any-to-any connectivity.

In trying to meet the networking requirements posed by evolving computing trends, network designers find themselves constrained by the limitations of current networks:

- *Complexity that leads to stasis*: Adding or moving devices and implementing network-wide policies are complex, time-consuming, and primarily manual endeavors that risk service disruption, discouraging network changes.
- *Inability to scale*: The time-honored approach of link oversubscription to provision scalability is not effective with the dynamic traffic patterns in virtualized networks—a problem that is even more pronounced in service provider networks with large-scale parallel processing algorithms and associated datasets across an entire computing pool.
- *Vendor dependence*: Lengthy vendor equipment product cycles and a lack of standard, open interfaces limit the ability of network operators to tailor the network to their individual environments.

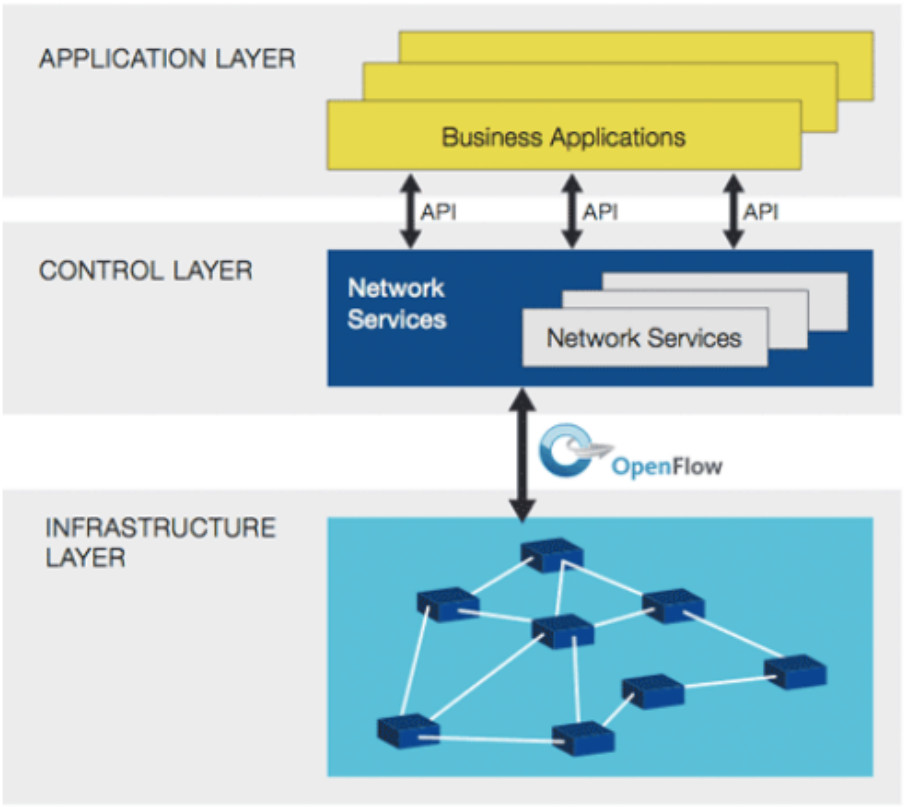

Figure 2.1: Software Defined Netowkrs architecture

2.4.4 Openflow

OpenFlow[18] is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. OpenFlow allows direct access to and manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual (hypervisor-based).

OpenFlow-based SDN technologies enable IT to address the high-bandwidth, dynamic nature of today's applications, adapt the network to ever-changing business needs, and significantly reduce operations and management complexity.

Openflow is principally characterized by the following features

- **Programmability**
  - *Enable innovation/differentiation*
  - *Accelerate new features and services introduction*
- **Centralized Intelligence**
  - *Simplify provisioning*
  - *Optimize performance*
  - *Granular policy management*
- **Abstraction**
  - Decouple:
    - *Hardware & Software*
    - *Control plane & Data plane*
    - *Physical & logical configuration*

2.5 Network Functions Virtualization

Network functions virtualization (NFV) [4] (also known as virtual network function (VNF)) offers a new way to design, deploy and manage networking services. NFV decouples the network functions, such as network address translation (NAT), firewalling, intrusion detection, domain name service (DNS), and caching, to name a few, from proprietary hardware appliances so they can run in software.

It's designed to consolidate and deliver the networking components needed to support a fully virtualized infrastructure – including virtual servers, storage, and even other networks. It utilizes standard IT virtualization technologies that run on high-volume service, switch and storage hardware to virtualize network functions. It is applicable to any data plane processing or control plane function in both wired and wireless network infrastructures.

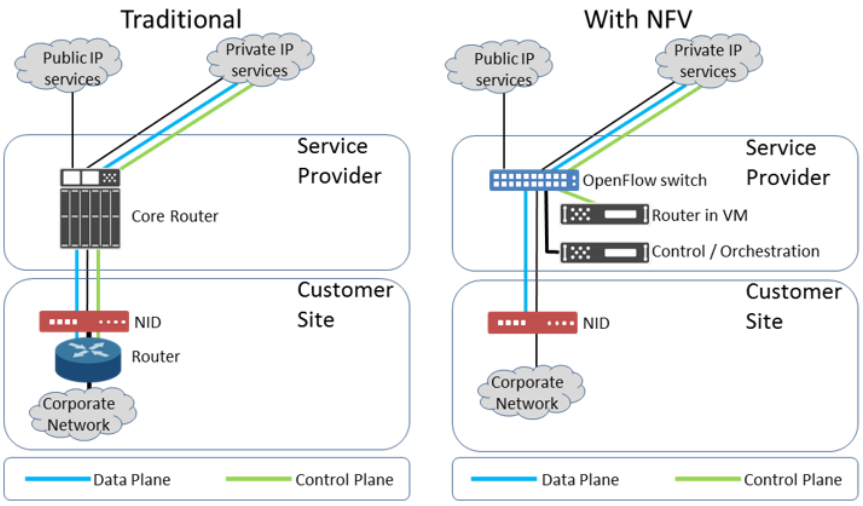

Figure 2.2: Network Functions Virtualization architecture

History of NFV

The concept for NFV originated from service providers who were looking to accelerate the deployment of new network services to support their revenue and growth objectives. The constraints of hardware-based appliances led them to applying standard IT virtualization technologies to their networks. To accelerate progress towards this common goal, several providers came together and created the European Telecommunications Standards Institute (ETSI). The ETSI Industry Specification Group for Network Functions Virtualization (ETSI ISG NFV), a group charged with developing requirements and architecture for virtualization for various functions within telecoms networks, such as standards like NFV MANO. ETSI is also instrumental in collaborative projects like the newly announced OPNFV.

The Benefits of NFV

NFV virtualizes network services via software to enable operators to:

- *Reduce CapEx*: reducing the need to purchase purpose-built hardware and supporting pay-as-you-grow models to eliminate wasteful overprovisioning.

- *Reduce OpEX*: reducing space, power and cooling requirements of equipment and simplifying the roll out and management of network services.

- *Accelerate Time-to-Market* : reducing the time to deploy new networking services to support changing business requirements, seize new market opportunities and improve return on investment of new services. Also lowers the risks associated with rolling out new services, allowing providers to easily trial and evolve services to determine what best meets the needs of customers. *Deliver Agility and Flexibility*: quickly scale up or down services to address changing demands; support innovation by enabling services to be delivered via software on any industry-standard server hardware.

Chapter 3

Highly available virtual routers using legacy protocols

As mentioned before, the motivation for this thesis was to design and implement an entity that would hold a user's public IP address and would forward traffic from the user's private isolated networks to the Internet. This entity is called router. It is easy to understand that a router can potentially be a single point of failure for the user's networking service and therefore it would be desirable if it could be characterized by high availability. The scope of this chapter is to describe an architecture that achieves this goal, and present two different implementations for this architecture.

3.1 Network Layout

The concept of high availability inherently assumes that when part of the infrastructure loses its functionality, there will be a spare component ready to take over and thus prevent a service from being compromised. In this context, the first step towards the development of a highly available virtual router is to understand that normally there will be two instances of the router running, without necessarily making this information available to the end user who asks for and therefore only sees a HA router. This reasoning leads to the following network layout:

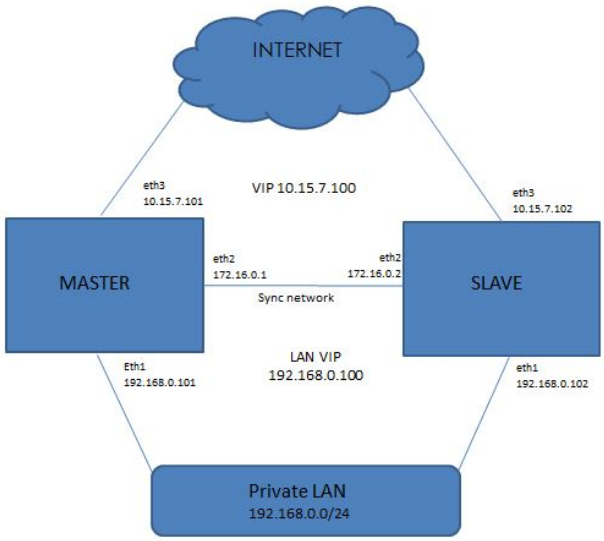

Figure 3.1: Network Layout with HA virtual router.

This layout represents a case of an "ACTIVE-PASSIVE" configuration where at any time one of the router instances is active and serving the network traffic and the second one is idle, a "hot standby"

waiting to take over should the first one fails. To better understand this layout there are a few questions that need to be answered;

- How can an IP address simultaneously 'belong' to more than one entities, since both need to be able at any given time to forward traffic?
- How can an IP address simultaneously match to more than one MAC addresses given that two network cards cannot have the same MAC address?
- How can two distinct entities (in our case VMs) have their kernel session tables synchronized so that when a failover occurs the established connections continue to be served by the HA router?

It can be easily understood that each one of the HA router entities needs to have at least 2 Network Interface Cards (NICs); one for the private network whose traffic it is responsible to forward, and one for the public network. The third interface that can be seen above belongs to a dedicated link that is used in order for each entity to be able to potentially exchange session table synchronization messages with its peer. It could be possible to use one of the other two interfaces in order to achieve the exchange of state information, however for reasons that will be explained further on we have decided to transmit them over this a third link.

3.2 Implementation

In order to provide an efficient implementation of the architecture described above and answer the questions that this architecture poses we propose two implementations. Their logic is similar, however each implementation is based on tools provided by two different operating systems, namely OpenBSD and Linux. In the following subsections we describe each of the two implementations in detail.

3.2.1 OpenBSD - Carp

The CARP protocol

CARP[7] stands for Common Address Redundancy Protocol and is a protocol available on hosts running OpenBSD that enhances system redundancy by allowing two hosts in the same network segment to form a 'redundancy' group and share an IP Address. This means that when one of the machines that participates in this cluster goes down, its tasks are taken over by another machine that belongs in the group. CARP is a protocol that ensures service continuity and possibly load balancing to a plethora of network services and not only to firewalls or routers. Regarding the motivation behind the creation of this protocol, it is worth mentioning that even though the OpenBSD team originally wanted to produce a free implementation of VRRP(Virtual Router Redundancy Protocol) [RFC3768] and HSRP (Hot Standby Router Protocol) [RFC2281] patent rights forced the developers to create a new competing protocol designed to be fundamentally different from VRRP.

CARP is considered to be a multicast protocol as every redundancy group constitutes a multicast group. Every group is characterized by one or more virtual addresses and in an 'ACTIVE-PASSIVE' configuration at any given time one member of the group is the master and responds to all packets destined for the group. The other members are called backups and wait to take the master's place. Once a member of the group assumes the master position it advertises his state at regular, configurable intervals using the 112 IP protocol number. These messages are the heartbeats that determine whether the master is online or not. If these messages stop, the backup nodes begin to advertise each

one at his own interval. The node that advertises his presence most frequently will eventually become the new master node of the group. If/When the old master comes back up, it will become a backup host, but it is also possible that it will try to become master again.

The ifconfig(8) manpage on OpenBSD provides detailed explanation of the different parameters of CARP. These details can be found below:

```
ifconfig carp-interface [advbase n] [advskew n] [balancing mode]
[carpnodes vhid:advskew,vhid:advskew,...]
[iface][[-]carppeer peer_address]
[passphrase] [state state][vhid host-id]
```

`advbase n`
Set the base advertisement interval to n seconds. Acceptable values are 0 to 254; the default value is 1 second.

`advskew n`
Skew the advertisement interval by n. Acceptable values are 0 to 254; the default value is 0.

`carpdev iface`
Attach to iface. If not specified, the kernel will attempt to select an interface with a subnet matching that of the carp interface.

`carppeer peer_address`
Send the carp advertisements to a specified point-to-point peer or multicast group instead of sending the messages to the default carp multicast group. The peer_address is the IP address of the other host taking part in the carp cluster. With this option, carp(4) traffic can be protected using ipsec(4) and it may be desired in networks that do not allow or have problems with IPv4 multicast traffic.

