5-2012

OneCloud: A Study of Dynamic Networking in an OpenFlow Cloud

Gregory Stabler
Clemson University, gstable@clemson.edu

OneCloud: A Study of Dynamic Networking in an OpenFlow Cloud

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science
Computer Science

by
Gregory Eugene Stabler
May 2012

Accepted by:
Dr. Sebastien Goasguen, Committee Chair
Dr. Kuang-Ching Wang
Dr. James Martin

Abstract

Cloud computing is a popular paradigm for accessing computing resources. It provides elastic, on-demand and pay-per-use models that help reduce costs and maintain a flexible infrastructure. Infrastructure as a Service (IaaS) clouds are becoming increasingly popular because users do not have to purchase the hardware for a private cloud, which significantly reduces costs. However, IaaS presents networking challenges to cloud providers because cloud users want the ability to customize the cloud to match their business needs. This requires providers to offer dynamic networking capabilities, such as dynamic IP addressing. Providers must expose a method by which users can reconfigure the networking infrastructure for their private cloud without disrupting the private clouds of other users. Such capabilities have often been provided in the form of virtualized network overlay topologies.

In our work, we present a virtualized networking solution for the cloud using the OpenFlow protocol. OpenFlow is a software defined networking approach for centralized control of a network's data flows. In an OpenFlow network, packets not matching a flow entry are sent to a centralized controller(s) that makes forwarding decisions. The controller then installs flow entries on the network switches, which in turn process further network traffic at line-rate. Since the OpenFlow controller can manage traffic on all of the switches in a network, it is ideal for enabling the dynamic networking needs of cloud users. This work analyzes the potential of OpenFlow to enable dynamic networking in cloud computing and presents reference implementations of Amazon EC2's Elastic IP Addresses and Security Groups using the NOX OpenFlow controller and the OpenNebula cloud provisioning engine.

Dedication

To my parents, Bill and Donna, and my fiancé, Angela, for their their love, support, and guidance.

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Sebastien Goasguen, for giving me the opportunity to work with him. I have learned a lot from you over the years through various classes and research. Your support and guidance have been instrumental in completing this work. Thank you for pushing me when I needed it and challenging me intellectually.

I would also like to thank the other members of my research committee, Dr. Kuang-Ching Wang and Dr. James Martin, for their time spent reviewing my work. Dr. Wang provided me with guidance on OpenFlow and networking during the last year and allowed me to use his existing OpenFlow network for my research.

Also, I thank my parents. They have always been there to support me in life and have made me the person I am today. Thank you for always encouraging me to work harder and achieve more in life.

I also thank my fiancé, Angela, for her love, encouragement, and patience. I am looking forward to our new life together.

I also thank Aaron Rosen for his expertise and support with OpenFlow and networking. Your help and guidance was instrumental in completing my work.

Throughout my academic career, I have had the support of numerous friends. I would like to thank all of you for your support and friendship over the years.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction and Proposed Work

1.1 Cloud Computing

Cloud computing has emerged as a new paradigm for on-demand access to shared computing resources for end-users over the network. Cloud computing has become very popular because it saves organizations and businesses money. Data centers are expensive to build and maintain, and hardware becomes outdated and needs to be replaced frequently. Additionally, data center resources are frequently under-utilized. By moving applications and services to a cloud infrastructure, entities no longer have to maintain the hardware for their services. Most cloud providers offer a "pay as you go" service, where users only pay for the resources they consume. Therefore, entities can scale their services as demand changes. This allows them to save money when demand is low, yet react quickly to spikes in demand. Since these resources are shared by other users, the resource utilization is much higher, which lowers costs for all users. [12]

Early cloud infrastructures offered "Software As a Service" (SaaS) to end-users. They provided access to applications that were hosted on the cloud instead of local computing resources. Users accessed these applications through a web browser on their local machine. Examples of SaaS applications are Google Docs and Microsoft Office 365. Developers that wanted to write their own applications for the cloud turned to clouds that offered "Platform As a Service" (PaaS). PaaS provides the ability to deploy custom applications on the cloud without managing the underlying cloud infrastructure. These applications are developed using programming languages and tools supported by the cloud provider. Google App Engine, Windows Azure, and Heroku are examples of PaaS clouds. “Infrastructure As a Service” (IaaS) clouds are exemplified by the Amazon Web Services (AWS) such as EC2 and S3 as well as Rackspace (`http://www.rackspace.com`) services. IaaS clouds allow users to provision computing resources for processing, storage, and networking. Users can control the operating systems, storage environments, and networking components of their applications deployed in the cloud without managing the underlying cloud infrastructure [26].

As enterprises adopted the cloud computing paradigm, four deployment models emerged: *private*, *community*, *public*, and *hybrid* clouds. A *private cloud* is a cloud that is used by a single organization. It may be run by the organization or managed by a third party. A *community cloud* is shared between several organizations that usually have a common interest. *Public clouds* are available to the general public and are owned and operated by an organization that sells cloud services. A cloud infrastructure that consists of two or more clouds is known as a *hybrid cloud*. The individual clouds in a hybrid infrastructure can be private, public, or community clouds “that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability” [26].

At the IaaS layer, virtualization has been the key technology enabling on-demand, elastic resource provisioning [28]. Virtualization allows multi-tenancy of resources while providing isolation between applications. Key research thrust in on-demand provisioning of virtual machines (VMs) has led to several IaaS solutions such as Nimbus [23], Eucalyptus [29] and OpenNebula [36]. Recently, OpenStack [4] has received a lot of attention as well.

All of these solutions provide an EC2 interface, but few provide networking services. Cloud networking research has up till now been limited to the development of efficient overlay strategies to create networks of virtual machine instances [17], [34]. Providing users with dynamic networking capabilities poses several challenges to IaaS providers. Providers must ensure that users cannot adversely affect the functionality of the network for other users. They must isolate network services from other users and ensure the stability of the network. In this paper, we explore these challenges and show how OpenFlow coupled with one of these cloud solutions can provide an implementation for networking services offered by Amazon.

1.2 Cloud Networking Scenarios

Cloud users range from individuals to large enterprises. This diverse range of users presents many challenges to cloud providers because of the significantly different requirements of each user group. Individuals may only use the cloud for experimentation and education. Or, they might be exploring the use of the cloud to offset costs for a small start-up company. On the other hand, large enterprises are looking for highly scalable, reliable infrastructures to support their existing IT infrastructures. Differences aside, all users are looking for customized solutions to fit their needs.

Standard cloud providers allow users to customize computing and storage environments. Users can select the operating system, hardware specifications, and execution environments of their virtual machines. They have access to various storage solutions such as block storage, relational databases, and non-structured database applications. However, most cloud providers do not allow users to extensively customize their network environments.

Amazon Web Services provides a very flexible network infrastructure for its users. Users can customize the network topology, quality of service, and security access of the network for their cloud applications. Unfortunately, Amazon has not published details on the implementation of their customizable network solutions because of the competitive market. Research into dynamic cloud networking services has focused on the use of virtual network overlays to enable such services. Yet, few cloud computing platforms provide implementations of dynamic cloud networking such as those offered by Amazon Web Services.

Dynamic network services can be grouped into several use-case scenarios based on users' desires. When launching an instance in the cloud, a typical user cares about the location of resources, security, portability, and scalability. An analysis of these scenarios will provide a basis for understanding how OpenFlow can enable networking services within the cloud.

1.2.1 Resource Location

1.2.1.1 Instances in Single Site

The most common use-case is launching multiple instances in the cloud within the same data center, or region. Nodes in the data center run multiple instances and a user's instances may reside on different nodes within the cluster. The user's instances are connected via a common layer 2 domain (possibly through VLAN tagging) and belong to a single IP subnet, allowing easy communication

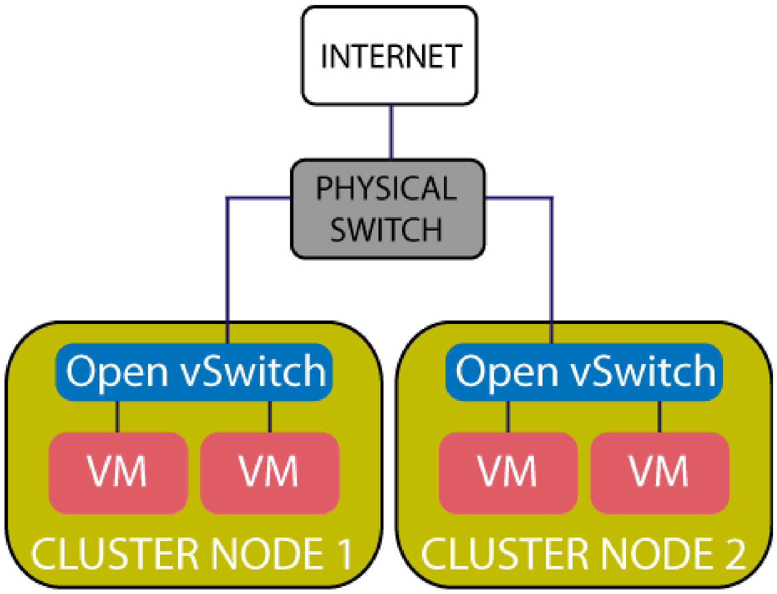


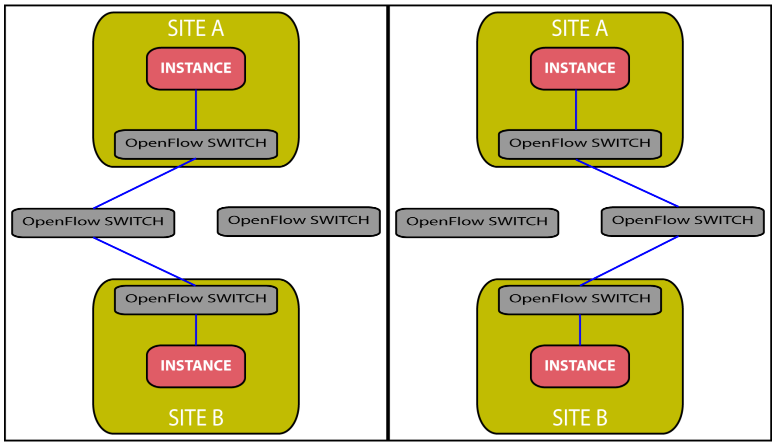
Figure 1.1: Connecting Nodes in a Single Site

within the user's network. More complicated configurations would partition instances into multiple subnets and layer 2 domains in the data center.

The cluster nodes would be connected together via OpenFlow-enabled network switches. As seen Figure 1.1, this could be done with virtual switches that support OpenFlow, such as Open vSwitch [5]. An OpenFlow enabled switch allows for greater control over inter-instance traffic. The switch can operate at layer 2, layer 3, or selectively between the two. Network Address Translation (NAT) can become customer specific and layer 2 domain specific. It can also be done for an entire group of instances on a single server or distributed among multiple servers. Using OpenFlow to implement NAT functionality also means the controller can dynamically alter flow rules to manage the translation between public IP addresses and private IP subnets.