`vhid n`
Set the virtual host ID to n. Acceptable values are 1 to 255. Taken together, the advbase and advskew indicate how frequently, in seconds, the host will advertise the fact that it considers itself master of the virtual host. The formula is `advbase` + (`advskew` / 256). If the master does not advertise within three times this interval, this host will begin advertising as master.

OpenBSD's Packet Filter

Packet Filter (PF) is a module in the OpenBSD kernel that enables filtering of TCP/IP traffic and Network Address Translation. It provides bandwidth control and packet prioritization by normalizing and applying specific conditions to TCP/IP traffic. PF has been a part of the OpenBSD kernel since verion 3.0 whereas previous versions used different NAT[8] modules that are no longer used.

According to the man pages packet filtering takes place in the kernel. A pseudo-device, /dev/pf, allows userland processes to control the behavior of the packet filter through an ioctl(2) interface. There are commands to enable and disable the filter, load rulesets, add and remove individual rules or state table

entries, and retrieve statistics. The most commonly used functions are covered by pfctl(8).

Manipulations like loading a ruleset that involve more than a single ioctl(2) call require a so-called ticket, which prevents the occurrence of multiple concurrent manipulations. Fields of ioctl(2) parameter structures that refer to packet data (like addresses and ports) are generally expected in network byte-order.

Rules and address tables are contained in so-called anchors. When servicing an ioctl(2) request, if the anchor field of the argument structure is empty, the kernel will use the default anchor (i.e., the main ruleset) in operations. Anchors are specified by name and may be nested, with components separated by '/' characters, similar to how file system hierarchies are laid out. The final component of the anchor path is the anchor under which operations will be performed. Anchor names with characters after the terminating null byte are considered invalid; if used in an ioctl, EINVAL will be returned.

PF operation can be managed using the pfctl(8) program. Some example commands are:

```
 # pfctl -f /etc/pf.conf  Load the pf.conf file
# pfctl -nf /etc/pf.conf Parse the file, but don't load it
# pfctl -sr  Show the current ruleset
# pfctl -ss  Show the current state table
# pfctl -si  Show filter stats and counters
# pfctl -sa  Show EVERYTHING it can show
```

The pfsync protocol

Pfsync is another OpenBSD protocol that leverages Packet Filter's capabilities to update and manage state tables and therefore enables stateful inspection and Network Address Translation. Pfsync as well as CARP use IP multicast messages to transmit state change messages and each machine participating in this multicast group has a synchronization interface for this purpose. Pfsync uses IP protocol number 240 and the multicast group that is used is 224.0.0.240. In our setup CARP and Pfsync operate in the same redundancy group in order to allow for gracious IP failover and synchronization of state tables so that in the event of a failover, network traffic will continue to flow uninterrupted through the new master router.

Pfsync creates a pseudo-device on the host on which PF state table changes are sent, and the interface's name is also pfsync. pfsync(4) can use a physical interface in order to merge and keep in sync the state tables among multiple firewalls (if pfsync is used for the creation of a firewall).

To setup pfsync and synchronize it with a physical interface we can use ifconfig(8) using a specific parameter, namely syncdev. So in our case, we could write for instance:

```
 # ifconfig pfsync0 syncdev rl2
```

It is worth noting however that the pfsync protocol does not provide any cryptography or authentication mechanisms. Therefore, it would be preferable to either use a secure or a dedicated network for

state table traffic in order to avoid spoofing pfsync packets and alter state tables.

Alternatively it is possible to use the syncpeer keyword in order to specify the address of the backup machine we would like to synchronize with. Pfsync will use this address instead of multicast as the destination of pfsync packets, thus allowing the use of IPsec to protect communication. In this case syncdev must be set to the enc(4) pseudo-device, which encapsulates/decapsulates ipsec(4) traffic. For instance:

```
# ifconfig pfsync0 syncpeer 192.168.1.101 syncdev enc0
```

3.2.2 Linux - VRRP

The VRRP protocol

The Virtual Router Redundancy Protocol (VRRP) is a networking protocol whose scope is similar to CARP's as described above. It provides for automatic assignment of available Internet Protocol (IP) routers to participating hosts. This increases the availability and reliability of routing paths via automatic default gateway selections on an IP subnetwork.

The protocol achieves the aforementioned goal by creating virtual routers, which practically in an 'ACTIVE-PASSIVE' configuration is a master and backups acting as a member of a (redundancy) group. The hosts lying behind this virtual router are unaware of the existence of this redundancy group and therefore the IP virtual router is assigned as the default gateway. If the physical entity holding the master position fails at any given time another entity will automatically be selected in order to replace it.

VRRP provides information and allows for manipulation of the state of a router but not the routes that are processed and exchanged by that router. VRRP can be used on top of any L2 architecture and protocol such as Ethernet, MPLS or token ring and supports both IPv4 and IPv6. More information on the protocol can be found here [19].

According to the RFC, a virtual router must use a specific MAC address which lies in the range of `00-00-5E-00-01-XX`, where the last byte of the address corresponds to the Identifier of the Virtual Router (VRID) which, of course, is unique for every virtual router in the network. Only the master can use this MAC address in order to answer ARP requests for the virtual router's IP address. Entities within the redundancy group, same as with the CARP protocol, exchange heartbeats in order to verify that the master (and possibly the other nodes) is alive. These messages are exchanged using either the multicast IP address `224.0.0.18` or using the unicast IP addresses of the peers. The IP protocol number is 18.

The physical routers in the redundancy group need to be in the same LAN segment as communication between them is necessary and occurs periodically. Physical routers advertise themselves at configurable intervals that are known to the peers. Failing to receive heartbeats from the master over a period of three times the advertisement interval leads to the assumption that the master router is offline. Straight after, a transitional unstable state begins during which an election process takes place in order to choose which one of the backup routers will assume the master role. The smaller the advertisement interval, the shorter the black hole period at the expense of higher traffic in the network.

The election process is made orderly through the use of skew time, derived from a router's priority and used to reduce the chance of the thundering herd problem occurring during election. The skew time is given by the formula `(256-Priority)/256` (expressed in milliseconds) RFC 3768 .

Every physical router in a VRRP redundancy group has a priority in the range of 1 to 255 and the master will be the router with the highest priority value. This priority field is configurable and modifying it can prove useful in cases where a planned change of master is to take place. In that case, the master's priority will be lowered and this means that a backup router will pre-empt the master router status and this will reduce the black hole period. Backup routers are only supposed to send multicast messages and advertise their existence only during an election process except for the case when a physical backup router is configured with a higher priority than the current master. This practically means that when this router joins the redundancy group, it will pre-empt the master status. This way we can force a physical router to occupy the master state immediately after booting, in case for instance the new physical router is stronger than the current master. However, in the normal case when a master fails the backup with the highest priority will assume master state.

Keepalived

According to the description found on `keepalived.org` the main goal of Keepalived is to provide robust facilities for load balancing and high-availability to Linux systems and Linux based infrastructures. High availability is achieved by VRRP protocol and keepalived implements a set of hooks to the VRRP finite state machine providing low level and high- speed protocol interactions. The basic idea is that keepalived manages the failover of the virtual IPs using the VRRP protocol meaning that at any time the router that owns the virtual IPs replies to ARP requests (or neighbor solicitations for IPv6) and thus receives the traffic. One of the benefits of keepalived is that it provides sync_groups – a feature to ensure that if one of the interfaces in the sync_group transitions from master to backup, all the other interfaces in the sync_group will also transition to backup. This is important for our Active-Backup router deployment where asymmetric routing is not desired. The sync_group also includes information about the scripts to call on the local machine in the event of a VRRP transition to master, backup or fault state. The main deviation from the VRRP protocol that keepalived allows for is that Virtual routers can use the MAC address of the physical router who currently acts as master. The keepalived configuration can be found in the keepalived.conf file.

conntrack-tools

The conntrack-tools allow system administrators working on GNU/Linux to interact with the kernel's connection tracking system from the user-space. As mentioned above, the connection tracking system enables the stateful inspection of packets that traverse a router or a firewall.

The conntrack-tools provide two main programs:

- *conntrack*. Conntrack is a command line interface that allows the administrator to interact in a more friendly way with the connection tracking system compared with /proc/net/ip_conntrack. Conntrack-tools would allow the sysadmin to show, delete or update state entries that already exist in the kernel's connection tracking table.
- *conntrackd*. Conntrackd is the user-space connection tracking daemon. It can be used in order to build highly available firewalls or routers and it can also be used so that flow statistics can be collected after stateful inspection of the network traffic of the firewall or router.

3.3 Shortcomings

This implementation even though at a first glance satisfies our specifications has some shortcomings that lead to the decision that a different architecture should be sought. This shortcomings refer to the protocols and the tools examined as well as important challenges that were posed by the current Synnefo codebase.

3.3.1 OpenBSD implementation

The first shortcoming of this implementation was about the fact that CARP uses virtual MAC addresses for the Virtual interface that will contain the shared IP in the redundancy group. This MAC address is always in the range `00:00:5e:00:xx`. This would not generally be a problem except that when Synnefo creates a private isolated network, the MAC address of all cards in that network share the same prefix, and that is how isolation among networks is achieved. Therefore, CARP could not take advantage of the fact that it sends multicast messages so that hosts in the redundancy group can exchange heartbeats. Instead, the only possible solution was to only use a cluster of two hosts and clearly state the IP addresses of the peers so that unicast can be used instead.

Furthermore, even though the OpenBSD operating system provides out-of-the box support for useful protocols and tools such as CARP and PFsync, this approach was rejected as it was our intention to provide a uniform set of tools and platforms for Synnefo and since Linux could provide us with tools in order to achieve the same goal, we opted for that solution instead.

3.3.2 Linux implementation

The set of tools that were selected in order to implement the aforementioned architecture proved to be more user-friendly and therefore as fas as this architecture is concerned, most of our efforts were concentrated on the Linux-based implementation.

`Keepalived` provided the advantage of not having to use Virtual MAC addresses for the floating IP and therefore this feature would allow us to overcome the limitation set by the use of MAC prefixes for enabling isolation among networks. However, the challenge that arose concerned the addressing scheme that would be used for achieving high availability on the public network interfaces.

Typically, for achieving high availability using VRRP on two interfaces we would need three IP addresses: one IP address for each interface so that the heartbeats can be transmitted and received and a third IP address in the same subnet as the other two addresses that will be the virtual IP, that will be visible to the rest of the network. However, it is obvious that it would be an overkill to use three public routable IP addresses in our infrastructure since the main constraint is that we can only afford one public IP per user. Therefore, a different solution had to be found. The solution that we proposed is the following:

1. The heartbeats would be redirected to a third, independent ‘management’ network and we made the assumption that if a failure occurred in that subnet on the master router, that would trigger a failover in the other two interfaces of that machine.

```
vrrp_script ping_check {
    script "/etc/keepalived/ping_check.sh"
    interval 1
    fall 2              # required number of failures for KO switch
    rise 2              # required number of successes for OK switch
}


vrrp_sync_group G1 {
    group {
        D1
    }
    notify_master "/etc/keepalived/tran_script_mas.sh"
    notify_backup "/etc/keepalived/tran_script_bac.sh"
    notify_fault "/etc/keepalived/tran_script_fault.sh"
}

vrrp_instance D1 {
    interface eth0
    state MASTER
    virtual_router_id 61
    priority 150
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass zzzz
    }
    virtual_ipaddress {
        10.0.100.150/24 dev eth0
    }
    unicast_peer {
  10.0.100.2
    }
    track_script {
        ping_check
    }
    nopreempt
    garp_master_delay 1
}
```

2. In routed mode, which is the setting upon which public networks are built in Ganeti (in order to be able to use proxy ARP for traffic going to the Internet) it was mandatory to assign an IP address to every NIC that was created. Therefore, since the virtual address would characterize more than one NICs, it was not ideal to assign the same IP address to more than one NICs. However, we proceeded with the assumption that we could accept this deviation from the norm.