1.2.1.2 Instances Across Multiple Sites

Some users desire the ability to launch instances at multiple locations, either to reduce latency or increase fault tolerance. However, firewalls and NAT translation may prevent instances from sharing layer 2 or layer 3 domains, complicating communication between them. Using OpenFlow enabled switches provides cloud providers with flexible tunnels between data centers. This allows instances in multiple data centers to share a common layer 2 or layer 3 domain. If multiple OpenFlow switches exist between the data centers, then Figure 1.2 shows how the tunnels can be



(a) Traffic Routed through Path 1 (b) Traffic Routed through Path 2

Figure 1.2: Re-Routing Traffic Between Multiple Sites

intelligently rerouted to improve network performance and adapt to network failures. Initially, traffic from Site A to Site B flows through the left OpenFlow switch in Figure 1.2a. When a failure occurs, traffic can easily be redirected through the right OpenFlow switch, as in Figure 1.2b. OpenFlow also allows providers to dynamically reroute IP addresses from an instance at one location to an instance an another location - the basis of Amazon's Elastic IP Addresses.

1.2.2 Security

Cloud computing raises many security concerns for users ([33] [14]). Users want their resources to be isolated from other users' resources. This can be done using VLAN tagging to create isolated layer 2 domains. An OpenFlow enabled cloud also supports VLAN tagging to isolate network traffic for each user. In addition, it is common practice to place some instances in a *security-heightened* subnet. Between this secure subnet and the default subnet, a firewall server is typically placed to filter and audit network traffic (see [19] for example). In an OpenFlow enabled cloud, the networking hardware becomes the firewall. OpenFlow rules are installed on the networking switches to block unwanted traffic to protected subnets - the basis for the Security Groups implementation presented later.

Another form of protection is against the distributed denial of service (DDoS) attacks that are increasingly problematic for popular service providers. In addition to the interest of detecting the onset of DDoS attempts, it is also useful to dynamically ramp up the service capability amidst an attack by launching new virtual servers and new, distinct IP subnets to cope with the short-term excessive demands. [22] demonstrates this capability using OpenFlow to mitigate the attack.

1.2.3 Scalability

In addition to the DDoS scenario described above, customer demands can increase for legitimate reasons as well. Addressing it requires either increased network throughput or increased server capacity or both. There is more than one way to increase a cloud's network and computing capacity depending on the physical and virtual organization of the infrastructure. The solution might involve utilizing more network interfaces and hosts in the data center, or it might require expanding the application across multiple data centers. As discussed previously, an OpenFlow cloud removes the limitations placed on applications that must scale across data centers, and can provide load balancing support to numerous instances ([41]).

1.3 Proposed Work

To enable such scenarios, we propose creating an OpenFlow cloud that enables dynamic networking services for its users. Our approach is to create an experimental cloud, Clemson OneCloud, that uses OpenFlow to manage network traffic within the cloud. The network traffic within OneCloud will be managed by a custom OpenFlow controller implemented on top of the NOX controller [3]. The cloud will use the OpenNebula 3.0 provisioning engine. The OpenNebula front-end server will be responsible for providing the OpenFlow controller with changes in the networking environment. Initially, OneCloud will focus on two networking services: dynamic mapping of IP addresses and dynamic firewall solutions. These services are inspired by Amazon's EC2 Elastic IP Addresses and EC2 Security Groups.

Our proposed solution will differ from current open-source cloud offerings in that it will provide an open-source implementation of dynamic networking services through the use of OpenFlow for network management. Other cloud offerings, such as OpenStack and Eucalyptus, provide Elastic IP and Security Group services through centralized network servers that analyze and rewrite each packet individually. OneCloud will use the network hardware to enable such services, leading to enhanced network performance. CloudNaaS and other cloud implementations have used OpenFlow in the cloud, but its use is limited to isolating networks with VLAN tagging and creating optimal paths in a data center. OneCloud will use OpenFlow to control network traffic routing and enforce higher-level security rules that cannot be implemented with VLAN tagging.

Chapter 2

Background and Technologies

2.1 Amazon Web Services

Amazon began offering IT infrastructure services to businesses in 2006 in the form of Amazon Web Services (AWS) [10]. These services offered users low-cost, agile solutions to cloud computing in the form of computing power, storage, databases, messaging, and networking. AWS solutions are flexible, scalable, elastic, and reliable [8]. Amazon's Elastic Compute Cloud (EC2) provides users with the ability to provision and scale compute power across the cloud platform. Simple Storage Service (S3) provides users with a distributed, scalable, storage architecture for fast and reliable data storage. Dynamic networking services provided include Virtual Private Cloud (VPC), which allows users to create custom network topologies, Elastic IP, which provides dynamic IP addressing, Route53, which provides a scalable dynamic Domain Name System (DNS) service, and many others. Database solutions include DynamoDB for NoSQL database services and Relational Database Service (RDS) for MySQL and Oracle databases. All of these services combine to form a very robust and flexible cloud computing platform that has revolutionized the way many businesses approach information technology.

2.1.1 Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) is a "web service that provides resizable compute capacity in the cloud" [8]. Users are provided with complete control over their computing environments such as networking, security, scaling, load balancing, and physical location. Users can provision instances in the cloud and control the networking infrastructure with Elastic IP addresses and Virtual Private Cloud (VPC). AWS also offers dynamic DNS, routing, and firewall services to its users. Instances can scale automatically to accommodate changes in demand, which lowers IT costs because users are charged only for the resources that they use. Additionally, users can control access to computing resources using EC2 Security Groups. The flexibility provided by Amazon's EC2 service has made it an extremely popular platform for cloud computing. Due to EC2's rising popularity, many major cloud platforms now offer an EC2 compatible interface [23][29][36][4].

2.1.1.1 Elastic IP Addresses

Amazon Elastic IP addresses are static IP addresses that can be dynamically remapped to running instances within the cloud [35]. Each instance in Amazon's EC2 is assigned static public and private IP addresses. The public address is mapped to the private address using a 1:1 NAT mapping [35]. Once the instance is terminated, the public address is no longer valid. Any new instances will have different IP addresses, so providing a service to users becomes difficult. However, Elastic IPs are associated with a user's account and not a specific instance. Therefore, users can programmatically remap the address to a new instance in the cloud. This enables the user to mask any network changes or failures without having to wait for a DNS update to propagate through the network or a new host to be configured and brought online.

2.1.1.2 Security Groups

AWS provides firewall services through EC2 Security Groups. EC2 Security Groups consist of a set of rules that govern incoming traffic destined for an instance in the cloud. All outbound traffic is automatically allowed for EC2 instances and cannot be controlled by EC2 security groups. An instance can be assigned to any number of security groups when it is launched. Once launched, it cannot be removed from or added to any groups; however, a group's rules can be added, removed, or modified, and the changes are applied to instances in real time. The rules for each group are aggregated together and applied to the instance. This allows the user greater flexibility in determining firewall rules for a virtual machine by assigning it to multiple groups at runtime. [9]

Security Group rules consist of a *protocol*, *from port*, *to port*, and *source*. Rules enable a specific *source* to access an instance using a certain *protocol* (TCP, UDP, or ICMP). For TCP and

UDP traffic, the *from* and *to* ports specify a range of ports to which the rule is applied. In the case of ICMP rules, the *from port* is the ICMP type number and the *to port* is the ICMP code. The *source* can be a single IP address, a range of IP addresses, or another Security Group. [9]

2.2 OpenFlow

OpenFlow is a protocol that provides the capability to control flows within a network from a centralized software controller. The OpenFlow standard is built on the fact that most networking hardware contains flow tables for managing network traffic [25]. The OpenFlow API provides a method for accessing and updating the hardware's flow tables to control the network without exposing the internal workings of the vendors' hardware. When a switch or router receives a packet, it checks to see if it matches any flows in the flow table. If not, it forwards the packet to the software controller. The controller examines the packet and decides where it should be sent. The resulting flow rule is then sent to the switch and installed in its flow table. All future packets that match this flow rule will be processed by the switch at line rate. This eliminates the bottleneck of previous software controlled networks in that once a flow rule has been created, packets are processed at line rate by the hardware, not by a central server. OpenFlow supports filtering network traffic at layers two, three, and four, but the controller can examine packet contents at any layer. [25]

OpenFlow is aptly suited for cloud computing because of its dynamic range of capabilities. Controlling a network with OpenFlow allows users to 1) flexibly associate computing end hosts (with layer two, three, or higher addresses or contexts) with network datapaths, 2) dynamically alter such associations for load balancing or failure fallback, 3) provision distinct network datapath properties such as security, isolation, or quality of service, and 4) enable virtualization of the physical network into traffic "slices" and delegate their control to different admin entities (as seen in [1]). These features are useful from either a provider or a user's perspective. A provider can leverage OpenFlow to implement such user features according to its policy preferences, while users can explore a range of customization of the cloud environment to better meet their applications.

2.3 Open vSwitch

Open vSwitch [30] is a "production quality open source software switch designed to be used as a [virtual switch] in virtualized server environments" [5]. It supports standard management interfaces such as sFlow, NetFlow, RSPAN, and CLI, and can be extended programmatically. It can be used to manage the networking for VMs on the same physical host or forward traffic between VMs on different physical hosts. Open vSwitch supports popular Linux-based virtualization platforms including KVM, VirtualBox, Xen, and XenServer. Additionally, Open vSwitch supports the OpenFlow protocol. This means that an Open vSwitch switch can be controlled by an OpenFlow controller within a network.

2.4 Global Environment for Network Innovations (GENI)

The Global Environment for Network Innovations (GENI) is an NSF sponsored "virtual laboratory for at-scale networking experimentation"[2]. It is a collaboration between research institutions, private industrial teams, and non-profit organizations. GENI is designed to facilitate future Internet research and provides a shared, heterogeneous infrastructure of federated compute systems across the United States.

GENI employs OpenFlow technology for its core network services. The GENI network is composed of OpenFlow-enabled switches managed by aggregate managers, allowing users to provision a slice of the network resources for research. The GENI network features two core VLANs that span the Internet2 and National Lambda Rail fiber networks [2] and interconnect several college campuses and research institutions. These VLANs provide experimenters with end-to-end layer 2 connectivity that spans multiple geographic locations. This enables experimenters to test new, innovative, non-IP based network protocols for the future Internet.

2.5 OpenNebula

OpenNebula is an open-source cloud computing framework "aimed at developing the industry standard solution for building and managing virtualized data centers and cloud infrastructures" [6]. The flexibility of the OpenNebula framework enables the creation and management of virtualized infrastructures that provide private, public, and hybrid IaaS clouds. It supports KVM, VMware, and

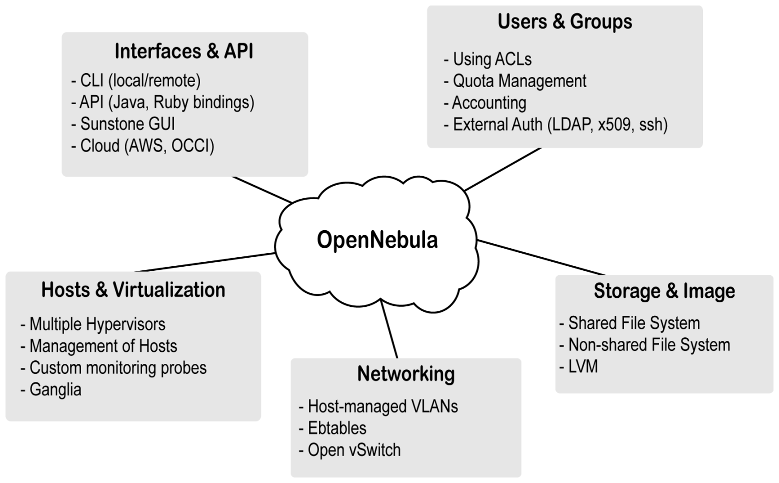


Figure 2.1: OpenNebula Architecture Components

Xen as the underlying hypervisors. OpenNebula also provides a centralized management interface for virtual and physical resources. OpenNebula allows seamless integration with other products and services - management tools, VM schedulers, virtual image managers, and hypervisors - through its highly extensible plug-in framework [6].