3. When routed mode is used, directions to direct traffic to a specific VM are passed using IP rules that are installed via specific scripts at a host level when a NIC appears or disappears. In our case when a failover occurs, even though potentially no NIC was modified, the IP rules would have to be modified at a host level, instantaneously. Therefore, that meant that the host would have to be aware of the state of the VM-router it is hosting (MASTER or SLAVE). For this reason we chose to implement a connection between the VM and the host machine using a serial port, that would send a signal from the VM to the host each time there was a change in the VM's status. The code below is the implementation on a custom setup where two physical hosts are informed through serial ports about changes in the status of the VMs in a redundancy group and modify

the respective `ip route` rules.

```
// Sample script to inform hosts about IP failover

#include <errno.h>
#include <error.h>
#include <fcntl.h>
#include <poll.h>
#include <signal.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/un.h>
#include <sys/wait.h>
#include <sys/time.h>
#include "virtserial.h"

struct guest_packet gpkt;
struct pollfd pollfd[2];
bool guest_ok;

static struct host_chars {
  char *path;
  int  sock;
  bool caching;
  bool throttled;
} chardevs[2] = {
  {
    .path = "/tmp/debianr2",
    .caching = false,
    .throttled = false,
  },
        {
    .path="/tmp/debian_r_3",
    .caching = false,
    .throttled = false,

        }
};

static int host_connect_chardev(int no)
{
  struct sockaddr_un sock;
  int ret;

  chardevs[no].sock = socket(AF_UNIX, SOCK_STREAM, 0);
  if (chardevs[no].sock == -1)
    error(errno, errno, "socket\n");

  sock.sun_family = AF_UNIX;
  memcpy(&sock.sun_path, chardevs[no].path, sizeof(sock.sun_path));
  ret = connect(chardevs[no].sock, (struct sockaddr *)&sock,
   sizeof(sock));
```

```
56
57    if (ret < 0)
58      error(errno, errno, "connect: %s", chardevs[no].path);
59
60    return chardevs[no].sock;
61  }
62
63  static int host_close_chardev(int no)
64  {
65    return close(chardevs[no].sock);
66  }
67
68
69  void sighandler(int signo)
70  {
71      int ret;
72      struct timeval trace;
73    pollfd[0].fd = chardevs[0].sock;
74          pollfd[0].events = POLLIN;
75
76      if (signo==SIGIO){
77    ret = poll(pollfd, 2, 1);
78    if (ret == -1)
79                  error(errno, errno, "poll %s", chardevs[0].path);
80          if (ret == 0){
81                  printf("no contact from Guest..\n");
82                  goto next;
83          }
84          if (pollfd[0].revents & POLLIN) {
85                  ret = read(chardevs[0].sock, &gpkt, sizeof(gpkt));
86                  if (ret < sizeof(gpkt))
87                          error(EINVAL, EINVAL, "Read error");
88                  if (gpkt.key == 1 && gpkt.value ) {
89        if (gpkt.value == 1) {      //master transition
90          guest_ok=true;
91          printf("debianr2 became master\n");
92          system("ip route add 10.0.100.150 dev tap0");
93          system("ip route add 10.0.100.150 dev tap0 table 100");
94          system("ip route flush cache");
95          system("date");
96        } else if (gpkt.value == 2) {
97          guest_ok=true;
98                                  printf("debianr2 became slave\n");
99                                  system("ip route del 10.0.100.150
    dev tap0");
100         system("ip route del 10.0.100.150 dev tap0 table 100");
101                                 system("ip route flush cache");
102                                 system("date");
103
104       } else if (gpkt.value == 3) {
105                           guest_ok = false;
106                           printf("debian r2 FAULT\n");
107         system("ip route del 10.0.100.150 dev tap0");
108         system("ip route del 10.0.100.150 dev tap0 table 100");
109         system("ip route flush cache");
110         system("date");
111       }
112                 } else  if (gpkt.key == 2) {
```

```
                        printf("to poulo\n");
                        error(EINVAL, EINVAL, "Guest error");
                }
        }
next:
  ;
  }

    return;
}



int main()
{

  int ret;
  struct sigaction action;

  memset(&action, 0, sizeof(action));
  action.sa_handler = sighandler;
  action.sa_flags = 0;
  sigaction(SIGIO, &action, NULL);

  ret = host_connect_chardev(0);
  fcntl(ret, F_SETOWN, getpid());
  fcntl(ret, F_SETFL, fcntl(STDIN_FILENO, F_GETFL) | FASYNC);
  printf("%d \n",ret);

  while(1){
  sleep(100);
  }

  return 0;
}
```

Chapter 4

Virtual Routers using Software Defined Networks

In this chapter, we concentrate our efforts on showing how we can leverage the concept of Software Defined Networks in order to implement virtual routers. As this started as a proof of concept in order to see to which extent we could integrate SDN in our current infrastructure, we decided to take a step back and not implement in our first prototype high availability features. The chapter is structured as follows: At first we describe the architecture for the virtual router and the simpler, in this case, network layout. Then we describe our implementation and finally, we present some shortcomings of this approach.

4.1 Architecture

Network Layout

In this implementation the network layout is rather simpler compared with the previous architecture. Every user is entitled to one virtual router (as he is entitled to only one public IP). This router is basically a virtual machine built from an image that contains all the required tools that allow the implementation of a Virtual router using SDN, namely support for OpenVSwitch[9] and for the controller on which all the required components are implemented. This VM has one Network Interface Card for the public IP, one or more NICs for the private networks whose traffic it manages and an extra management interface to exchange control information with the network service of Cyclades. More specifically, the router is associated with subnets through interfaces and the router has an interface for every subnet it represents. Consequently, the IP address of that interface will be the gateway of the subnet. Besides the internal subnets, the router, can also be associated with an external network in order to NAT traffic from internal networks to the external one.

A graphical representation of the layout can be found below.

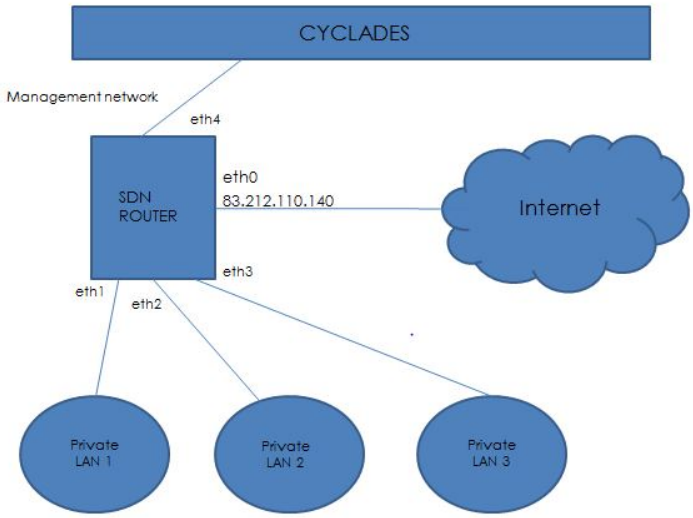

Figure 4.1: Network Layout with SDN virtual router.

Router Architecture

Following the principles of NFV, the virtual router should be an isolated Virtual Machine that would contain all the necessary components for the deployment of a virtual router. This assumption offers significant advantages regarding the ease of deployment and migration of Virtual Routers after taking into consideration that Synnefo is a cloud stack built on top of Ganeti, a cluster management tool that guarantees the persistence and ease of migration of VMs. In this context, the required components for this architecture are as follows:

- Virtual switches that support the OpenFlow protocol
- A SDN controller that will be used in order to implement the required components for a virtual router.

A VM that acts as a Virtual Router contains one virtual switch that supports OpenFlow. From now on we will call this switch VSwitch. Our goal is to be able to control the network traffic that the router manages using an OpenFlow controller. Therefore, it is necessary that the controller can 'see' that traffic. The purpose of the VSwitch therefore is to redirect network traffic that the router should manage to the controller and for that reason all the interfaces that belong to either the public network or the user's private networks upon creation are attached to the VSwitch. This action, of course, will turn the router useless and unable to process any traffic unless there is central logic that will configure and send the right rules to the VSwitch so that network traffic can be properly served. And that is the reason why a SDN controller is needed.

The term 'controller' is rather generic and describes the combination of a controller framework and controller modules. The controller framework is more or less an empty shell that allows for the establishment of connections with the VSwitches and provides the mechanisms and the basic tools so that a developer of network applications can shape network traffic based on specific OpenFlow events. This architecture requires components that will implement the following functions:

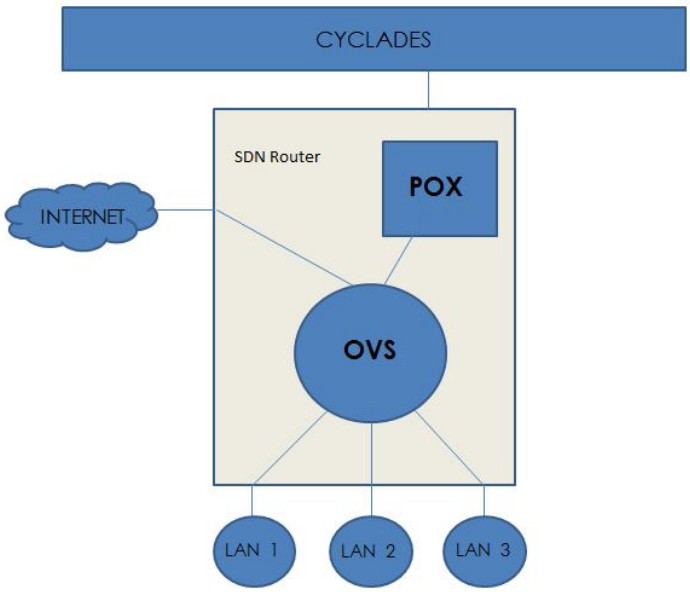

Figure 4.2: SDN virtual router architecture

- *NAT*. This is the most important component of the implementation. As the virtual router's main scope is to forward traffic from the realm of private networks to the Internet the SDN based implementation needs to allow for the implementation of a Network Address Translation, following the requirements mentioned in the second chapter.

- *Communication with Cyclades*. The router is one VM per user that is managed and controlled by Cyclades. According to the architecture of Synnefo, the component that is responsible for holding in a database the information regarding the characteristics of every network and of every VM, as well as sending the commands to Ganeti for any modification is Cyclades. In our architecture, it is essential that a router can in an asynchronous way ask for such information. Therefore, it is necessary to implement a messenger service to enable this direct communication.

- *Topology Recognition*. The router needs to know at any given point the number of VMs that the user has deployed and in which of the user's private networks they belong to. With this information, the controller will be able to build the right forwarding rules at the VSwitch so that the traffic can be managed properly.

In the following section we will discuss the workflow that our virtual router should implement. This workflow refers to actions that need to be taken upon router creation, modification of its interfaces, modifications in the number of hosts lying behind the router and finally modifications in the network interface cards of the aforementioned hosts.

Given our assumptions, every user is entitled to one virtual router to hold his one public routable IP address. The user has the option to choose which networks to attach the router to (zero or more networks) upon creation. The network service of Cyclades is responsible for handling and executing the user's request by sending the request in question to Ganeti and verifying that it has been properly executed. When the router-VM is spawned two events will occur: The controller will be launched and the VSwitch inside the VM will be created. All the interfaces that should be served by the router are automatically attached to the VSwitch that lies within the VM and this modification in the number of ports on the VSwitch will trigger a switch-to-controller event. This event signals that the controller

needs to be informed about a change in the number of router interfaces.

However, how can the controller and the router-VM know about the current status of the private isolated networks (e.g number of hosts in each network) and how can the controller gather all the required information in order to push the required flows to the Vswitch?

At this point, the controller only knows that it should represent x interfaces and cannot have access to information regarding the IP of its interfaces, the subnet, or the MAC prefixes. For this reason, asynchronous communication with the networking service and Cyclades is necessary and that is the reason behind the messenger service. Cyclades upon completion of the creation of the router or the modification of its network interfaces will initiate a connection with the router itself in order to send to it required information for each and every one of the interfaces it represents. These information include: IP, IP subnet, MAC, MAC prefix, and network ID as well relevant information regarding every host within each private network so that the router can establish the required flows that will allow for Network Address Translation and port forwarding on specific port numbers.

4.2 Implementation

This section describes the implementation of the aforementioned architecture after providing a brief description on the tools that were used. To begin with, we will provide more insight into the two major tools that were used for the implementation of the virtual router; OpenVSwitch and POX, a SDN controller written in Python. Afterwards, we will describe the code that was written and its integration in Synnefo.

4.2.1 Tools

OpenVSwitch

Providing network connectivity to VMs so that they can connect to the Internet requires that hypervisors will be able to bridge this traffic. The legacy tool to do that on Linux machines and Linux-based hypervisors is the built-in L2 switch, also known as the Linux bridge which is a fast an reliable way to do so. The Open vSwitch is an alternative to using Linux bridges that provides some significant advantages. OVS is mostly targeted at multi-server virtualization deployments, a scheme for which the Linux bridge is not very suitable. According to the official documentation of OVS, these environments are often characterized by highly dynamic end-points, the maintenance of logical abstractions, and (sometimes) integration with or offloading to special purpose switching hardware.

OVS has a series of characteristics that allow it to cope with the requirements mentioned above and these are the following:

- The mobility of state: All network state associated with a network entity (say a virtual machine) can be easily identifiable and migratable between different hosts. This may include traditional ”soft state” (such as an entry in an L2 learning table), L3 forwarding state, policy routing state, ACLs, QoS policy, monitoring configuration (e.g. NetFlow, IPFIX, sFlow), etc.

  Open vSwitch has support for both configuring and migrating both slow (configuration) and fast network state between instances. For example, if a VM migrates between end-hosts, it is possible to not only migrate associated configuration (SPAN rules, ACLs, QoS) but any live network

state (including, for example, existing state which may be difficult to reconstruct). Furthermore, Open vSwitch state is typed and backed by a real data-model allowing for the development of structured automation systems.

- Responsiveness to network dynamics: Virtual environments are often characterized by high-rates of change. VMs coming and going, VMs moving backwards and forwards in time, changes to the logical network environments, and so forth.