The core of OpenNebula provides operations related to storage, networking, and virtualization through a set of *drivers* [36]. The drivers are key to OpenNebula's extensibility and flexibility. Figure 2.1 shows the various components of OpenNebula and drivers supported by each component. The networking component supports drivers for creating host-managed VLANs through the Linux Kernel and interfacing with Open vSwitch. This allows users to tailor the networking infrastructure to meet their environment. Similarly, the storage component supports shared file systems, non-shared file systems, and Logical Volume Manager (LVM). The storage component is configured for each host in the cloud, so users can share NFS volumes on some hosts, but use SSH to copy virtual machine images between other hosts. As mentioned previously, the virtualization module supports multiple hypervisors. Each hypervisor is encapsulated in a driver that abstracts the basic virtualization functionality needed to provision and manage instances. OpenNebula also supports drivers for its authentication component. A cloud can use the built-in support for access control lists (ACLs) and local user accounts, or rely on external authorization such as LDAP or X509.

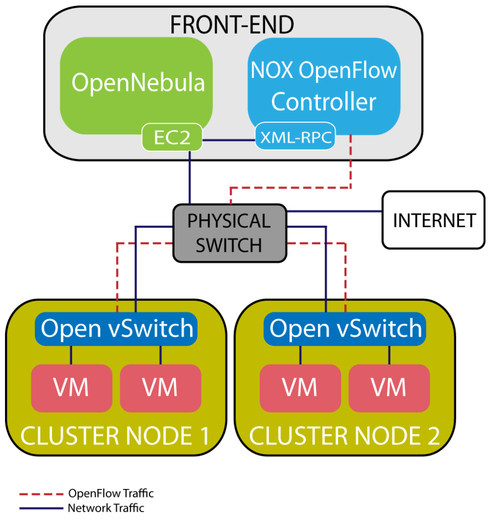


Figure 2.2: OneCloud Architecture

2.6 OneCloud

Clemson University OneCloud (`https://sites.google.com/site/cuonecloud/`) is an experimental cloud infrastructure based on the OpenNebula cloud framework. It is an IaaS system that enables users to provision virtual machine instances using the KVM hypervisor. OneCloud offers an EC2 front-end to its users.

OneCloud was created to research *dynamic cloud networking* infrastructures; it is available to external users upon request. Current research is focused on the dynamic networking capabilities of the OpenFlow protocol and how they can be leveraged within the cloud. Early results of developing OneCloud are presented in this paper. All hypervisors of OneCloud are physically connected on an OpenFlow network and use Open vSwitch [5] as the virtual machine bridge.

2.6.1 OneCloud Architecture

The OneCloud infrastructure consists of a front-end server and cluster nodes. The OneCloud front-end server is running OpenNebula 3.0 and the OpenNebula EC2 Query service, which provides an Amazon EC2 Query API compatible interface. The network within OneCloud is controlled by an OpenFlow controller running inside a virtual machine on the front-end node. Our controller application is written in Python and uses the NOX OpenFlow controller [3]. It also exposes an XML-RPC interface for interaction with the OpenNebula EC2 API.

OneCloud cluster nodes use the Kernel-based Virtual Machine (KVM) hypervisor for virtualization. In addition, virtualized networking for each node is provided by Open vSwitch 1.2.2 with the bridge compatibility layer enabled. Open vSwitch is a software-based virtual switch that supports the OpenFlow protocol. As seen in Figure 2.2, each node is configured with a single Open vSwitch bridge that is connected to our OpenFlow controller. When a virtual machine instance is started, it is assigned a single private IP address. The instance's network interface is attached to the Open vSwitch bridge on the host node, allowing our controller to manage its network traffic.

Chapter 3

Related Work

Networking research has shifted towards a focus on future Internet architectures in recent years. This shift has led to an increased interest in the use of OpenFlow for managing networking environments. However, dynamic networking in cloud environments has mainly focused on network overlays to deliver custom topologies to cloud users. Other OpenFlow research has focused on load balancing and virtual machine migrations, but not in the context of IaaS cloud offerings. Some research has been conducted on the use of OpenFlow in cloud computing, but it does not focus on providing networking services to the end user. We will examine current research related to these topics and how OneCloud differs from these solutions.

3.1 Virtual Network Topologies and Overlays

The majority of research in dynamic networking in grid computing has focused on the use of virtual network topologies, or network overlays, to provide custom network topologies to users. Most overlays are implemented at the application-level, using custom software to translate between the virtual and physical networks, as seen in [11][37][38][39][40]. More recent research has investigated the use of peer-to-peer (P2P) technology for managing network overlays [16][17][28][18][42].

Application-level overlays require processes on nodes to pass packets between the virtual network space and the physical network space. These processes translate packet headers between the two domains. One such implementation of this approach is ViNe [39]. ViNe creates a mapping between private IP addresses in the virtual address space and physical IP addresses in the physical infrastructure. IP aliasing is used to map the physical addresses to the nodes hosting the corresponding virtual addresses. A similar project, VNET [37][38], uses a proxy application to create a virtual network at layer 2 of the network stack. The proxy application resides on physical hosts and creates a layer 2 network topology across multiple physical nodes. [43] formalizes the virtual network embedding problem and presents a brokered architecture for creating custom topologies using VLAN tagging at layer 2.

P2P technology has also been used to implement robust, scalable overlays in [16][17][28][18][42]. The P2P approach takes advantage of the stability and reliability of P2P networks. Additionally, the overlays scale well due to the decentralized nature of P2P networking. The IPOP system [16] implements the IP protocol over a distributed P2P network overlay. Existing IP-based protocols can be implemented on top of the IPOP overlay with no additional modifications. IPOP uses the Brunet P2P library to handle P2P routing and NAT/firewall traversal. The WOW project [17] uses the IPOP system to implement self-organizing wide-area network overlays. It provides a user-level framework for generating such overlays and demonstrates its capabilities to process high-throughput sequential jobs in a grid environment. IPOP is also used by [28] to create self-provisioning and adaptable clusters. While these approaches have the reliability and scalability properties of peer-to-peer networking, they still incur overhead due to application-level processing of packets to maintain the network overlay topologies. Initial IPOP performance experiments added 6-10ms of latency to traffic in the network overlay [16].

Another approach to network overlays is to implement overlay components as virtual machines. VIOLIN [20] uses User Mode Linux (UML) virtual machines to implement virtual hosts, routers, and switches to create a virtual network topology. It uses a centralized manager to setup network links and node addresses. Each component is a separate, light-weight VM instance. This approach relies on the isolation provided by machine virtualization libraries to isolate user applications from the underlying physical resources. VIOLIN has already been used to create middleware for grid computing [34]. While VIOLIN uses light-weight VM instances to implement the overlay architecture, it still suffers from the performance overhead of overlay networks. Initial tests of VIOLIN on PlanetLab demonstrated a 5% degradation of TCP throughput over the underlying physical network [20].

[21] argues that overlay networks are too complicated and not necessary. It presents a single router abstraction to the end-user that attempts to avoid complicated overlay networks. The user

configures a single router to implement the desired network topology and the router abstraction handles the physical network infrastructure.

Virtual network topologies require application-level or VM level solutions. These solutions suffer from performance degradation due to the overhead of processing network packets. Some solutions require packets to traverse the kernel's network stack two times. The additional overhead adds latency to all network traffic in the overlay throughout the overlay's lifetime. In our OpenFlow approach, the cost overhead of dynamic networking is very small and short-lived. Initial tests indicate a 2-5 ms average increase in packet delay for one or two packets while OpenFlow rules are generated and installed. Once the OpenFlow rules are installed, future packet processing occurs at line-rate as it would on the underlying physical network, so there is no degradation of performance.

3.2 OpenFlow and the Cloud

Research into the uses of OpenFlow for networking within a cloud or grid environment has only recently increased in popularity. OpenFlow has been used to improve the performance of live migrations, load balancing, and network stitching.

Amazon's EC2 provides users with the ability to migrate instances and images between data center regions. [31] presents a migration model using Xen and OpenFlow. A VM is migrated from physical node to another and OpenFlow is used to re-route network traffic to the VM. The use of OpenFlow also removed the need for Xen to create/manage virtual networks. This implementation resulted in faster migrations with zero packet loss.

EC2 also provides users with Elastic Load Balancing to balance network traffic between multiple instances. [41] presents an OpenFlow-based load balancing solution. OpenFlow is used to partition the client IP address space and direct traffic from each partition to a specific instance. The algorithm minimizes the number of flow rules required to balance client traffic through the use of wildcard rules. The algorithm also adapts to changes in network traffic by re-partitioning the address space. It uses microflow rules to allow existing TCP connections to continue while it installs the new rules.

CloudNaaS [13] is the closest effort related to our work. CloudNaaS is a cloud networking platform for enterprise applications that uses OpenFlow for its network management. CloudNaaS (Cloud Networking-as-a-Service) extends the self-service provisioning model for cloud services to include networking services. The CloudNaaS Network Controller is used to provision virtual network segments between network devices that meet the constraints of the requested resources. It then uses OpenFlow to join the segments to create the desired path between resources. This network segment provisioning algorithm is used to guarantee quality of service (QoS) requirements and direct traffic through middleboxes placed in the network. The resulting path is also used when considering where to place a VM in the cloud to meet latency requirements. CloudNaaS also uses OpenFlow to allow applications to use custom addressing. OpenFlow rules installed on each cloud node using a software switch translate addresses from application addresses to cloud-assigned addresses.

While CloudNaaS presents a cloud environment that uses OpenFlow to enable networking services, it is not the same as our OneCloud solution. CloudNaaS mainly uses OpenFlow to stitch together network segments between two cloud resources. It also uses OpenFlow to enable applications to use custom addressing. However, it does not use OpenFlow to enable a range of customizable network services for end-users nor does it present an API for end-users to manage the networking environment. The networking environment is controlled by the attributes of the resources being provisioned. Additionally, CloudNaaS uses OpenNebula 1.4 for its provisioning engine, which is now obsolete. OneCloud is built using the latest version of OpenNebula and its features are being integrated into the source code.

Another network virtualization framework for the cloud is presented in [24]. This framework builds upon an abstract cloud model by creating an abstract representation of virtualized cloud networking. It discusses the use of OpenFlow to implement the virtualized networking framework in a real cloud environment, but does not provide an actual implementation. Furthermore, the framework only addresses the issue of custom network topologies in a cloud environment. It does not address dynamic networking features such as Elastic IP addresses, load balancing, or security groups.