  Open vSwitch supports a number of features that allow a network control system to respond and adapt as the environment changes. This includes simple accounting and visibility support such as NetFlow, IPFIX, and sFlow. But perhaps more useful, Open vSwitch supports a network state database (OVSDB) that supports remote triggers. Therefore, a piece of orchestration software can ”watch” various aspects of the network and respond if/when they change. This is used heavily today, for example, to respond to and track VM migrations. Open vSwitch also supports OpenFlow as a method of exporting remote access to control traffic. There are a number of uses for this including global network discovery through inspection of discovery or link-state traffic (e.g. LLDP, CDP, OSPF, etc.).

- Maintenance of logical tags: Distributed virtual switches (such as VMware vDS and Cisco’s Nexus 1000V) often maintain logical context within the network through appending or manipulating tags in network packets. This can be used to uniquely identify a VM (in a manner resistant to hardware spoofing), or to hold some other context that is only relevant in the logical domain. Much of the problem of building a distributed virtual switch is to efficiently and correctly manage these tags.

  Open vSwitch includes multiple methods for specifying and maintaining tagging rules, all of which are accessible to a remote process for orchestration. Further, in many cases these tagging rules are stored in an optimized form so they don’t have to be coupled with a heavyweight network device. This allows, for example, thousands of tagging or address remapping rules to be configured, changed, and migrated. In a similar vein, Open vSwitch supports a GRE implementation that can handle thousands of simultaneous GRE tunnels and supports remote configuration for tunnel creation, configuration, and tear-down. This, for example, can be used to connect private VM networks in different data centers.

The goal with Open vSwitch is to keep the in-kernel code as small as possible (as is necessary for performance) and to re-use existing subsystems when applicable (for example Open vSwitch uses the existing QoS stack). As of Linux 3.3, Open vSwitch is included as a part of the kernel and packaging for the userspace utilities are available on most popular distributions.

The POX controller

POX[1] is a platform for the rapid development and prototyping of network control software using Python. Practically, it is the Python version of the NOX controller, one of the first SDN controllers. At a very basic level, it’s one of a growing number of frameworks (including NOX, Floodlight, Trema, Floodlight, etc.) for enabling the development of an OpenFlow controller.

POX serves as a framework for interacting with OpenFlow switches and it has served as one of the basic tools for the development of Software Defined Networking. It is used for prototyping, SDN debugging, network virtualization and controller design. POX is under active development and is the ideal tool for a developer who is willing to familiarize himself with SDN as it offers a rather smooth learning curve.

Some POX features:

- *'Pythonic' OpenFlow interface.*
- *Reusable sample components for path selection, topology discovery, etc.*
- *'Runs anywhere' – Can bundle with install-free PyPy runtime for easy deployment.*
- *Specifically targets Linux, Mac OS, and Windows.*
- *Supports the same GUI and visualization tools as NOX.*
- *Performs well compared to NOX applications written in Python.*

4.2.2 Code Walkthrough

The code that has been written for this thesis can be divided in 3 major sections:

1. Virtual router integration in the original Synnefo code.
2. Development of required modules for the POX controller.
3. Adjustments for OpenVSwitch integration.

Integration to Synnefo code

As mentioned, Cyclades uses Ganeti for management of VMs. From an administrator's perspective, Cyclades VMs can be handled like any Ganeti instance. Synnefo, provides a versatile implementation of the most common tasks that are related to Virtual Machines via `snf-manage server-*` commands. For Synnefo, every VM is a virtual server and consequently a virtual router is also a virtual server and should be treated that way.

The `snf-manage server-create` command can be used to create a new VM for some user. Our addition to this code was that we allow the user to choose whether this server is going to be a router or not. To achieve this goal, the steps that needed to be followed are mentioned below:

- Modify all server-related commands to allow for an extra option regarding creating a router or showing whether a specific VM is a router.

- Modify the database of Cyclades to allow for an extra attribute in the server's model, thus defining a concrete database representation of the server acting as a router.

- Ensure that when the virtual router is spawned, it will have a NIC in the management network that is required for its communication with Cyclades.

Every VM that has Internet connectivity and of course, our virtual router, needs to have (virtual) Network Interfaces. OpenStack Neutron and consequently Cyclades define ports as the representation of Network Interface Cards. Therefore, Synnefo provides an implementation of tasks that are related to Ports via `snf-manage port-*` commands. The commands `snf-manage port- {create,modify,remove}` allow us to manipulate a router's NICs and therefore had to be adjusted. The steps that were taken to achieve this goal are mentioned below:

- The router holds the gateway IP for the subnet whose traffic it forwards. We make the assumption that the gateway IP will be the first IP in the subnet so that any VM within the subnet will know its gateway IP. When Ganeti creates a new virtual network it allocates a pool of addresses for the Network Interface cards on that network. It also allows for some addresses to be reserved externally and therefore the gateway IP will be an external reservation. However, in order for an IP that is externally reserved to be allocated to a NIC, upon creation of this NIC, the parameter `--no-conflicts-check` needs to be passed. We implemented the logic for this procedure.

- After attaching a new NIC to a router. same as above we needed to make sure that the required communication with Cyclades takes place so that the router can be informed about the hosts that it should represent.

- When we attach a host to a private network, Cyclades is responsible for checking whether there is a router associated with this network, and if yes inform it with the right message about the addition of a new host. This way, the SDN controller will be able to push the required flows to the router for NAT and port forwarding.

POX modules

For the completion of this thesis, the development of two POX modules was required, namely the NAT and messenger modules. In this section we will walk through each of the two modules and explain in detail their functions.

Network Address Translation Module

In this module we define the class that will be responsible for implementing Network Address Translation functions. An object of the class NAT will be instantiated and registered in the core module of POX. These operations will take place within the `launch` method which is the first method that is called when a POX module is run.

```
1 def launch (dpid):
2
3 # Initializes the NAT module. Launches the ARP_helper module
4 # creates an instance of the NAT class and registers that instance
5 # in POX's core module.
6
7   ah.launch(use_port_mac = True)
8   dpid = str_to_dpid(dpid)
```

```
  log.debug('Starting NAT')

  n = NAT(dpid)
  core.register(n)
```

Upon instantiation of a NAT object, the constructor will initialize the required fields for the proper handling of switch, host and connection information.

```
class NAT (object):
  def __init__ (self, dpid):

    self.dpid = dpid
    self.subnet = None
    self.outside_ip = None
    self.gateway_ip = None
    self.inside_ips = {}

    self._outside_portno = None
    self._gateway_eth = None
    self._connection = None

    #Information about machines in the network
    # to reach x host we need to send traffic through port y
    self._mac_to_port = {}

    # Which NAT ports have we used?
    self._used_ports = set()
    self._used_forwarding_ports = set()
    self._managed_ips = set() #(net_id, ip, mac)

    #Which NAT ports have we used for forwarding rules ?
    self._forwarding = {} # (key=port, value=(net_id, ip, mac)
    self._nic_to_forw = {} #(key=cyclades_nic_id, value=forw_port)

    #Match ICMP seq numbers to IPs
    self._icmp_seq = {}

    # Flow records indexed in both directions
    # match -> FlowRecord
    self._record_by_outgoing = {}
    self._record_by_incoming = {}
    self._icmp_records = {}
```

After the connection between the switch and the controller is established, a function that is already implemented by the controller framework and is outside the scope of this analysis, we need to register handlers for OpenFlow events that our module should handle and also launch other required necessary services for our module, specifically the messenger service.

```
  def make_match (o):
    return NAT.strip_match(of.ofp_match.from_packet(o))

  def __handle_dpid_ConnectionUp (self, event):
    if event.dpid != self.dpid:
      return
    self._start(event.connection)

```

```
  def _start (self, connection):
    self._connection = connection
    self._connection.addListeners(self)
    messenger.launch()
```

Most of the methods implemented in the NAT class are handlers to OpenFlow events. According to the OpenFlow 1.0 spec, there are 3 types of events, namely *Controller-to-switch*, *Asynchronous* and *Symmetric*. In this case we are more interested in the *Asynchronous* case, which practically represent the messages that the controller receives from the switch when a specific event occurs. These events can represent modifications in the port status of a switch, packets that arrive at the switch but are subsequently forwarded to the controller because there is no installed flow that matches those packets etc. The method that handles `PacketIn` OpenFlow messages follows and is explained in detail below:

```

  def _handle_PacketIn (self, event):

    if self._outside_eth is None: return
    incoming = event.port == self._outside_portno
    if self._gateway_eth is None:
      self._arp_for_gateway()
      return
```

```
  def _arp_for_gateway (self):
    log.debug('Attempting to ARP for gateway (%s)', self.gateway_ip)
    done = False
    while not done:
      try:
          self._ARPHelper_.send_arp_request(self._connection,
                                            ip = self.gateway_ip,
                                            port = self._outside_portno,
                                            src_ip = self.outside_ip)
          done = True
      except:
          log.debug("OVS - connection not synced yet --- retry")
          time.sleep(1)


  def _handle_ARPHelper_ARPReply (self, event):
    if event.dpid != self.dpid: return
    if event.port != self._outside_portno: return
    if event.reply.protosrc == self.gateway_ip:
      self._gateway_eth = event.reply.hwsrc
      log.info("Gateway %s is %s", self.gateway_ip, self._gateway_eth)

  def _handle_ARPHelper_ARPRequest (self, event):
    if event.dpid != self.dpid: return
    dstip = event.request.protodst
    if event.port == self._outside_portno:
      if dstip == self.outside_ip:
        if self._connection is None:
          log.warn("Someone tried to ARP us, but no connection yet")
        else:
          event.reply = self._outside_eth
    else:
      if dstip in self.inside_ips.keys() or not self._is_local(dstip):
        if self._connection is None:
```

```
35            log.warn("Someone tried to ARP us, but no connection yet")
36          else:
37            event.reply = self._connection.ports[event.port].hw_addr
```

After the router has an external IP and has received the ARP reply for the public network's IP address (which is known for Synnefo's public networks by the administrator) then the router can begin processing `PacketIn` messages from the switch.

At first we need to determine the protocol of the packet received as icmp packets will be treated in a different way than TCP or UDP (for DNS or DHCP) packets.

```
1     # TCP UDP and ICMP handling
2     tcpp = False
3     udpp = False
4     icmpp = False
5
6     tcpp = packet.find('tcp')
7     if tcpp:
8         ipp=tcpp.prev
9     elif not tcpp:
10      udpp = packet.find('udp')
11      if udpp:
12        if udpp.dstport == 53 and udpp.prev.dstip in inside_ips:
13            if self.dns_ip and not incoming:
14                dns_hack = True
15        ipp = udpp.prev
16      else:
17        icmpp = packet.find('echo') #FIXME : replace echo with icmp
18        if icmpp:
19            ipp = packet.find('icmp').prev
20        else:
21            return
22
23    log.debug("incoming is %s",incoming)
24    if not incoming:
25      #Assume we only NAT public addresses
26      if self._is_local(ipp.dstip) and not dns_hack:
27         return
28    else:
29      # Assume we only care about ourselves
30      if ipp.dstip != self.outside_ip: return
```

At this point, we have all the information we need to start pushing flows to the switches. The first distinction that needs to be made is about the direction of the traffic. Outgoing traffic (from the private networks to the Internet) should be handled in a different way than incoming traffic. The difference lies in the fact that for incoming traffic there has to be a port forwarding rule installed that would allow traffic to cross the NAT.

Records

The first step when dealing with ingress/egress traffic to the router is to check whether the `PacketIn` that we have received corresponds to a new connection or to an established one whose flow has expired. OpenFlow 1.0 as well as the version of OpenVSwitch that we have used do not support connection tracking and therefore a different solution needed to be found. For that reason we have developed

a Record class that allows us to register information for established connections that will be used in order to renew flows that expired without the connection being closed.

```
class Record(object):
  def __init__(self):
    self.touch()

  @property
  def expired(self):
    return time.time() > self._expires_at

  def touch(self):
    self._expires_at = time.time() + FLOW_MEMORY_TIMEOUT


class FlowRecord (Record):
  def __init__(self):
    super(FlowRecord, self).__init__ ()
    self.touch()
    self.outgoing_match = None
    self.incoming_match = None
    self.real_srcport = None
    self.fake_srcport = None
    self.fake_dstport = None
    self.outgoing_fm = None
    self.incoming_fm = None

  def __str__ (self):
    s = "%s:%s" % (self.outgoing_match.nw_src, self.real_srcport)
    if self.fake_srcport != self.real_srcport:
      s += "/%s" % (self.fake_srcport,)
    s += " -> %s:%s" % (self.outgoing_match.nw_dst,
      self.outgoing_match.tp_dst)
    return s
```

Outgoing Traffic

When dealing with outgoing traffic, meaning traffic that flows from the private networks to the Internet, the first step is to check whether there is an existing record for this connection. If yes, a flow whose match/action fields are populated from the record values is pushed to the switch. Otherwise, we use the values of the PacketIn in order to build the two flows that need to be pushed; one for egress and one for ingress traffic. A new record is created and added in the list of records. The function `pick_port` allows to pick a port so that traffic to a private IP can be mapped through that port.

```
      log.debug("outgoing check")
      if icmpp:
        if ipp.dstip in inside_ips or ipp.dstip == str(self.outside_ip):
          self.respond_to_icmp(event)
          return
        elif not self._is_local(ipp.dstip):
          if icmpp.id not in self._icmp_seq:
            icmp_record = ICMPRecord(icmpp.id, ipp.srcip, packet.src)
            self._icmp_records[icmpp.id] = icmp_record
          self._icmp_seq[icmpp.id] = (ipp.srcip, packet.src)
          self.forward_icmp(event)
          return
```

```
13
14      record = self._record_by_outgoing.get(match)
15      if record is None:
16        record = FlowRecord()
17        if tcpp:
18          record.real_srcport = tcpp.srcport
19        elif udpp:
20          record.real_srcport = udpp.srcport
21        record.fake_srcport = self._pick_port(match)
22
23        # Outside heading in
24        fm = of.ofp_flow_mod()
25        fm.flags |= of.OFPFF_SEND_FLOW_REM
26        fm.hard_timeout = FLOW_TIMEOUT
27        fm.match = match.flip()
28        fm.match.in_port = self._outside_portno
29        fm.match.nw_dst = self.outside_ip
30        fm.match.tp_dst = record.fake_srcport
31        fm.match.dl_src = self._gateway_eth
32        fm.match.dl_dst = None
33
34        fm.actions.append(of.ofp_action_dl_addr.set_src(packet.dst))
35        fm.actions.append(of.ofp_action_dl_addr.set_dst(packet.src))
36        fm.actions.append(of.ofp_action_nw_addr.set_dst(ipp.srcip))
37
38        if record.fake_srcport != record.real_srcport:
39
    fm.actions.append(of.ofp_action_tp_port.set_dst(record.real_srcport))
40
41        fm.actions.append(of.ofp_action_output(port = event.port))
42
43        record.incoming_match = self.strip_match(fm.match)
44        record.incoming_fm = fm
45
46        # Inside heading out
47        fm = of.ofp_flow_mod()
48        fm.data = event.ofp
49        fm.flags |= of.OFPFF_SEND_FLOW_REM
50        fm.hard_timeout = FLOW_TIMEOUT
51        fm.match = match.clone()
52        fm.match.in_port = event.port
53
    fm.actions.append(of.ofp_action_dl_addr.set_src(self._outside_eth))
54        fm.actions.append(of.ofp_action_nw_addr.set_src(self.outside_ip))
55        if record.fake_srcport != record.real_srcport:
56
    fm.actions.append(of.ofp_action_tp_port.set_src(record.fake_srcport))
57
    fm.actions.append(of.ofp_action_dl_addr.set_dst(self._gateway_eth))
58        fm.actions.append(of.ofp_action_output(port =
    self._outside_portno))
```

```
1   def _pick_port (self, flow):
2     """
3     Gets a possibly-remapped outside port
4
5     flow is the match of the connection
6     returns port (maybe from flow, maybe not)
```

```
    """

    port = flow.tp_src
    if port < 1024:
      # Never allow these
      port = random.randint(49152, 65534)


    cycle = 0
    while cycle < 2:
      if (flow.nw_proto,port) not in self._used_ports:
        self._used_ports.add((flow.nw_proto,port))
        return port
      port += 1
      if port >= 65534:
        port = 49152
        cycle += 1

    log.warn("No ports to give!")
    return None
```

Incoming Traffic

As far as incoming traffic is concerned, our goal was to block all incoming traffic that did not belong to a connection that was already known to the router except for incoming requests on specific ports for which port forwarding rules could be used in order to allow traffic to pass for specific protocols (ssh, HTTP, FTP, etc). The router keeps a list of hosts in private networks and forwarding ports for these hosts and therefore can create the required rules in order to achieve the required functionality.

```
      match2 = match.clone()
      match2.dl_dst = None # See note below
      record = self._record_by_incoming.get(match)
      if record is None:
        if icmpp:
            if ipp.dstip == self.outside_ip:
                self.respond_to_icmp(event)
                return
        if tcpp:
          #if port can be found in dictionnary of forwarding rules
          if tcpp.dstport in self._forwarding:
            record = FlowRecord()
            record.real_srcport = tcpp.srcport
            record.fake_srcport = tcpp.dstport
            record.fake_dstport = self._pick_port(match)

            #Port forwarding rule --incoming
            fm = of.ofp_flow_mod()
            fm.flags |= of.OFPFF_SEND_FLOW_REM
            fm.hard_timeout = FLOW_TIMEOUT
            fm.match=match.flip()
            fm.match.in_port = self._outside_portno
            fm.match.nw_src = ipp.srcip
            fm.match.nw_dst = self.outside_ip
            fm.match.tp_dst = tcpp.dstport
            fm.match.tp_src = tcpp.srcport
            fm.match.dl_src = packet.src
            fm.match.dl_dst = self._outside_eth
```

```
29
30              fm.actions.append(of.ofp_action_dl_addr.set_src( \
31        self._connection.ports[ \
32        self._forwarding[tcpp.dstport][0]].hw_addr))
33              fm.actions.append(of.ofp_action_dl_addr.set_dst( \
34              self._forwarding[tcpp.dstport][2]))
35              fm.actions.append(of.ofp_action_nw_addr.set_dst( \
36          self._forwarding[tcpp.dstport][1]))
37
    fm.actions.append(of.ofp_action_tp_port.set_src(record.fake_dstport))
38              fm.actions.append(of.ofp_action_tp_port.set_dst(22))
39
40              fm.actions.append(of.ofp_action_output( \
41        port=self._forwarding[tcpp.dstport][0]))
42
43              record.incoming_match = self.strip_match(fm.match)
44              record.incoming_fm = fm
45              log.debug("port forw match %s", record.incoming_match)
46              log.debug("Added my flow")
47
48              # PORT FORWARD OUTGOING RULE
49
50              fm = of.ofp_flow_mod()
51              log.debug("Add pf outgoing flow")
52              fm.flags |= of.OFPFF_SEND_FLOW_REM
53              fm.hard_timeout = FLOW_TIMEOUT
54              fm.match=match.flip()
55              fm.match.in_port = self._forwarding[tcpp.dstport][0]
56              fm.match.dl_src = self._forwarding[tcpp.dstport][2]
57              fm.match.dl_dst = self._connection.ports[ \
58              self._forwarding[tcpp.dstport][0]].hw_addr
59              fm.match.nw_src = self._forwarding[tcpp.dstport][1]
60              fm.match.nw_dst = ipp.srcip
61              fm.match.tp_dst = record.fake_dstport
62              fm.match.tp_src = 22
63
64              fm.actions.append(of.ofp_action_dl_addr.set_src(packet.dst))
65              fm.actions.append(of.ofp_action_dl_addr.set_dst(packet.src))
66
    fm.actions.append(of.ofp_action_nw_addr.set_src(self.outside_ip))
67              fm.actions.append(of.ofp_action_nw_addr.set_dst(ipp.srcip))
68
    fm.actions.append(of.ofp_action_tp_port.set_dst(tcpp.srcport))
69
    fm.actions.append(of.ofp_action_tp_port.set_src(tcpp.dstport))
70              fm.actions.append(of.ofp_action_output(port =
    self._outside_portno))
71
72              record.outgoing_match = self.strip_match(fm.match)
73              record.outgoing_fm = fm
74              log.debug("%s installed", record)
75              log.debug("added outgoing fw flow")
76
77              self._record_by_incoming[record.incoming_match] = record
78              self._record_by_outgoing[record.outgoing_match] = record
```

```
1   def _pick_forwarding_port (self):
2
```

```
    port = random.randint(49152, 65534)
    cycle = 0
    while cycle < 2:
      if port not in self._used_forwarding_ports:
        self._used_forwarding_ports.add(port)
        return port
      port += 1
      if port >= 65534:
        port = 49152
        cycle += 1
    log.warn("No ports to give!")
    return None
```

ICMP protocol handling

Even though OpenFlow 1.0 allows for efficient handling of of TCP/UDP packets, this was not the case for ICMP packets from the internal networks that were supposed to be forwarded by the router to the Internet. Let us consider the following example. Imagine host with IP X in a private network that wants to ping IP address Y and Z simultaneously. IP addresses Y and Z are public routable IP addresses, so this means that all ICMP messages have to be forwarded by the router. In the case of TCP or UDP traffic, it would be easy to tell the two connections apart because of the different source ports and therefore we could push the required flows to the switch. However, this cannot be the case for ping messages as the ICMP protocol lies in the network layer of the TCP/IP stack and therefore one cannot use transport layer protocol ports to in telling different connections apart. This posed a significant shortcoming in our implementation and the work around that we decided to implement is to send every ICMP to the controller and separate ICMP connections by their ICMP sequence number. Our implementation might not work as expected as different hosts might choose the same sequence number for their ICMP connection and, therefore, it would be advisable to create a unique representation of the ICMP connection by combining in a unique representation the sequence number as well as the internal host's IP address and thus ensure that no connections can be mistaken.

```
  def forward_icmp(self,event):
    packet = event.parsed
    icmp_rec = packet.find('echo')
    #Create ICMP ECHO REQUEST
    icmp=pkt.icmp()
    icmp.type = pkt.TYPE_ECHO_REQUEST
    icmp.payload = packet.find('icmp').payload

    #Wrap it in an IP packet
    ipp = pkt.ipv4()
    ipp.protocol = ipp.ICMP_PROTOCOL
    ipp.srcip = self.outside_ip
    ipp.dstip = packet.find('ipv4').dstip

    #Wrap it in an Ethernet frame
    e = pkt.ethernet()
    e.dst = self._gateway_eth
    e.src = self._outside_eth
    e.type = e.IP_TYPE

    ipp.payload=icmp
    e.payload = ipp

    #Send
    msg = of.ofp_packet_out()
```

```
    msg.actions.append(of.ofp_action_output(port = self._outside_portno))
    msg.data = e.pack()
    msg.in_port = event.port
    event.connection.send(msg)

  def respond_to_icmp(self,event):
    packet=event.parsed
    icmp_rec = packet.find('echo')
    print "icmp id " + str(icmp_rec.id)
    print self._icmp_seq
    #Create ICMP ECHO REPLY
    icmp = pkt.icmp()
    icmp.type = pkt.TYPE_ECHO_REPLY
    icmp.payload = packet.find('icmp').payload

    #Wrap it in an IP packet
    ipp = pkt.ipv4()
    ipp.protocol = ipp.ICMP_PROTOCOL
    if icmp_rec.id not in self._icmp_seq:
      ipp.srcip = packet.find('ipv4').dstip
      ipp.dstip = packet.find('ipv4').srcip
    else:
      ipp.srcip = packet.find('ipv4').srcip
      ipp.dstip = IPAddr(self._icmp_seq[icmp_rec.id][0])

    #Wrap it in an Ethernet frame
    e = pkt.ethernet()
    if icmp_rec.id not in self._icmp_seq:
      e.src = packet.dst
      e.dst = packet.src
    else:
      e.dst = self._icmp_seq[icmp_rec.id][1]
      e.src = EthAddr(self._connection \
    .ports[self._mac_to_port[e.dst]].hw_addr)
    e.type = e.IP_TYPE

    ipp.payload=icmp
    e.payload = ipp

    #Send
    msg = of.ofp_packet_out()
    if icmp_rec.id not in self._icmp_seq:
      msg.actions.append(of.ofp_action_output(port = of.OFPP_IN_PORT))
    else:
      msg.actions.append(of.ofp_action_output( \
      port = self._mac_to_port[e.dst]))
    msg.data = e.pack()
    msg.in_port = event.port
```

```
        if icmpp:
            if ipp.dstip == self.outside_ip:
                self.respond_to_icmp(event)
                return
```

```
      if icmpp:
        if ipp.dstip in inside_ips or ipp.dstip == str(self.outside_ip):
          self.respond_to_icmp(event)
          return
```