3.3 Comparison of Cloud Solutions

There are numerous cloud solutions available to users. Table 3.1 compares popular solutions based on supported APIs, networking features, and their limitations. OpenStack, OpenStack Quantum (a Cisco Systems, Inc.-led initiative), and Eucalyptus all provide Elastic IP functionality. The OpenStack implementation relies on NAT translation provided by `iptables`, which can be

	API	**Networking Features**	**Limitations**
Amazon EC2 [9]	– EC2 Query – EC2 SOAP	– Elastic IP – Security Groups – VPC – Route Tables – VPN Connections – Route 53 DNS – Elastic Load Balancing	– Implementation unknown – Paid Service
Rackspace [32]	– Cloud Servers	– DDoS Mitigation – Persistent Public IPs – Shared IP Addresses – Cloud DNS – Cloud Load Balancers	– Implementation Unknown – No Elastic IPs – No Custom Network Topologies – Paid Service
OpenStack [4]	– EC2 Query – Cloud Servers	– Floating IPs – Network Models: VLAN, Flat, FlatDHCP	– No control over IP addressing – No custom topologies
OpenStack Quantum [7]	– EC2 Query – Cloud Servers	– Plugins for network implementations (SDNs, Overlays, etc) – QoS Guarantees – Firewall & Security Groups – VPC	– Still in Beta
OpenNebula [27][6]	– EC2 Query – EC2 SOAP – OCCI – XML-RPC	– Custom Subnets – Firewall – Host-Managed VLANs – Ebtables Support – Open vSwitch Support – Native VMware Networking Support	– No control over IP addressing – Firewall only supports iptables rules
Eucalyptus [15]	– EC2 Query – EC2 SOAP	– VM Network Isolation through VLAN – Elastic IP – Security Groups	– Cloud Controller maintains Elastic IP mappings, not networking hardware – Network Controller performs packet translations
CloudNaaS [13]	– EC2 Query – EC2 SOAP – OCCI – XML-RPC	– 1:1 NAT at Host nodes allowing apps to use existing address spaces – Guarantees QoS – Middlebox Interposition	– Implemented on outdated version of OpenNebula 1.4 – No API for controlling network – Experimental

Table 3.1: Comparison of Cloud Solutions

cumbersome to manage. The Eucalyptus network controller manages translating packets for Elastic IP addresses, which requires it to process all packets destined for that address. OpenStack Quantum provides plugins to use OpenFlow or other technologies for software-defined networking, however, it is still under development. The OneCloud approach will use a centralized controller to generate rules when a mapping is first created, but all packet modifications will occur on the individual network switches. OpenStack Quantum, OpenNebula, and Eucalyptus also provide implementations of Security Groups. The Quantum implementation is still under early development, but provides a pluggable infrastructure that can use `iptables` or OpenFlow. OpenNebula and Eucalyptus both install firewall rules on cloud nodes using `iptables`. The OneCloud solution will use OpenFlow rules to turn each network switch into a firewall for instances in the cloud.

Our survey of open-source cloud solutions (see Table 3.1) indicates that dynamic network services remained largely unavailable to cloud users. Amazon's EC2 is the leading provider of dynamic networking services, followed closely by RackSpace, but both require paid subscriptions. Eucalyptus provides Elastic IP and Security Groups functionality, but packets must be processed by the network controller. Clemson's OneCloud attempts to fill this void by enabling dynamic networking services in an open-source cloud platform through the use of OpenFlow. OpenFlow will provide a centralized management infrastructure for controlling the network flow within the cloud, but will use the distributed network resources to enforce the network policies. This will enable users to dynamically alter the networking environment in real time without impacting network performance. In addition, using OpenFlow can enhance other capabilities that are currently available in open-source cloud offerings, such as quality of service enforcement, DDoS mitigation, load balancing, and custom network topologies.

Chapter 4

Solution

4.1 OpenFlow Controller

The main component of the OpenFlow cloud is the OpenFlow controller (`http://code.google.com/p/onenox/`). The OpenFlow controller implements the network control logic, which determines how switches should route traffic within the cloud. OneCloud's OpenFlow controller is written in Python and built on top of the NOX controller platform. The OneCloud OpenFlow controller exposes an XML-RPC server to the OpenNebula front-end server. This enables the OpenNebula server to send commands to the OpenFlow controller requesting changes in the network environment. The controller maintains a set of data structures that describes the relationships between instances in the cloud, security groups to which they belong, and Elastic IP address associations. When it receives commands from OpenNebula, it constructs the appropriate OpenFlow rules and installs them on the networking hardware in the cloud.

The controller module registers event handlers with the NOX platform when it loads. It registers handlers that are called when a switch issues a join or leave event to the controller. This allows the controller to manage a list of the switches that it controls. In addition, it registers handlers for `packet_in` events, which are generated whenever a switch sends a packet to the controller for analysis. This allows the controller to analyze packets sent by the switches to determine where to send them.

Listing 4.1: OpenFlow Controller Layer 2 Switch Functionality

```python
def learnAndForward(self, dpid, inport, packet, buf, bufid):
   # Convert src and dst MAC addr to string
   mac_src = mac_to_str(packet.src)
   mac_dst = mac_to_str(packet.dst)

   # Get Switch data structure
   sw = self.SwitchesMap[dpid]

   # Learn the port for the source MAC addr
   sw.MacToPortMap[mac_src] = inport

   # If IP packet, save IP-to-MAC relationship
   iph = packet.find('ipv4')
   if iph != None:
      srcip = ip_to_str(iph.srcip)
      sw.IpToMacMap[srcip] = mac_src

   # If destination MAC of the packet is known
   # send packet out appropriate switch port
   if mac_dst in sw.MacToPortMap:
      outport = sw.MacToPortMap[mac_dst]

      # Get flow attributes
      flow = extract_flow(packet)
      flow[core.IN_PORT] = inport
      # Send packet on outport
      actions = [[openflow.OFPAT_OUTPUT, [0, outport]]]

      # install flow rule on the switch
      self.install_datapath_flow( dpid, flow, IDLE_TIMEOUT, HARD_TIMEOUT, actions
          , bufid, openflow.OFP_DEFAULT_PRIORITY, inport, buf)

      if iph!=None:
         # Check if src ip is one of the instances with a security group
         if srcip in self.instances:
            # Need to install temp rule to allow reply packets
            replyFlow = flow.copy()
            replyFlow[core.IN_PORT] = outport
            replyFlow[core.DL_DST] = flow[core.DL_SRC]
            replyFlow[core.DL_SRC] = flow[core.DL_DST]
            replyFlow[core.NW_DST] = flow[core.NW_SRC]
            replyFlow[core.NW_SRC] = flow[core.NW_DST]
            replyFlow[core.TP_DST] = flow[core.TP_SRC]
            replyFlow[core.TP_SRC] = flow[core.TP_DST]
            actions = [[openflow.OFPAT_OUTPUT, [0, inport]]]

            # install flow rule on the switch
            # this rule is temporary and will expire
            self.install_datapath_flow( dpid, replyFlow, IDLE_TIMEOUT,
                HARD_TIMEOUT, actions, None, SG_PRIORITY, None, None)
   else:
      # flood packet out everything but the input port
      self.send_openflow(dpid, bufid, buf, openflow.OFPP_FLOOD, inport)
```

4.1.1 Default Behavior

By default, the OneCloud OpenFlow controller behaves as a Layer 2 learning switch, forwarding packets based on their destination MAC address. Listing 4.1 presents the main logic in the learning switch. When the controller receives a packet, it saves the incoming port and source MAC address on line 10. This is how the switch learns which hosts are on which ports. The controller also learns the IP address associated with a packet's source MAC address in lines 13-16. This allows the controller to resolve the MAC address of the Elastic IP network gateway when installing flow rules for Elastic IP addresses.

The controller then decides how to forward the packet. If the packet's destination MAC is not known, the switch floods the packet out on all switch ports on line 51. If the destination host is connected to the switch, it will receive the packet. If the destination MAC is known, the controller creates a flow rule for the packet and installs the rule on the switch (lines 20-30). This ensures the switch will forward future packets instead of sending them to the controller.

Lines 34-48 in Listing 4.1 handle a special case for EC2 Security Groups. This code block generates temporary flow rules that allow response traffic to reach a virtual machine that belongs to a Security Group. We will discuss this in greater detail in subsection 4.3.4.

4.2 Elastic IP Addresses

OneCloud provides users with the capabilities of EC2 Elastic IP addresses by managing the network with the OpenFlow protocol. When an instance is launched in OneCloud, its network interface is attached to the Open vSwitch bridge on the host node. The OpenFlow controller installs rules on the bridge interface to manage the Elastic IP translations. In this section we discuss the EC2 interface for Elastic IP addresses, how the OpenFlow controller handles ARP Requests for Elastic IP addresses, and how the controller creates flow rules for an Elastic IP address.

4.2.1 EC2 Interface

OneCloud users can use the Amazon EC2 API to request and allocate Elastic IP addresses. Table 4.1 summarizes the Elastic IP EC2 API commands and their functions. Users can request Elastic IP addresses using the Amazon EC2 API command `AllocateAddress`. This command instructs the OpenNebula server to reserve an Elastic IP address from the address pool and associate

EC2 API Command	**Description**
`AllocateAddress`	Reserve Elastic IP address from pool
`AssociateAddress`	Map an Elastic IP to an instance
`DisassociateAddress`	Remove mapping of Elastic IP to an instance
`ReleaseAddress`	Return Elastic IP to address pool

Table 4.1: OneCloud Elastic IP EC2 Commands

it with a user's account. When the user no longer needs the Elastic IP address, the API command `ReleaseAddress` instructs the OpenNebula server to return the reserved address to the address pool. Elastic IP addresses are associated with an instance using API command `AssociateAddress`, which instructs the OpenFlow controller to install the flow rules. The command `DisassociateAddress` removes this associate and the corresponding flow rules.

4.2.2 Handling ARP Requests

Since an instance does not know if it has an Elastic IP associated with it, it cannot respond to ARP requests for the Elastic IP address. Therefore, the OpenFlow controller in OneCloud is responsible for sending ARP replies for Elastic IP addresses. Listing 4.2 contains an excerpt of the `packet_in` handler that examines ARP Requests. When an ARP request is sent out in the network, the switch on which it originated forwards the packet to the controller. The controller analyzes the packet and checks if it is an ARP request (line 4). If the packet is an ARP request, it retrieves the ARP Target Protocol Address field, which is the IP address of the intended receiver. On line 10, the controller checks if this IP address matches an Elastic IP address that is associated with an instance in the cloud. If it is, the controller generates an ARP reply message and sends it to the source of the request packet. If the target address is not an Elastic IP address, then the controller instructs the switch to flood the packet out on all switch ports so the appropriate host can reply.