Messenger Module

The messenger module is responsible for the correct communication of the network service of Cyclades and the virtual router. Normally Cyclades can only communicate with a VM by issuing commands to the Ganeti-master of the cluster the VM belongs to using the Ganeti RAPI. However, our architecture required that a router can communicate asynchronously with Cyclades when a change in a user or public network occurs. POX's messenger component provides an interface for POX to interact with external processes via bidirectional JSON-based messages. The messenger by itself is really just an API, actual communication is implemented by transports. Currently, transports exist for TCP sockets and for HTTP. Actual functionality is implemented by services. We implemented a custom service called CycladesService which is responsible for deserializing JSON messages that are received from Cyclades and raise Messenger Events that our NAT component handles in order to read and manipulate information about hosts that join or leave the network or about NICs that are attached or removed from the router. We also implemented a specific kind of event, namely `MessengerEvent` that will be handled by the NAT module.

```
class CycladesService (EventMixin):

    _eventMixin_events = set([MessengerEvent])

    def __init__(self, parent, con, event):
        EventMixin.__init__(self)
        self.parent = parent
        self.con = con
        self.count = 0
        self.listeners = con.addListeners(self)
        core.openflow.addListeners(self)
        #First message - handle manually
        self._handle_MessageReceived(event, event.msg)

    def _handle_ConnectionClosed(self, event):
        self.con.removeListeners(self.listeners)
        self.parent.clients.pop(self.con, None)

    def _handle_MessageReceived(self, event, msg):
        if self.count == 0:
            #self.con.send(reply(msg,msg="send next message with router
    info "))
            self.count += 1
        else:
            print "received message with new NIC_params. Updating router"
            print event.msg["msg"]
            if event.msg["msg"]["router"] == "yes":
                entry = MessengerEvent(event.msg["msg"]["router"],
                                       event.msg["msg"]["router_nics"],
                                       event.msg["msg"]["user_nics"],
                                       event.msg["msg"]["rem_nics"])
            else:
                entry = MessengerEvent(event.msg["msg"]["router"],
                                       {},
                                       event.msg["msg"]["host_nics"],
                                       event.msg["msg"]["rem_nics"])

            self.raiseEventNoErrors(entry)
            print core.components["NAT"]._forwarding
```

```
            self.con.send(reply(msg,msg="OK"))


class CycladesBot (ChannelBot):
    def _init(self, extra):
        self.clients = {}

    def _unhandled(self, event):
        connection = event.con
        if connection not in self.clients:
            self.clients[connection] = CycladesService(self, connection,
    event)
            core.register("messenger", self.clients[connection])


class MyMessenger (object):
    def __init__(self):
        core.listen_to_dependencies(self)

    def _all_dependencies_met(self):
        CycladesBot(core.MessengerNexus.get_channel("cyclades"))


def launch():
    MyMessenger()
```

```
class MessengerEvent (Event):
    def __init__(self, router, router_nics, user_nics, rem_nics={},
                 join=False, leave=False):
        super(MessengerEvent, self).__init__()
        self.router = router
        self.router_nics = router_nics
        self.user_nics = user_nics
        self.rem_nics = rem_nics
        self.join = join
        self.leave = leave
```

The NAT module by registering for messenger events can receive and update the information it has on router and hosts NICs. What is worth noting here is that the NAT module might receive the message from Cyclades about the modifications in the network topology before the actual switch in the router is updated (which the controller will find out by the `PortStatus` messages that it will receive from the switch). Therefore, it is important that we 'stall' the updates on the NAT module in order to resolve this race condition. For this reason, the updates will be performed by threads that will be kept busy until they have permission from the NAT module to do so.

```
  def _handle_core_ComponentRegistered(self,event):
    print event.name
    if event.name == "messenger":
      event.component.addListenerByName("MessengerEvent",
                                self.__handle_messenger_MessengerEvent)

  def __handle_messenger_MessengerEvent(self, event):
      if event.router == "yes":
         router_nics = event.router_nics
         user_nics = event.user_nics
         rem_nics = event.rem_nics
```

```
12          t = threading.Thread(target=self.modify_router_nics,
13                               args=(router_nics,user_nics,rem_nics,))
14          t.daemon = True
15          t.start()
16        else:
17          host_nics= event.user_nics
18          rem_nics = event.rem_nics
19          t = threading.Thread(target=self.modify_host_nics,
20                               args=(host_nics,rem_nics,))
21          t.daemon = True
22          t.start()
23
24    def modify_router_nics(self, router_nics, user_nics, rem_nics):
25      time.sleep(6)
26      for x in router_nics.keys():
27        mac = EthAddr(router_nics[x][0])
28        #mac_prefix = router_nics[x][1]
29        ip = IPAddr(router_nics[x][2])
30        net_id = router_nics[x][5]
31        for z,y in self._connection.ports.iteritems():
32          try:
33            if self._connection.ports[z].hw_addr == mac:
34              iface = str(y).split(":")[0]
35              comm = "ovs-vsctl -- set Interface " + iface + \
36         " ofport_request="+str(net_id)
37              os.system(comm)
38          except:
39              log.debug("Changing port numbers")
40        if not self._is_local(ip): #str(ip) == "10.2.1.3":
41          self.subnet = router_nics[x][4]
42          self.outside_ip = ip
43          self.gateway_ip = IPAddr(router_nics[x][3])
44          self._outside_portno = net_id
45          self._outside_port_handling()
46        else:
47          self.inside_ips[net_id] = (ip, mac)
48          for x in user_nics.keys():
49            mac = EthAddr(user_nics[x][0])
50            #mac_prefix = user_nics[x][1]
51            ip = IPAddr(user_nics[x][2])
52            net_id = user_nics[x][5]
53            if (net_id, ip, mac) not in self._managed_ips:
54              tcp_port=self._pick_forwarding_port()
55              self._forwarding[tcp_port] = (net_id, ip, mac)
56              self._managed_ips.add((net_id, ip, mac))
57              self._mac_to_port[mac] = net_id
58      for x in rem_nics.keys():
59        tup_to_check=(rem_nics[x][0], rem_nics[x][1], rem_nics[x][2])
60        for man_ip in self._managed_ips.copy():
61          if rem_nics[x][0] == man_ip[0]:
62            self._managed_ips.remove(man_ip)
63            del self._mac_to_port[man_ip[2]]
64            port_to_remove = -1
65            for z in self._forwarding.keys():
66              if self._forwarding[z][0] == rem_nics[x][0]:
67                port_to_remove = z
68                if port_to_remove != -1:
69                  del self._forwarding[port_to_remove]
```

```
70        if rem_nics[x][0] in (self.inside_ips and  \
71        self.inside_ips[rem_nics[x][0]]
72        == (rem_nics[x][0],rem_nics[x][1])):
73          del self.inside_ips[rem_nics[x][0]]
74
75    def modify_host_nics(self, host_nics, rem_nics):
76      for x in host_nics.keys():
77        mac = EthAddr(host_nics[x][0])
78        #mac_prefix = user_nics[x][1]
79        ip = IPAddr(host_nics[x][2])
80        net_id = host_nics[x][4]
81        if (net_id, ip) not in self._managed_ips:
82          tcp_port=self._pick_forwarding_port()
83          self._forwarding[tcp_port] = (net_id, ip, mac)
84          self._managed_ips.add((net_id, ip, mac))
85          self._mac_to_port[mac] = net_id
86
87      for x in rem_nics.keys():
88        tup_to_check=(rem_nics[x][0], rem_nics[x][1], rem_nics[x][2])
89        if tup_to_check in self._managed_ips:
90          self._managed_ips.remove(tup_to_check)
91          del self._mac_to_port[rem_nics[x][2]]
92          port_to_remove = -1
93          for z in self._forwarding.keys():
94            if self._forwarding[z] == tup_to_check:
95              port_to_remove = z
96            if port_to_remove != -1:
```

VM modifications

For the aforementioned architecture to work, we need to make slight modifications in the network manager of the router VM. The modifications regard the actions that occur after a change in the VM's NICs. Specifically, when a NIC is added, we check whether this NIC belongs to a public or private Synnefo networks (not the management network that we have provisioned for communication with Cyclades) and if that condition is true, then the Network Manager will attach this interface to the OVS that already exists within the VM.

4.3 Shortcomings

In this chapter we showed how we designed and implemented an architecture for a virtual router using SDN and NFV principles. This approach proves to be rather versatile and less invasive compared with the previous approach where we used keepalived, the VRRP protocol and conntrack tools and also gave significant insight into how SDN can be used within the Synnefo infrastructure. However, this approach comes with a few shortcomings that we discuss below:

For the purpose of this thesis as explained before we decided to use the POX controller. At the time of the development of the project, POX could only support Openflow v1.0, which is the first stable release of OpenFlow. OpenFlow 1.0 allows for the manipulation of only 12 packet header fields that cover packets from L2 to L4. It is easy to understand therefore, that some operations could not be implemented due to protocol limitations. The most characteristic example is that `traceroute` cannot be used because OpenFlow 1.0 does not allow the manipulation of the Time To Live field in the IP

header. As a result, when packets traverse the router either from the private networks toward the public Internet or the other way around the TTL field will not be decreased and therefore `traceroute` will not work properly.

Another shortcoming that is not a direct result of our work but rather a general issue of SDN at this stage is the possible conflicts that might arise from modules that might push overlapping flows to the switches. POX works by importing at start time modules that implement some logic or policy by pushing flows to switches in a reactive or proactive manner. Two flows are conflicting when their match fields and their priorities are the same. If a conflict occurs there are two possible options:

- If the `OFPFF_CHECK_OVERLAP` flag is set, then an error message is sent to the controller should a conflict occurs.
- If the `OFPFF_CHECK_OVERLAP` flag is not set, the new flow will overwrite the flow it causes a conflict with.

Consequently, conflicting modules can cause serious disruptions in the network and therefore at this stage it is the programmer's responsibility to ensure that now such conflicts occur. Ongoing research on this issue has led to various interesting projects, such as the Pyretic project which aims at the development of a SDN programming language based on Python that facilitates the development of elaborate network modules by providing high-level network abstractions.

In the introductory chapter, we stated and explained the requirements that RFC xxxx sets for Network Address Translation. Our module implements only a subset of these requirements, e.g it does implement Endpoint-Dependent mapping for TCP connections instead of Endpoint-Independent mapping. Despite the fact that not all requirements were implemented our module is fully functional and serves its purpose of proving the concept that SDN virtual routers could be integrated into the network service of Cyclades.

A final shortcoming which could also be a starting point for further discussion concerns the 'liberties' that the virtual router has with respect to choosing by itself which ports to assign to which host etc. In the current implementation, the virtual router is responsible for assigning forwarding ports to the hosts whose traffic it represents in an arbitrary way; When a host joins the network, the router picks a random unused port from a given pool of ports and matches this one to the host's traffic. In a different implementation, for instance, the user or administrator might be able to define specific policies or ports that could be assigned to a host through a specific API.

Chapter 5

Evaluation

In this chapter we intend to give a brief description of the advantages and disadvantages of each solution in terms of performance and versatility. We focus more on the case of the SDN router, where an important question that had to be answered regarding performance is how a virtual switch performs compared with legacy forwarding methods provided by the Linux kernel. Finally we conclude this chapter by providing a demonstration of the capabilities of the SDN virtual router.

5.1 Performance

5.1.1 HA virtual router

As far as our Highly Available router implementation is concerned, we can make a few observations regarding the expected and actual performance of the router. One architectural decision that we made was that we would redirect all management traffic, specifically the heartbeats and the state synchronization packets, through one dedicated subnet and therefore the throughput of the interfaces responsible for forwarding traffic would not be particularly affected when compared with the throughput of a simple Debian VM that we would use as reference. Indeed when comparing the throughputs we found them almost identical. On a side note, it should be mentioned that the traffic created by conntrack tools for state synchronization between two machines was rather heavy and potentially, in cases when the router would have to service many connections that would require a high capacity sync link to support the management network.

What was important however when evaluating the performance of this implementation was how fast the transition occurred when compared with the transition time specified by the VRRP protocol itself. According to the VRRP protocol, heartbeats are exchanged between the master and the slaves in the redundancy group at configurable intervals. A slave will begin advertising himself as a master after not having received any heartbeats from the master for three times the advertisement interval. The minimum value for the advertisement interval is 1 sec and, therefore, the minimum transition time when a failover occurs is 3 seconds. Our implementation showed no divergence from the numbers mentioned above and no extra delays were observed when experimenting with the implementation of this architecture. It should be noted however that even though the VMs (one master and one slave) where hosted in different physical hosts within the same data center, we did not apply any heavy traffic that would stress the system and potentially create discrepancies from what was theoretically expected.

5.1.2 SDN virtual router

The first question that we should pose to ourselves when evaluating the performance of a SDN virtual router is how much of an impact each one of its components has on router performance. And the component that needs to be evaluated primarily is the OpenVswitch as it is responsible for manipulating all router traffic. Luckily, a number of studies have been made comparing the performance of OVS with a traditional Linux bridge and their results as well as basic explanation for them are presented below.

The question of whether virtual switches pose significant overhead in the network has been hotly debated among developers. Specifically an issue of utmost importance to the network community is how OVS performance relates to the performance of a Linux bridge in terms of CPU utilization, and throughput [20]. Recent changes in the architecture of OVS have led to tremendous improvement in performance which has shown comparable throughput and in some scenarios CPU utilization eight times below the utilization of the Linux bridge. This performance can also be increased by using frameworks such as Intel's DPDK which accelerates OVS's performance even further.

Open VSwitch, which has been used for this project is widely accepted as the de facto standard OpenFlow implementation and the focus of the development team was on providing software that would not compromise neither generality in terms of features that it would support nor speed. The forwarding plane that OVS provides consists of two components: the userspace daemon called `ovs-vswitchd` and a kernel-module. All the forwarding decisions and network protocol processing are handled by the userspace daemon. The module is only responsible for caching traffic handling and tunnel termination. So when a packet is received the cache of flows is consulted. If a relevant entry is found, the associated actions with that entry take place otherwise the packet is passed to `ovs-vswitchd` or the controller so that a decision can be made. After a decision is made a cache entry is inserted so that subsequent packets can be handled promptly.