Packet Field	Replaced with...
Source IP	Elastic IP
Source MAC	Open vSwitch bridge's MAC
Destination MAC	Elastic IP Gateway's MAC
VLAN ID	Elastic IP VLAN ID

Table 4.2: Outgoing Packet Field Modifications

Listing 4.2: OpenFlow Controller ARP Request Handling

```
arph = packet.find('arp')
if arph != None:
   # Found ARP Packet
   if arph.opcode == arp.REQUEST:
      # ARP Request
      # Get target IP address
      targetIp = ip_to_str(arph.protodst)

      # Check if target IP is an Elastic IP
      if targetIp in self.ElasticIpMap:
         log.error("Replying to ARP Request for IP "+targetIp)
         # Create ARP Reply packet and send it
         return self.sendArpReply(dpid, inport, packet)
```

4.2.3 Creating OpenFlow Rules

When the OpenNebula server receives the `AssociateAddress` command, it will instruct the OpenFlow controller to associate the Elastic IP address with the instance's private IP address. To do this, the OpenNebula server must provide the controller with information about the instance: private IP, MAC address, Open vSwitch bridge and port number to which the instance is attached, and the Elastic IP address. If the Elastic IP belongs to a VLAN, OpenNebula also provides the VLAN ID and gateway IP address of the Elastic IP subnet. When the OpenFlow controller receives this information, it creates a set of wildcard OpenFlow rules for incoming and outgoing IP packets to the instance. Tables 4.2 and 4.3 show the fields in the packet headers that will be modified by the OpenFlow rules for *outgoing* and *incoming* packets respectively and their new values after modification. The net effect of the installed rules is to make the Open vSwitch bridge act like a NAT router for the instances on the node.

Once these flow rules have been created, they are installed on the bridge to which the instance is attached. These rules are given a higher OpenFlow priority so that they will override any rules that were created by the basic layer 2 switch (the default behavior of the controller). The

Packet Field	Replaced with...
Destination IP	Private IP
Destination MAC	Instance's MAC
Source MAC	Open vSwitch bridge's MAC
VLAN ID	Instance's VLAN ID

Table 4.3: Incoming Packet Field Modifications

rules are installed permanently without soft or hard timeout values so they last the lifetime of the instance. When the Elastic IP is mapped to an instance, the OpenFlow controller assumes the responsibility of replying to ARP requests for the Elastic IP address.

Listing 4.3: Installing Incoming Elastic IP Flows

```
   def installIncomingElasticIpFlows(self, eip):
 2    # Get switch that virtual machine is attached to
      sw = self.SwitchesMap[eip.dpid]
      try:
         gatewayMac  = sw.IpToMacMap[eip.gatewayIp]
         gatewayPort = sw.MacToPortMap[gatewayMac]
 7    except KeyError:
         raise Exception("FAIL: Unable to install Incoming flows. Unknown Gateway
             MAC or Port Number")

      # Set matching criteria for this flow rule
      flow = {}
12    flow[core.DL_TYPE] = ethernet.IP_TYPE
      flow[core.NW_DST]  = eip.elasticIp
      flow[core.IN_PORT] = gatewayPort

      # Add actions to translate packet fields
17    actions = []
      actions.append([openflow.OFPAT_SET_NW_DST, eip.privateIp])
      actions.append([openflow.OFPAT_SET_DL_SRC, mac_to_str(eip.dpid)])
      actions.append([openflow.OFPAT_SET_DL_DST, eip.privateMac])
      actions.append([openflow.OFPAT_STRIP_VLAN])
22    actions.append([openflow.OFPAT_OUTPUT, [0, int(eip.port)]])

      # Install flow on switch with elevated priority
      # and add it to EIP data structure
      self.install_datapath_flow( eip.dpid, flow, openflow.OFP_FLOW_PERMANENT,
          openflow.OFP_FLOW_PERMANENT, actions, None, SG_PRIORITY, None, None)
27    eip.incomingFlows.append(flow)
```

Listing 4.3 contains the code that creates OpenFlow flow rules for *incoming* packets destined for an Elastic IP address. The controller first obtains the data structure that represents the switch on which the virtual machine is attached. It then tries to get the MAC address and switch port of the gateway host for the Elastic IP's subnet (lines 4-8). If it cannot acquire this information, then the controller will not know where to send the packet after it is processed. Lines 12-14 specify the attributes of packets that should match this flow rule, namely IP packets destined for the Elastic IP address. Lines 16-22 configure the actions that are applied to a packet that matches this flow rule. These actions translate the packet fields as defined in Table 4.3. Finally, the controller permanently installs the flow rule on the switch and saves the rule in the Elastic IP address's data structure so it can be uninstalled in the future. The controller function that generates flow rules for outgoing packets is very similar, except the actions translate the packet according to the rules in Table 4.2.

To remove an Elastic IP address mapping, the `DisassociateAddress` command is issued. This command instructs the OpenFlow controller to remove the installed flow rules and stop replying to ARP requests for the Elastic IP address. The code for `DisassociateAddress` can be found in Listing 4.4. When `DisassociateAddress` is called, the controller verifies the Elastic IP address is associated with an instance (line 3). It then removes all incoming and outgoing flow rules that were created for that address and removes the Elastic IP address from its data structures. Removing the installed flow rules is trivial, as seen in the `removeIncomingElasticIpFlows` method (lines 20-24). The installed flows for an Elastic IP are saved in an array, so the controller just iterates over the array and uninstalls each flow from the switch (lines 23-24).

Listing 4.4: Disassociate Elastic IP and Remove Flows

```
   def DisassociateAddress(self, elasticIp):
      # Verify EIP is actually associated with instance
3     if elasticIp in self.InstalledElasticIps:
         eip = self.InstalledElasticIps[elasticIp]

         # Remove incoming/outgoing flows that are installed
         # Calls removeIncomingElasticIpFlows() and removeOutgoingElasticIpFlows()
8        result = self.removeElasticIpFlows(elasticIp)

         # Remove EIP from data structures
         if result == 'SUCCESS':
            del self.ElasticIpMap[elasticIp]
13          del self.ElasticIpReverseMap[eip.privateIp]
            del self.InstalledElasticIps[elasticIp]

         return result
      else:
18       return 'SUCCESS'

   def removeIncomingElasticIpFlows(self, elasticIp):
      eip = self.InstalledElasticIps[elasticIp]
      # Remove all saved incoming flows for this EIP
23    for flow in eip.incomingFlows:
         self.delete_datapath_flow(eip.dpid, flow)
```

If a user issues the `AssociateAddress` command for an address that is already mapped to a running instance, the OpenFlow controller will disassociate the address from the current instance and install flow rules to map the address to the new instance. Similarly, if the instance already has an Elastic IP address mapped to it, the existing mapping will be removed before the new mapping is created. Also, if the user terminates an instance that is mapped to an Elastic IP address, the OpenFlow controller will remove any flow entries for that address.

4.3 Security Groups

OneCloud provides firewall services to users by implementing the Amazon EC2 Security Groups API. OneCloud's Security Groups implementation uses OpenFlow rules to manage network access to virtual machines. However, the OneCloud implementation currently does not support specifying another Security Group as the source of a traffic in a rule. In order to implement EC2 Security Groups, extensive modifications to the OpenNebula source code were made to support storing Security Groups in its database. This section describes these modifications, the EC2 Security Groups interface, converting Security Group rules to OpenFlow rules, and managing instances that belong to a Security Group.

4.3.1 OpenNebula Extensions

In order to implement EC2 Security Groups, it was necessary to extend the core OpenNebula source code. Modifications included adding objects for Security Groups and group rules. These extensions enabled OpenNebula to save information such as the owner of a group, user's permitted to use the group, and the group's rules in its database. The OpenNebula server already manages user access rights to other resources in the cloud, so it is natural for it to manage security groups. This also meant the OpenFlow controller did not have to store all of this information. This leads to a faster, streamlined controller that focuses solely on network management.

4.3.2 EC2 Interface

OneCloud users can use the Amazon EC2 API to create and manage Security Groups for their account. Table 4.4 lists the available commands and their functions. Security Groups are created and removed using the API commands `CreateSecurityGroup` and `DeleteSecurityGroup` respectively. Once a security group has been created, a user can add or remove rules for *incoming packets only.* The Amazon API does not permit outgoing packet rules for EC2 Security Groups.

EC2 API Command	**Description**
`CreateSecurityGroup`	Create a new Security Group
`DeleteSecurityGroup`	Delete an existing Security Group
`AuthorizeSecurityGroupIngress`	Add a new rule to a Security Group
`RevokeSecurityGroupIngress`	Remove a rule from a Security Group

Table 4.4: OneCloud Security Group EC2 Commands

A rule is added by issuing the command `AuthorizeSecurityGroupIngress` and removed with the command `RevokeSecurityGroupIngress`.

4.3.3 Generating OpenFlow Rules

If an instance is launched as part of one or more Security Groups, the OpenNebula server notifies the OpenFlow controller. The controller is provided with attributes about the launched instance and a list of its Security Groups and their rules. The controller converts each Security Group rule into a `Rule` object. Each object is responsible for converting the rule into the proper OpenFlow rule(s). When the controller installs the rules on a switch, it simply requests the OpenFlow rule(s) from the `Rule` object.

Conversion of the security group rules is fairly trivial. The `protocol` field is translated from the EC2 API constant to the corresponding NOX controller constant. If the rule is a TCP or UDP rule and specifies a port range, then an OpenFlow rule must be created for each port in that range. The port number is used as the destination port in the rule. If the protocol is ICMP, the `from port` becomes the ICMP `type` field and the `to port` becomes the ICMP `code` field. When these fields are converted, their values are stored in a Python dictionary object, creating a single flow rule. In the case of a range of ports, multiple flow rule dictionaries are created. These flow rules are stored in a list, which is then passed to the method `__createFlowsFromIprange` (see Listing 4.5).

Listing 4.5: Converting Source Field into Security Group Flows

```
def __createFlowsFromIpRange(self, flows):
   # Ip is 0.0.0.0
   if self.__ipRange == "0.0.0.0":
      log.error("Rule from 0.0.0.0")
      self.__flows = flows
      return

   # Ip addr range, i.e. 192.168.2.1-196.168.2.5
   m = re.match('((\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3}))([ ]*\-[ ]*)((\d
       {1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3}))$', self.__ipRange)
   if m != None:
      startIp = m.groups()[1:5]
      endIp   = m.groups()[7:11]
      # Enumerate IP addresses in the range
      for s4 in range(int(startIp[3]), int(endIp[3])+1):
         for flow in flows:
            f = flow.copy()
            f[NW_SRC] = ".".join([startIp[0],startIp[1],startIp[2],str(s4)])
            log.error(f[NW_SRC])
            self.__flows.append(f)
      return

   # Ip addr block using full Netmask 192.168.2.1/255.255.255.0
   m = re.match('((\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3}))(/)((\d{1,3})\.(\d
       {1,3})\.(\d{1,3})\.(\d{1,3}))$', self.__ipRange)
   if m != None:
      ip = m.groups()[0]
      # Convert netmask to number of bits set to 1
      netmask = int_to_bits(ipstr_to_int(m.groups()[6]))
      for flow in flows:
         f = flow.copy()
         f[NW_SRC] = ip
         # Number of bits in mask is opposite of standard CIDR notation
         f[NW_SRC_N_WILD] = 32 - netmask
         self.__flows.append(f)
      return

   # Ip addr block using CIDR 192.168.2.1/24
   m = re.match('(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/(\d{1,2})$', self.__ipRange
       )
   if m != None:
      ip = m.group(1)
      cidr = m.group(2)
      for flow in flows:
         flow[NW_SRC] = ip
         # Number of bits in mask is opposite of standard CIDR notation
         flow[NW_SRC_N_WILD] = 32 - int(cidr)
         self.__flows.append(flow)
      return

   # Single IP address
   if (re.match('\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$', self.__ipRange) != None):
      for flow in flows:
         flow[NW_SRC] = self.__ipRange
         self.__flows.append(flow)
      return
```

The final field, `source`, is more complicated to parse. The `source` can have one of several values: 0.0.0.0, a single IP address, or a range of IP addresses. Listing 4.5 contains the Python code that converts the `source` field into the proper OpenFlow flow rules. The code uses Python regular expressions to match the various formats of the `source` field. Lines 3-6 handle the simplest case: the value is *0.0.0.0*. In this case, a wildcard is used in the flow rule to match all source IP addresses. Lines 9-20 check if the value is an IP address range. If this is the case, the IP addresses within the range are enumerated because a flow rule must be created for each individual address. The next case to consider is if the source is an IP address and netmask. In this case, the controller determines the number of bits set to 1 in the netmask to be used in the OpenFlow rule. Lines 37-46 detect if the value is in CIDR notation. If so, the CIDR prefix length is used in the OpenFlow rule. Finally, the last case is checked in lines 49-53: the source is a single IP address. In this case, the IP address is used directly in the OpenFlow rule.