Until OVS 1.11, this fast-path cache contained exact-match 'microflows'. Each cache entry specified every field of the packet header, and was therefore limited to matching packets with this exact header. While this approach works well for most common traffic patterns, unusual applications, such as port scans or some peer-to-peer rendezvous servers, would have very low cache hit rates. In this case, many packets would need to traverse the slow path, severely limiting performance. OVS 1.11 introduced megaflows, enabling the single biggest performance improvement to date. Instead of a simple exact-match cache, the kernel cache now supports arbitrary bitwise wildcarding. Therefore, it is now possible to specify only those fields that actually affect forwarding. For example, if OVS is configured simply to be a learning switch, then only the ingress port and L2 fields are relevant and all other fields can be wildcarded. In previous releases, a port scan would have required a separate cache entry for, e.g., each half of a TCP connection, even though the L3 and L4 fields were not important. As a result, the introduction of 'megaflows' allowed OVS to drastically reduce the number of packets that traversed the slow path via the user space.

The change mentioned above along with others improve many aspects of performance. The performance that has been examined and presented refers to gains in flow setup, which was the area of greatest concern. In order to measure the performance of OVS accurately, the development team used netperf's TCP_CRR test, which measures the number of short-lived transactions per second (tps) that can be established between two hosts. OVS was compared to the Linux bridge, a fixed-function Ethernet switch implemented entirely inside the Linux kernel.

In the simplest configuration, the two switches achieved identical throughput (18.8 Gbps) and similar TCP_CRR connection rates (696,000 tps for OVS, 688,000 for the Linux bridge), although OVS used more CPU (161% vs. 48%). However, when one flow was added to OVS to drop STP BPDU packets and a similar ebtable rule to the Linux bridge, OVS performance and CPU usage remained constant whereas the Linux bridge connection rate dropped to 512,000 tps and its CPU usage increased over 26-fold to 1,279%. This is because the built-in kernel functions have per-packet overhead, whereas OVS's overhead is generally fixed per-megaflow. Enabling other features, such as routing and a firewall, would similarly add CPU load.

A second analysis [5] compared the latencies and throughputs between virtual switches and linux bridges in various setups. Those results are presented below.

	OVS	DPDK-OVS	Linux Bridge
Gbits/sec	1.159	9.9	1.04
Mpps	1.72	14.85	1.55

Table 5.1: NIC-OVS-NIC Throughput

	OVS	DPDK-OVS	Linux Bridge	NIC-NIC	VM-OVS-OVS-VM
TCP	46	33	43	27	72.5
UDP	51	32	44	26.2	66.4

Table 5.2: NIC-OVS-NIC Latency (microseconds/packet)

For the scope of this thesis, the aforementioned analysis was enough to prove that the performance of a virtual router created using OVS and SDN principles would not be significantly worse than a router where forwarding would take place using IPtables rules on Linux bridges. Some hits in performance, especially regarding latencies were known all along and appear in the following cases:

1. When a new unknown connection between two endpoints is initiated. In this case, the first packet of this packet stream has to leave the switch, reach the controller and come back followed by an extra packet that will install the required flows to the switch, flows that will determine how the switch will handle the subsequent packets of that connection. During this time and until the flows are installed all the packets that belong to this connection will reach out to the controller (for the same reason as the first one) and therefore this will cause some extra latency in the network.

2. When ICMP (or similar protocols) have to traverse the virtual router. As mentioned in the previous chapter, Openflow 1.0 which has been used for this project cannot differentiate between ICMP flows, especially when they traverse the NAT and are all characterised by the external IP of the router. For that reason, our approach consisted of sending every single ICMP packet to the controller so that the controller could inspect fields in the IP header such as the identification and sequence numbers and thus manage to differentiate between ICMP flows. This induces much higher latencies than when the kernel processes ICMP packets. More specifically, our results showed that the round trip time for a ping might be up to ten times higher than the normal round trip duration.

5.2 Demo

In this section we will provide a set of screenshots that depict the workflow of the SDN based virtual router from the moment it is created and until a host within a private network can be reached from the Internet. As this demo took place in a private Synnefo cluster with no access to public routable IP addresses, there is no public network but we will use the subnet `10.2.1.0/24` instead.

At first the virtual router is created. The OpenVSwitch inside it has been created but contains no ports (as at first the router is not connected to any user networks) and the POX controller is running. The router is ready to accept incoming TCP connections from Cyclades.

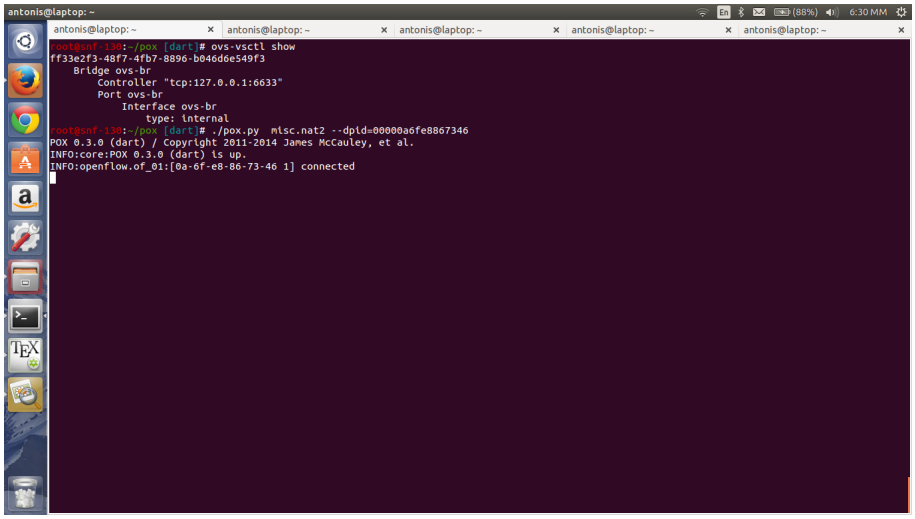

Figure 5.1: Controller connected and OVS contains no ports.

After the router has been created and initialized we begin adding network interfaces to it and thus we connect it to user networks, private or public. In this demo, we first connect the router to the 'public' network.

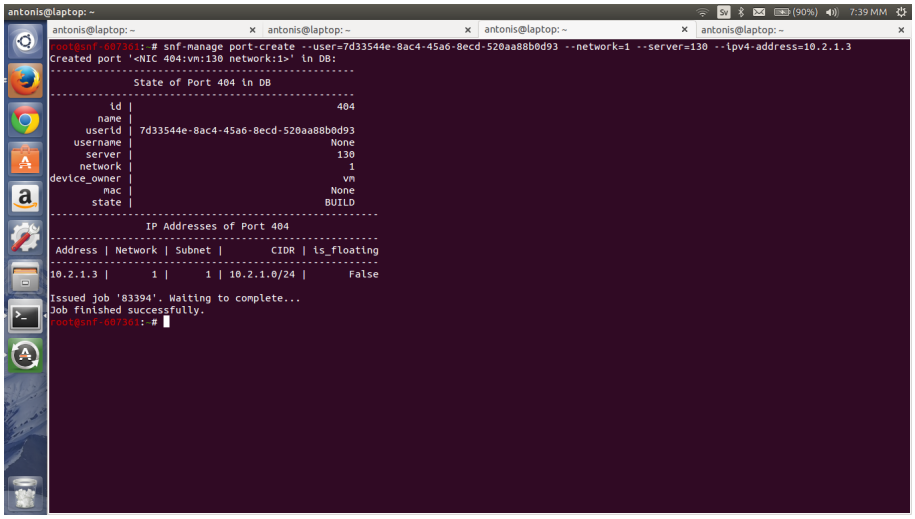

Figure 5.2: Router is connected to public network.

In the screenshot above we see the command that had to be used to create a port for an IP that belongs in the subnet 10.2.1.0/24 which, for the purposes of this demo is out public subnet. Upon successful

completion of the respective `Ganeti` job we see the following messages appearing in the router VM where the POX controller is running.

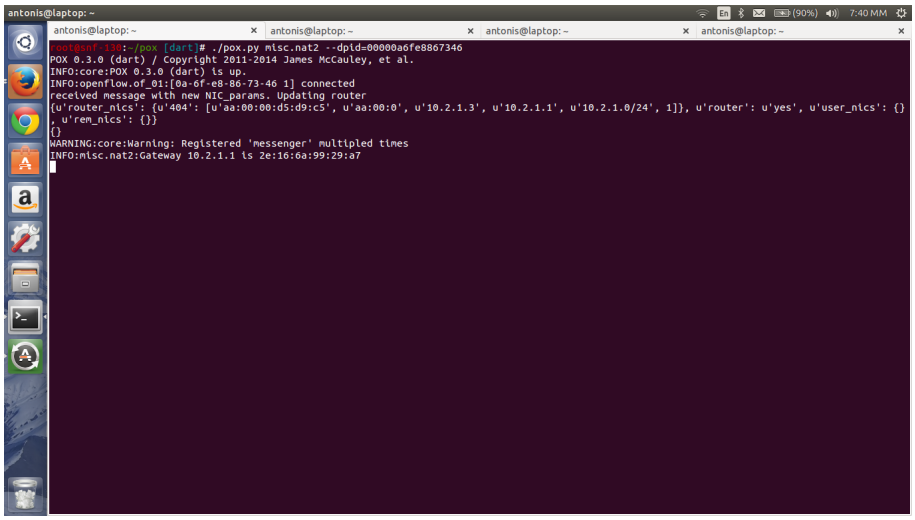

Figure 5.3: View of controller when public IP is connected to router.

Here we can see that after the port was added, a connection was initiated by Cyclades so that the controller of the virtual router can be informed about the changes in the network interfaces. By observing the message received by the router, we can see that a new NIC has been added to the router machine as well as the characteristics of this card; its IP and MAC addresses, the network id, the mac prefix of that network etc.

We continue by adding interfaces on a private network, namely 10.0.1.0/24, to both the router and to a simple VM.

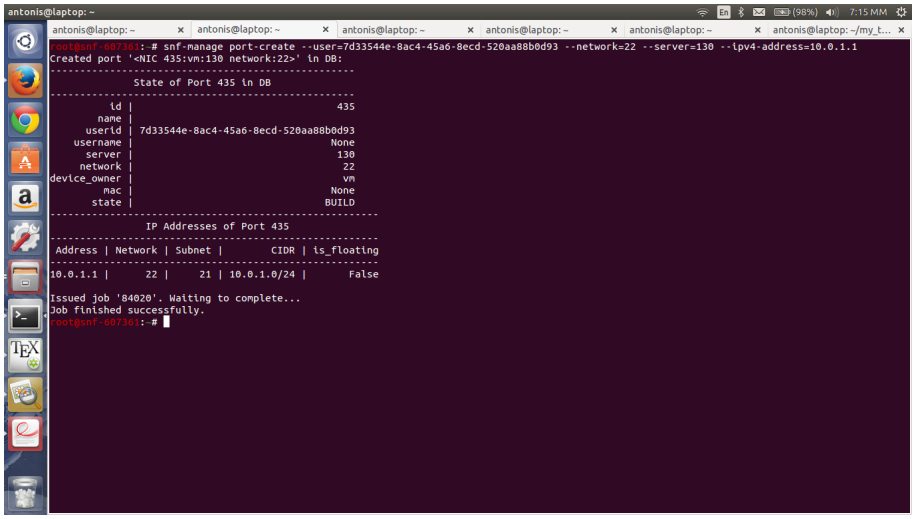

Figure 5.4: Router is connected to private network.