While some of the values of `source` generate a single flow rule, one may note that each case loops through an array named `flows`. This array contains partial flow rules that were generated based on the ports specified in the Security Group. If multiple ports were specified, then a flow rule for each port must be created. Similarly, if multiple source IP addresses were specified, multiple rules must be created. If both cases occur, a flow rule must be generated for every possible combination of port and source IP address. Therefore, the code iterates over the partial flow rules, creates all combinations of ports and source addresses, and stores them in the `Rule` object's class variable `__flows`.

4.3.4 Installing OpenFlow Rules for Security Groups

When an instance is launched, the user provides a list of security groups to which the instance should belong. After the instance has been deployed to a node in the cloud, the OpenNebula server notifies the OpenFlow controller that an instance has been started. It provides the controller with the instance's private IP address, the node on which the instance is launched, and the list of security groups and their rules. The OpenFlow controller then converts the security group rules into OpenFlow rules before installing them on the switch.

Once the rules are converted to the OpenFlow format, the controller must install the rules on the switch. The default behavior for EC2 Security Groups is to block all new incoming traffic, allow all outgoing traffic from an instance, and allow related incoming network traffic. Therefore, the controller installs a default rule that drops all incoming packets to the instance (we will refer to this as the DROP rule), as seen in Listing 4.6 (lines 1-14). The OpenFlow specification does not have an explicit DROP action. Instead, an empty `actions` list is passed to `install_datapath_flow` to implicitly drop matching packets (line 7). This rule is given an elevated priority over existing rules on the switch. The rules for a security group are then installed on the switch with a priority higher than the DROP rule. This ensures all packets matching the incoming rules will be allowed and all other packets will be dropped by the switch. This occurs in the method `installIncomingSecurityGroupFlows` on lines 16-32 of Listing 4.6, which iterates over an instance's Security Groups and installs each groups' rules on the switch. The DROP rule and all incoming rules are installed permanently on the switch and removed only when an instance is terminated.

Listing 4.6: Installing the DROP Rule

```
    def installDefaultSecurityGroupFlows(self, instance):
 2     log.error("Installing default security group flows")
       sw = self.SwitchesMap[instance.dpid]

       flow = {}
       # NO actions == DROP packet
 7     actions = []

       flow[core.DL_TYPE] = ethernet.IP_TYPE
       flow[core.DL_DST]  = instance.privateMac
       flow[core.NW_DST]  = instance.privateIp
12
       self.install_datapath_flow( instance.dpid, flow, openflow.OFP_FLOW_PERMANENT,
           openflow.OFP_FLOW_PERMANENT, actions, None, SG_DROP_PRIORITY, None, None)
       instance.incomingFlows.append(flow)

    def installIncomingSecurityGroupFlows(self, instance):
17     sw = self.SwitchesMap[instance.dpid]

       # Iterate over instance's security groups
       for gid in instance.groups:
          # Iterate over each rule in the group
22        for rule in instance.groups[gid].rules():
             for flow in rule.getFlows():
                actions = []
                # Add instance specific attributes to rule
                flow[core.DL_TYPE] = ethernet.IP_TYPE
27              flow[core.DL_DST]  = instance.privateMac
                flow[core.NW_DST]  = instance.privateIp
                actions.append([openflow.OFPAT_OUTPUT, [0, int(instance.port)]])

                self.install_datapath_flow( instance.dpid, flow, openflow.
                    OFP_FLOW_PERMANENT, openflow.OFP_FLOW_PERMANENT, actions, None,
                    SG_PRIORITY, None, None)
32              instance.incomingFlows.append(flow)
```

When an instance generates outgoing traffic, the switch installs a temporary rule allowing it to reach the network. However, the current rules for the security group may not allow the response traffic to reach the instance. Therefore, when the outbound rule is generated, the controller installs a corresponding temporary rule for the response packet. The code responsible for this action is found in the learning switch code (see lines 34-48 in Listing 4.1) because it analyzes all packets sent to the controller by the switch and creates flow rules for them. This rule is given a priority higher than the DROP rule to ensure the packet is allowed to reach the instance. Unlike the incoming rules for the security group, this rule does not have wildcard values. It matches only the flow created by the outbound traffic. Once the outbound flow of traffic stops, the idle timeout on the incoming flow rule expires and it is removed from the switch.

4.3.5 Modifying Security Group Rules

The Amazon EC2 specification [9] allows users to add or remove rules to a security group and have the changes automatically applied to all instances in the group. OneCloud also provides this capability to its users. When a user adds a rule to a security group, the OpenNebula server notifies the OpenFlow controller of the new rule with the XML-RPC command `AuthorizeSecurityGroupIngress`. Listing 4.7 contains the source code for this method.

The OpenFlow controller generates a list of instances that belong to the modified security group on line 5. For each instance, it adds the new rule to the instance's local copy of the Security Group. Once the rule is added, the generated flow rules are installed on the switch (lines 13-22). In a similar manner, if the user removes a rule from a security group, the OpenFlow controller will uninstall the corresponding flow rules for each instance from their respective switches.

Listing 4.7: Authorizing New Security Group Rules

```
def AuthorizeSecurityGroupIngress(self, sgid, protocol, fromPort, toPort, ipRange
    ):
   # Check that an instance belongs to this group
   if str(sgid) in self.groupsInstancesMap:
      # Get list of instances in the group
      instances = self.groupsInstancesMap[str(sgid)]
      for privateIp in instances:
         log.error("Updating instance %s" %(privateIp))
         instance = self.instances[privateIp]
         group = instance.groups[str(sgid)]
         # Add the new rule to the group
         group.addRule(Rule(protocol, fromPort, toPort, ipRange))
         # Get the flows created for the NEW RULE ONLY
         flows = group.rules()[-1].getFlows()
         # Install only the flows created by the new rule
         for flow in flows:
            actions = []
            flow[core.DL_TYPE] = ethernet.IP_TYPE
            flow[core.DL_DST]  = instance.privateMac
            flow[core.NW_DST]  = instance.privateIp
            actions.append([openflow.OFPAT_OUTPUT, [0, int(instance.port)]])
            self.install_datapath_flow( instance.dpid, flow, openflow.
                OFP_FLOW_PERMANENT, openflow.OFP_FLOW_PERMANENT, actions, None,
                SG_PRIORITY, None, None)
            instance.incomingFlows.append(flow)
```

Chapter 5

Results

5.1 Elastic IP Addresses

In order to test our Elastic IP implementation, we recorded network traffic between instances on Clemson's OneCloud and a client on the Clemson network. We used `ping` to generate traffic because it provides measurements on packet loss and packet round trip time (RTT) between the client and instance. It is important to note that the Elastic IP address pool in OneCloud is assigned a specific VLAN ID for Clemson's core network. Therefore, our implementation must also add and remove VLAN tags as needed during the packet modification. Also of importance is the fact that our OpenFlow controller's default behavior is a learning switch. So, before any Elastic IP addresses are mapped to an instance, the instance's network traffic will appear to originate from node within our cloud using Network Address Translation (NAT). The Elastic IP rules installed by our OpenFlow controller are given an elevated priority to override any rules that exist from the default learning switch behavior.

There are several scenarios of interest when testing the functionality of our Elastic IP implementation. The most basic case is mapping a single Elastic IP address to a single virtual machine instance in the cloud. To ensure compatibility with Amazon's EC2 Elastic IP addresses, we must also consider two other cases. If we have an Elastic IP address mapped to an instance, we have a scenario where a user may remap the Elastic IP to another instance in the cloud. Additionally, if we have an address mapped to an instance, a user may decide to map a new Elastic IP address to that same instance. This section presents the results of these scenarios in the Clemson OneCloud.

5.1.1 Scenario 1: Mapping Elastic IP to Single Host

The most common scenario for Elastic IPs occurs when a user maps a single Elastic IP address to a single instance in the cloud. Prior to this mapping, the instance did not have an Elastic IP address mapped to it nor did the Elastic IP address map to another instance in the cloud.

In this scenario, the instance has a private IP address of 10.10.1.50. The Elastic IP address to be mapped to the instance is 130.127.38.236. The instance is sending packets to a client whose IP address is 130.127.39.58. Since our OpenFlow controller behaves as a learning switch by default, we note that the IP address of the cloud node hosting the virtual machine instance is 130.127.39.10.

Listing 5.1 shows the OpenFlow rules installed on the cluster node to apply the Elastic IP to incoming and outgoing IP traffic. The first rule rewrites the source IP address of outgoing traffic from 10.10.1.50 to 130.127.38.236 using the action `mod_nw_src`. It also replaces the source MAC address with the MAC address of the cluster node using the action `mod_dl_src`. This ensures reply packets will find the correct node in the cloud upon return. It also replaces the destination MAC address of the packet with the MAC address of the Elastic IP VLAN's gateway to ensure proper routing within the campus network.

Listing 5.1: Elastic IP Scenario 1: OpenFlow Rules

```
priority=32778,ip,in_port=8,nw_src=10.10.1.50 actions=mod_nw_src:130.127.38.236,
    mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:00:17:df:2b:08:00,output:1
priority=32778,ip,in_port=1,nw_dst=130.127.38.236 actions=mod_nw_dst:10.10.1.50,
    mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:02:00:0a:0a:01:32,strip_vlan,output:8
```

The second rule rewrites incoming traffic's destination IP address from 130.127.38.236 to 10.10.1.50 using the action `mod_nw_dst`. It also replaces the destination MAC address with the MAC address of the virtual machine instance using the action `mod_dl_dst`. Since our Elastic IP address is also VLAN tagged, the rule uses the action `strip_vlan` to remove the VLAN tag from the incoming packet before sending it to the instance. The last action in the OpenFlow rule, `output`, directs the switch to send the packet out on the port on which the instance is attached.

Listing 5.2 shows the output from `ping` during our scenario. The session statistics indicate that the instance had 0% packet loss when the Elastic IP address mapping was established and then removed. Also, line 7 indicates a spike in the packet delay from less than 1 ms to 13.5 ms. This is due to the time it takes the OpenFlow controller to install the rules on the switch, the switch to re-route the network traffic, and the client to send ARP Requests for the Elastic IP. After this initial spike in delay, the packet delay is noticeably higher (lines 8-14) because our Elastic IP belongs to a VLAN on the campus network. Therefore, it is routed through the VLAN gateway server, whereas before we mapped the Elastic IP address, the packets were only routed through the switch connecting the cluster node and the client. When we removed the Elastic IP mapping, the next packet sent had an ICMP sequence number of 13 (see line 25 in Listing 5.3). A corresponding spike in the packet delay is seen on line 15 as our OpenFlow controller installs the new flow rules. The `ping` output shows the packet delay returning to values below 1 ms on lines 16-18, indicating the packet returned to its original route.

Listing 5.2: Elastic IP Scenario 1: `ping` Output

```
   root@ec2-vm-589:~# ping 130.127.39.58
 2 PING 130.127.39.58 (130.127.39.58) 56(84) bytes of data.
   64 bytes from 130.127.39.58: icmp_req=1 ttl=63 time=68.5 ms
 4 64 bytes from 130.127.39.58: icmp_req=2 ttl=63 time=3.81 ms
   64 bytes from 130.127.39.58: icmp_req=3 ttl=63 time=0.604 ms
 6 64 bytes from 130.127.39.58: icmp_req=4 ttl=63 time=0.685 ms
   64 bytes from 130.127.39.58: icmp_req=5 ttl=63 time=13.5 ms
 8 64 bytes from 130.127.39.58: icmp_req=6 ttl=63 time=3.16 ms
   64 bytes from 130.127.39.58: icmp_req=7 ttl=63 time=3.05 ms
10 64 bytes from 130.127.39.58: icmp_req=8 ttl=63 time=3.38 ms
   64 bytes from 130.127.39.58: icmp_req=9 ttl=63 time=2.97 ms
12 64 bytes from 130.127.39.58: icmp_req=10 ttl=63 time=3.14 ms
   64 bytes from 130.127.39.58: icmp_req=11 ttl=63 time=3.05 ms
14 64 bytes from 130.127.39.58: icmp_req=12 ttl=63 time=2.93 ms
   64 bytes from 130.127.39.58: icmp_req=13 ttl=63 time=15.0 ms
16 64 bytes from 130.127.39.58: icmp_req=14 ttl=63 time=0.690 ms
   64 bytes from 130.127.39.58: icmp_req=15 ttl=63 time=0.627 ms
18 64 bytes from 130.127.39.58: icmp_req=16 ttl=63 time=0.660 ms

20 --- 130.127.39.58 ping statistics ---
   16 packets transmitted, 16 received, 0% packet loss, time 15010ms
22 rtt min/avg/max/mdev = 0.604/110.967/1317.517/321.744 ms, pipe 2
```

Listing 5.3 contains the `tcpdump` trace of our `ping` session. The traffic is originating from the instance and `tcpdump` is collecting the traffic on the client machine. Without the Elastic IP address, the virtual machine traffic is filtered through NAT on the cloud node. Initially, the ICMP Echo Request packets originate from 130.127.39.10, our cloud node's network interface (see lines 1-8). After we associate the Elastic IP address with the instance, the packets appear to originate from 130.127.38.236, our Elastic IP address. This is seen in lines 9-24 in network trace. We then disassociate the Elastic IP address and the packets originate from our cluster node again (lines 25-32). The trace results show a continuous flow of ICMP packets between the instance and the physical host as evidenced by the continuous sequence numbers for the same ICMP identifier value.

Listing 5.3: Elastic IP Scenario 1: `tcpdump` Output

```
22:47:20.493454 IP 130.127.39.10 > 130.127.39.58: ICMP echo request,id 1348,seq 1
22:47:20.493514 IP 130.127.39.58 > 130.127.39.10: ICMP echo reply,id 1348,seq 1
22:47:21.459825 IP 130.127.39.10 > 130.127.39.58: ICMP echo request,id 1348,seq 2
22:47:21.459870 IP 130.127.39.58 > 130.127.39.10: ICMP echo reply,id 1348,seq 2
22:47:22.458633 IP 130.127.39.10 > 130.127.39.58: ICMP echo request,id 1348,seq 3
22:47:22.458683 IP 130.127.39.58 > 130.127.39.10: ICMP echo reply,id 1348,seq 3
22:47:23.457816 IP 130.127.39.10 > 130.127.39.58: ICMP echo request,id 1348,seq 4
22:47:23.457896 IP 130.127.39.58 > 130.127.39.10: ICMP echo reply,id 1348,seq 4
22:47:24.464819 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1348,seq 5
22:47:24.464892 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1348,seq 5
22:47:25.459463 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1348,seq 6
22:47:25.459488 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1348,seq 6
22:47:26.460872 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1348,seq 7
22:47:26.460952 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1348,seq 7
22:47:27.462211 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1348,seq 8
22:47:27.462296 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1348,seq 8
22:47:28.463693 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1348,seq 9
22:47:28.463731 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1348,seq 9
22:47:29.464951 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1348,seq 10
22:47:29.465029 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1348,seq 10
22:47:30.466285 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1348,seq 11
22:47:30.466366 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1348,seq 11
22:47:31.465821 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1348,seq 12
22:47:31.465880 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1348,seq 12
22:47:32.479222 IP 130.127.39.10 > 130.127.39.58: ICMP echo request,id 1348,seq 13
22:47:32.479290 IP 130.127.39.58 > 130.127.39.10: ICMP echo reply,id 1348,seq 13
22:47:33.466915 IP 130.127.39.10 > 130.127.39.58: ICMP echo request,id 1348,seq 14
22:47:33.466993 IP 130.127.39.58 > 130.127.39.10: ICMP echo reply,id 1348,seq 14
22:47:34.465882 IP 130.127.39.10 > 130.127.39.58: ICMP echo request,id 1348,seq 15
22:47:34.465939 IP 130.127.39.58 > 130.127.39.10: ICMP echo reply,id 1348,seq 15
22:47:35.465831 IP 130.127.39.10 > 130.127.39.58: ICMP echo request,id 1348,seq 16
22:47:35.465908 IP 130.127.39.58 > 130.127.39.10: ICMP echo reply,id 1348,seq 16
```

5.1.2 Scenario 2: Transferring an Elastic IP Between Hosts

Another common scenario for Elastic IPs is remapping the Elastic IP address to a failover instance. In this scenario, an Elastic IP is mapped to one virtual machine instance in the cloud. The user then maps the same Elastic IP address to a different instance in the cloud. End-users should not notice a significant disruption in the service provided by the instances.

In this scenario, we have two instances, I1 and I2, with private IP addresses of 10.10.1.50 and 10.10.1.51 respectively. The Elastic IP address to be mapped between the instances is 130.127.38.236. The client generating traffic to the instance has an IP address of 130.127.39.58.

Listing 5.4 shows the two OpenFlow rules that were installed when the Elastic IP address was mapped to I1 (lines 2-3), and the two new rules installed after the Elastic IP address was mapped to I2 (lines 5-6). When the mapping changed, the first two rules (lines 2-3) were removed. The first rule on line 2 modifies outgoing packets, setting their source IP address to the Elastic IP address. The second rule on line 3 modifies incoming packets and rewrites their destination IP address to the instance's private IP address. When the Elastic IP is mapped to I2, the private IP address used in the rules changes to 10.10.1.51. Also, the action `mod_dl_dst` on line 6 now changes the destination MAC address of the packet to the MAC address of I2.

Listing 5.4: Elastic IP Scenario 2: OpenFlow Rules

```
1 # Rules for Elastic IP mapped to Instance 1
2 priority=32778,ip,in_port=8,nw_src=10.10.1.50 actions=mod_nw_src:130.127.38.236,
      mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:00:17:df:2b:08:00,output:1
3 priority=32778,ip,in_port=1,nw_dst=130.127.38.236 actions=mod_nw_dst:10.10.1.50,
      mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:02:00:0a:0a:01:32,strip_vlan,output:8
4 # Rules for Elastic IP mapped to Instance 2
5 priority=32778,ip,in_port=9,nw_src=10.10.1.51 actions=mod_nw_src:130.127.38.236,
      mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:00:17:df:2b:08:00,output:1
6 priority=32778,ip,in_port=1,nw_dst=130.127.38.236 actions=mod_nw_dst:10.10.1.51,
      mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:02:00:0a:0a:01:33,strip_vlan,output:9
```

Listing 5.5: Elastic IP Scenario 2: `ping` Output

```
bash-3.2$ ping 130.127.38.236
PING 130.127.38.236 (130.127.38.236): 56 data bytes
64 bytes from 130.127.38.236: icmp_seq=0 ttl=63 time=15.563 ms
64 bytes from 130.127.38.236: icmp_seq=1 ttl=63 time=4.471 ms
64 bytes from 130.127.38.236: icmp_seq=2 ttl=63 time=2.970 ms
64 bytes from 130.127.38.236: icmp_seq=3 ttl=63 time=3.014 ms
64 bytes from 130.127.38.236: icmp_seq=4 ttl=63 time=3.300 ms
64 bytes from 130.127.38.236: icmp_seq=5 ttl=63 time=2.968 ms
64 bytes from 130.127.38.236: icmp_seq=6 ttl=63 time=4.475 ms
64 bytes from 130.127.38.236: icmp_seq=7 ttl=63 time=2.899 ms
64 bytes from 130.127.38.236: icmp_seq=8 ttl=63 time=2.933 ms
64 bytes from 130.127.38.236: icmp_seq=9 ttl=63 time=3.266 ms
64 bytes from 130.127.38.236: icmp_seq=10 ttl=63 time=2.920 ms
64 bytes from 130.127.38.236: icmp_seq=11 ttl=63 time=3.290 ms
64 bytes from 130.127.38.236: icmp_seq=12 ttl=63 time=3.134 ms
64 bytes from 130.127.38.236: icmp_seq=13 ttl=63 time=5.176 ms
64 bytes from 130.127.38.236: icmp_seq=14 ttl=63 time=3.377 ms
64 bytes from 130.127.38.236: icmp_seq=15 ttl=63 time=3.249 ms
64 bytes from 130.127.38.236: icmp_seq=16 ttl=63 time=3.492 ms
64 bytes from 130.127.38.236: icmp_seq=17 ttl=63 time=5.177 ms

--- 130.127.38.236 ping statistics ---
18 packets transmitted, 18 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 2.899/4.204/15.563/2.851 ms
```

Listing 5.5 shows the output from `ping` during our scenario. The session statistics indicate that the client experienced 0% packet loss when the Elastic IP address was remapped to I2 and then back to I1. Before the ping session begins, the Elastic IP is mapped to instance I1. After `ping` sends packet 5, the Elastic IP address is remapped to I2. Line 9 shows a small spike in the round-trip delay of the next packet. The Elastic IP address is remapped back to instance I1 after packet 12 is sent. The round-trip delay of the next packet indicates a small spike on line 16. These spikes correspond to the overhead required to remap the Elastic IP address and re-route network traffic.