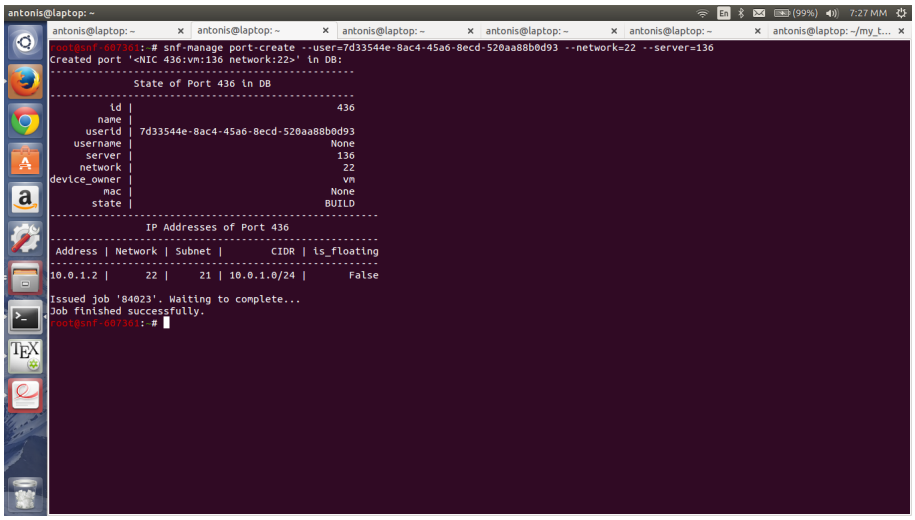

Figure 5.5: Host is connected to private network

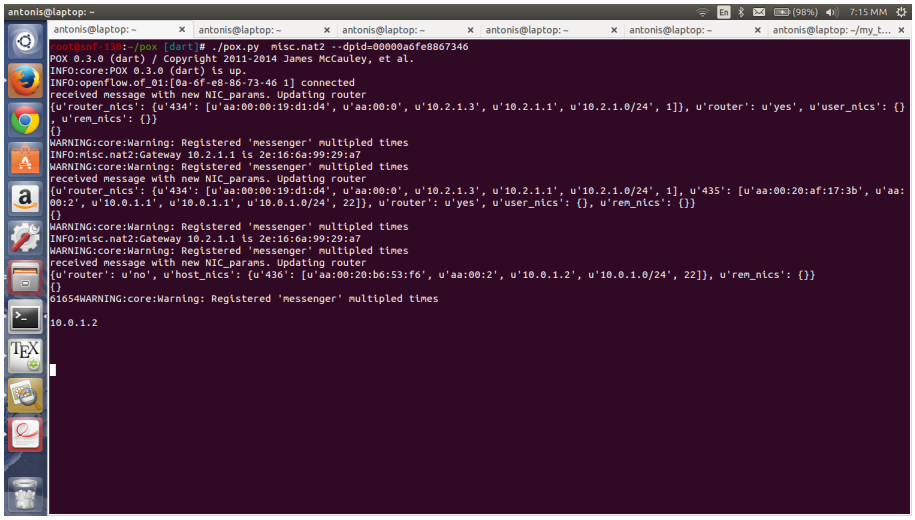

Figure 5.6: View of the controller after all NICs have been added.

At this point, the controller knows that the router has 2 interfaces, one in a public and one in a private network and that there is one host that has an interface in the private network the router is connected to. Therefore, it is possible at this point for the host in the private network to reach the Internet and for a host outside the private network to connect (via ssh for instance) to that host. The port that allows for port forwarding is, as shown on the screenshot, 61654.

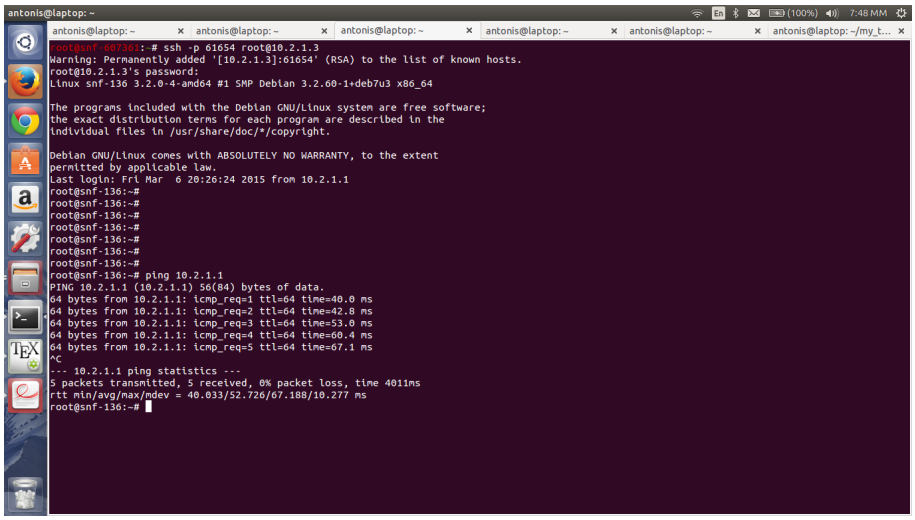

Figure 5.8: Established ICMP connection.

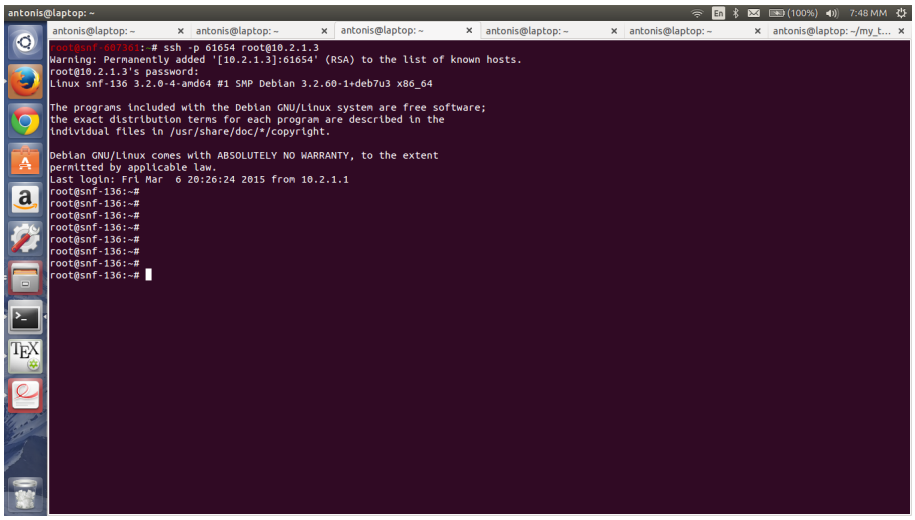

Figure 5.7: Established SSH connection.

In the picture above we show that from an external host (that intentionally has a NIC in the subnet 10.2.1.0/24) we can initiate a ssh connection to the host that lies in the private network behind the our NAT. From that host we ping back the host in the ‘public’ network and thus we prove that that our NAT can forward ICMP traffic as well. This demo also showcases one of the shortcomings of this SDN based implemntation and that is that the round trip time for a ping takes significantly more time than the time it would take in a legacy based setup.

Chapter 6

Conclusion

6.1 Concluding remarks

This thesis deals with the issue of implementing virtual routers in the networking infrastructure of Synnefo. This issue emerged from the necessity to provide a router entity in the public instance of our cloud, namely õkeanos as we could only provide only one public routable IPv4 address per user at the time. As the networking service of Synnefo is compliant with OpenStack's Neutron, an effort has been made to follow the paradigm set by the latter regarding architectural decisions that were made during the course of the project. For this thesis our efforts were focused on designing and implementing two architectures that would allow for the creation of an entity that would hold a user's public IPv4 address and would forward traffic from his virtual private networks to the Internet.

The first approach was based on legacy tools and protocols such as the VRRP and conntrack, tools that eventually allowed the implementation of a virtual router that is also characterized by high availability. For this architecture, upon router creation, we create a redundancy cluster that is comprised by a minimum of two virtual machines in an Active-Passive configuration. The virtual machines communicate via network messages, 'heartbeats', on a management network that all virtual machines in the cluster belong to and decide upon which one is going to be the master who will hold the shared IP (the one that the end hosts in the network see). This is a versatile implementation that performs well and meets the requirements of this thesis. However, it also presented some shortcomings since VRRP, the protocol that we are using requires configurations that Ganeti cannot provide out of the box, such as having two IP addresses on the same virtual NIC. Another shortcoming has to do with the fact that there are `ip route` rules on the physical hosts hosting the router VMs that need to be changed every time an IP failover occurs. For that reason we had to implement some kind of signaling protocol between the VMs and their respective hosts. This configuration even though it proved successful it introduces some serious security concerns and therefore was not optimal. The most important drawback of this implementation however was the fact that VRRP does not allow dynamic update of the number of NICs that a router has. Therefore, we opted for a different architecture.

The second approach consisted of designing and implementing virtual routers based on the principles of Software Defined Networks. For this implementation, we opted for routers that would not be characterized by high availability. The virtual router would essentially be a Virtual Machine with an OpenVSwitch that would be responsible for manipulating traffic from all the interfaces that would be attached to the router and would belong to user networks, private or public. The OpenVSwitch is connected to an OpenFlow controller that will dictate the logic upon which the router will handle incoming and outgoing network traffic. For that, we developed the required controller modules as well as a messenger service that allows for the communication between the router and Cyclades so that the router can query for specific information about a host or a network card, such as its IP or MAC address. Integrating this architecture proves to be successful and does not require significant modifications to

the existing Synnefo codebase. Regarding performance the major question that had to be answered is how an OpenVSwitch that is connected to a controller would compare to a Linux bridge that would forward traffic based on IPtables. The answer is that in most cases, the newest versions of OpenVSwitch perform substantially better, something which indicates that an SDN based implementation of virtual routers should be explored further and potentially be integrated in the production version of the platform. However, it should be noted that for this implementation we used the POX controller that only supports OpenFlow version 1.0. OpenFlow 1.0 lacks many important network features and capabilities that are provided by legacy tools and therefore we had to implement workarounds for some basic features to work (with some hits in performance) such as proper handling and forwarding of ICMP traffic by a NAT.

6.2 Future work

Regarding future work on the concept of virtual routers we can note the following.

6.2.1 HA virtual routers

1. Integrate fully our implementation for highly available routers in the Synnefo infrastructure. This would require more elaborate scripts for VM-host communication as well as the development of specific code in the Cyclades codebase that will allow for the creation of a redundancy group when a virtual HA router is spawned.

2. Handle security concerns that emerge from the concept of VM-host communication. The signaling protocol that we built in order to inform the host about changes in the router state could potentially by compromised by an attacker and therefore, security policies need to be created around it. A potential solution would be to host these router VMs in a specific, isolated set of physical hosts in the data center and prevent the user from having access to these VMs.

3. Implement fully the OpenStack Neutron API for virtual routers and also provide support for router related commands from the UI.

6.2.2 SDN virtual routers

1. Improve current code to implement more of the requirements of the RFC describing NAT. As this was intented to be a proof of concept not all requirements described in the NAT are implemented.

2. Develop methods that would allow for high availability. At the moment the virtual router constitutes a single point of failure for the network and this is an issue that should be dealt with in future work.

3. Port code to newer versions of OpenFlow in order to take advantage of more recent OpenFlow features. Specifically Ryu, another Python controller, currently supports OpenFlow 1.3 and would constitute a good option in order to implement missing features from the current proof of concept such as `traceroute`. Another alternative would be to explore the option of building similar modules for the OpenDaylight controller, as at the moment of the writing of this thesis is the only controller that can support handling of networks in big production setups.

4. Deal with the issue of potential conflicts that might arise from different modules that the OpenFlow controller in the router might execute at any given point. Our implementation only needs to run two different modules, the NAT module and the messenger module. Assuming that in the future we might want to implement more elaborate security policies such as security groups or custom firewalling per interface we will have to implement modules for those capabilities that will run alongside current modules, hence risking creating conflicts that will disrupt the network. Resolving flow conflicts that might be caused by different modules is an object of active research at the time of the writing of this thesis and it would be interesting to see how a solution could be implemented in the case of the virtual router.

Bibliography

[1] A. Al Shabibi. Pox wiki.

[2] O. Andreasson. Iptables tutorial 1.2.2.

[3] P. Ayuso Neira. Netfilter's connection tracking system. June 2006.

[4] SDX Central. What is nfv – network functions virtualization?

[5] M. Challa. Openvswitch performance measurements and analysis. November 2014.

[6] Synnefo developers. Cyclades network service.

[7] OpenBSD development team. Pf: Firewall redundancy with carp and pfsync.

[8] OpenBSD development team. Pf: Network address translation (nat).

[9] OpenVSwitch development team. Production quality, multilayer open virtual switch.

[10] K. Egevang and Francis P. The ip network address translator (nat).

[11] K. Egevang and P Srisuresh. Traditional ip network address translator (traditional nat).

[12] Open Networking Foundation. Software-defined networking (sdn) definition.

[13] S. Guha, K. Biswas, and S. Sivakumar. Nat behavioral requirements for tcp.

[14] Qusay F. Hassan. Demystifying cloud computing. 2011.

[15] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. okeanos: Building a cloud, cluster by cluster. IEEE Internet Computing, 17(3):67–71, May 2013.

[16] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. Synnefo: A complete cloud stack over ganeti. 38(5):6–10, October 2013.

[17] K. Lahey. Tcp problems with path mtu discovery.

[18] H. Balakrishnan G. Parulkar L. Peterson J. Rexford S. Shenker N. McKeown, T. Anderson and J. Turner. Openflow: Enabling innovation in campus networks. In ACM SIGCOMM Computer Communication Review, pages 69–74, March 2008.

[19] S. Nadas. Virtual router redundancy protocol (vrrp) version 3 for ipv4 and ipv6.

[20] J. Pettit. Accelerating open vswitch to "ludicrous speed". November 2014.

[21] J. Postel and J. Reynolds. File transfer protocol (ftp).

[22] R. Russell and H. Welte. Linux netfilter hacking.

[23] P. Srisuresh and M. Holdrege. Ip network address translator (nat) terminology and considerations.

[24] RackSpace support. Understanding the cloud computing stack: Saas, paas, iaas. IEEE Internet Computing, October 2013.