Listing 5.6: Elastic IP Scenario 2: `tcpdump` Output

```
   # Captured on Instance 1
 2 23:50:28.800251 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 0
   23:50:28.800281 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 0
 4 23:50:29.796307 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 1
   23:50:29.796333 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 1
 6 23:50:30.794960 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 2
   23:50:30.794989 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 2
 8 23:50:31.795226 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 3
   23:50:31.795259 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 3
10 23:50:32.795599 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 4
   23:50:32.795625 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 4
12 23:50:33.795365 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 5
   23:50:33.795396 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 5
14 23:50:42.796956 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 14
   23:50:42.796985 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 14
16 23:50:43.797241 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 15
   23:50:43.797271 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 15
18 23:50:44.797660 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 16
   23:50:44.797687 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 16
20 23:50:45.799463 IP 130.127.39.58 > 10.10.1.50: ICMP echo request,id 25411,seq 17
   23:50:45.799487 IP 10.10.1.50 > 130.127.39.58: ICMP echo reply,id 25411,seq 17
22 # Captured on Instance 2
   23:50:35.164414 IP 130.127.39.58 > 10.10.1.51: ICMP echo request,id 25411,seq 6
24 23:50:35.164450 IP 10.10.1.51 > 130.127.39.58: ICMP echo reply,id 25411,seq 6
   23:50:36.163047 IP 130.127.39.58 > 10.10.1.51: ICMP echo request,id 25411,seq 7
26 23:50:36.163077 IP 10.10.1.51 > 130.127.39.58: ICMP echo reply,id 25411,seq 7
   23:50:37.163218 IP 130.127.39.58 > 10.10.1.51: ICMP echo request,id 25411,seq 8
28 23:50:37.163247 IP 10.10.1.51 > 130.127.39.58: ICMP echo reply,id 25411,seq 8
   23:50:38.163722 IP 130.127.39.58 > 10.10.1.51: ICMP echo request,id 25411,seq 9
30 23:50:38.163754 IP 10.10.1.51 > 130.127.39.58: ICMP echo reply,id 25411,seq 9
   23:50:39.163486 IP 130.127.39.58 > 10.10.1.51: ICMP echo request,id 25411,seq 10
32 23:50:39.163516 IP 10.10.1.51 > 130.127.39.58: ICMP echo reply,id 25411,seq 10
   23:50:40.164016 IP 130.127.39.58 > 10.10.1.51: ICMP echo request,id 25411,seq 11
34 23:50:40.164050 IP 10.10.1.51 > 130.127.39.58: ICMP echo reply,id 25411,seq 11
   23:50:41.164054 IP 130.127.39.58 > 10.10.1.51: ICMP echo request,id 25411,seq 12
36 23:50:41.164081 IP 10.10.1.51 > 130.127.39.58: ICMP echo reply,id 25411,seq 12
   23:50:42.166206 IP 130.127.39.58 > 10.10.1.51: ICMP echo request,id 25411,seq 13
38 23:50:42.166233 IP 10.10.1.51 > 130.127.39.58: ICMP echo reply,id 25411,seq 13
```

Listing 5.6 contains the `tcpdump` traces of our `ping` session between the client and the Elastic IP address. The traffic originates from the client and `tcpdump` is collecting the traffic on instances I1 (lines 2-21) and I2 (lines 23-38). Lines 2-13 show that I1 received six packets from our client at 130.127.39.58. Note that the IP address for I1 in the trace is its private IP address, 10.10.1.50. This is because it does not know that an Elastic IP address has been mapped to it. After line 13, we observe a break in the ICMP sequence numbers of the packets received by I1. This is because the Elastic IP address has been remapped to I2. I2 received eight packets on lines 23-38 from the client machine. The private IP address in the network trace is 10.10.1.51, which is the IP address of I2. After packet 13 is received, the Elastic IP address is remapped back to I1. Lines 14-21 show that I1 received the remaining packets sent by the client to the Elastic IP address. Note that between the two network traces, all of the packets sent by the client were received and acknowledged by one of the two instances.

5.1.3 Scenario 3: Assigning New Elastic IP to Host With Elastic IP

The final scenario discussed for Elastic IP addresses is assigning a new Elastic IP address to an instance that already has an Elastic IP address mapped to it. In this scenario, the new Elastic IP address will replace the current mapping because only one Elastic IP can be mapped to a single instance.

We have a single instance with a private IP address of 10.10.1.50. We have been allocated two Elastic IP addresses from the address pool, 130.127.38.236 and 130.127.38.237. We will refer to these as E1 and E2 respectively. Traffic will be directed from the instance to a client with an IP address of 130.127.39.58 and the network traffic is recorded using `tcpdump` on the client.

Listing 5.7: Elastic IP Scenario 3: OpenFlow Rules

```
1 # Rules when E1 is installed
2 priority=32778,ip,in_port=8,nw_src=10.10.1.50 actions=mod_nw_src:130.127.38.236,
      mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:00:17:df:2b:08:00,output:1
3 priority=32778,ip,in_port=1,nw_dst=130.127.38.236 actions=mod_nw_dst:10.10.1.50,
      mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:02:00:0a:0a:01:32,strip_vlan,output:8
4 # Rules when E2 is installed
5 priority=32778,ip,in_port=8,nw_src=10.10.1.50 actions=mod_nw_src:130.127.38.237,
      mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:00:17:df:2b:08:00,output:1
6 priority=32778,ip,in_port=1,nw_dst=130.127.38.237 actions=mod_nw_dst:10.10.1.50,
      mod_dl_src:00:1a:a0:1d:ba:81,mod_dl_dst:02:00:0a:0a:01:32,strip_vlan,output:8
```

Listing 5.7 shows the flow rules installed during our scenario. When E1 is mapped to the instance, two flow rules are installed (lines 2-3). The first rule modifies outgoing packets by changing the instance's private IP address, 10.10.1.50, to E1. The second rule converts the destination IP address from E1, 130.127.38.236, to that of the instance's private IP address, 10.10.1.50. When E2 is mapped to the instance, the rules for E1 are removed from the switch. Then the rules for E2 (lines 5-6) are installed on the switch. These rules are identical to the original rules, except the Elastic IP address is now that of E2.

Listing 5.8: Elastic IP Scenario 3: `ping` Output

```
   root@ec2-vm-589:~# ping 130.127.39.58
 2 PING 130.127.39.58 (130.127.39.58) 56(84) bytes of data.
   64 bytes from 130.127.39.58: icmp_req=1 ttl=63 time=12.2 ms
 4 64 bytes from 130.127.39.58: icmp_req=2 ttl=63 time=6.29 ms
   64 bytes from 130.127.39.58: icmp_req=3 ttl=63 time=2.93 ms
 6 64 bytes from 130.127.39.58: icmp_req=4 ttl=63 time=3.00 ms
   64 bytes from 130.127.39.58: icmp_req=5 ttl=63 time=19.2 ms
 8 64 bytes from 130.127.39.58: icmp_req=6 ttl=63 time=2.94 ms
   64 bytes from 130.127.39.58: icmp_req=7 ttl=63 time=3.44 ms
10 64 bytes from 130.127.39.58: icmp_req=8 ttl=63 time=3.24 ms
   64 bytes from 130.127.39.58: icmp_req=9 ttl=63 time=3.24 ms
12 64 bytes from 130.127.39.58: icmp_req=10 ttl=63 time=2.97 ms
   64 bytes from 130.127.39.58: icmp_req=11 ttl=63 time=2.87 ms
14 64 bytes from 130.127.39.58: icmp_req=12 ttl=63 time=11.2 ms
   64 bytes from 130.127.39.58: icmp_req=13 ttl=63 time=6.22 ms
16 64 bytes from 130.127.39.58: icmp_req=14 ttl=63 time=2.99 ms
   64 bytes from 130.127.39.58: icmp_req=15 ttl=63 time=4.48 ms
18
   --- 130.127.39.58 ping statistics ---
20 15 packets transmitted, 15 received, 0% packet loss, time 14019ms
   rtt min/avg/max/mdev = 2.872/5.825/19.264/4.629 ms
```

Listing 5.8 shows the output from `ping` during our scenario. The session statistics indicate that the instance had 0% packet loss when the new Elastic IP address, E2, was mapped to the instance. Before the ping session begins, the Elastic IP E1 is mapped to the instance. E2 is mapped to the instance after packet 4 has been sent to the client. Line 7 shows a large spike in the packet delay from 3.00 ms to 19.2 ms. This corresponds to the time it takes to uninstall the old Elastic IP mapping, install the new mapping, and for the ARP cache to update on the client machine. The packet delay seen in lines 8-13 are stable values averaging 3.11 ms. Then the address E1 is mapped back to the instance and the E2 mapping is removed. Line 14 shows another large spike in the packet delay when this remapping occurs.

The `tcpdump` trace of our `ping` session is contained in Listing 5.9. The traffic is originating from the instance and `tcpdump` is collecting the traffic on the client machine. Initially, the address E1 is mapped to the instance, so the client sees traffic originating from 130.127.38.236 (see lines 1-8). We then map address E2 to the instance. The controller recognizes that the instance already has an address mapped to it and removes any associated flow rule entries. It then installs the new flow rules on the switch to map E2 to the instance. This can be observed on line 9 when the source of the ICMP traffic changes to 130.127.38.237, or E2. Note that the ICMP Identification number remains the same. The traffic continues to originate from E2 until we remap E1 to the instance.

Lines 23-30 indicate the source address of the traffic is back to E1.

Listing 5.9: Elastic IP Scenario 3: `tcpdump` Output

```
 1  23:23:52.859807 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1430,seq 1
    23:23:52.859870 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1430,seq 1
 3  23:23:53.859938 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1430,seq 2
    23:23:53.860019 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1430,seq 2
 5  23:23:54.857920 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1430,seq 3
    23:23:54.857999 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1430,seq 3
 7  23:23:55.859103 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1430,seq 4
    23:23:55.859184 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1430,seq 4
 9  23:23:56.867188 IP 130.127.38.237 > 130.127.39.58: ICMP echo request,id 1430,seq 5
    23:23:56.867267 IP 130.127.39.58 > 130.127.38.237: ICMP echo reply,id 1430,seq 5
11  23:23:57.861837 IP 130.127.38.237 > 130.127.39.58: ICMP echo request,id 1430,seq 6
    23:23:57.861907 IP 130.127.39.58 > 130.127.38.237: ICMP echo reply,id 1430,seq 6
13  23:23:58.862934 IP 130.127.38.237 > 130.127.39.58: ICMP echo request,id 1430,seq 7
    23:23:58.863014 IP 130.127.39.58 > 130.127.38.237: ICMP echo reply,id 1430,seq 7
15  23:23:59.864618 IP 130.127.38.237 > 130.127.39.58: ICMP echo request,id 1430,seq 8
    23:23:59.864710 IP 130.127.39.58 > 130.127.38.237: ICMP echo reply,id 1430,seq 8
17  23:24:00.866064 IP 130.127.38.237 > 130.127.39.58: ICMP echo request,id 1430,seq 9
    23:24:00.866143 IP 130.127.39.58 > 130.127.38.237: ICMP echo reply,id 1430,seq 9
19  23:24:01.867464 IP 130.127.38.237 > 130.127.39.58: ICMP echo request,id 1430,seq 10
    23:24:01.867533 IP 130.127.39.58 > 130.127.38.237: ICMP echo reply,id 1430,seq 10
21  23:24:02.868574 IP 130.127.38.237 > 130.127.39.58: ICMP echo request,id 1430,seq 11
    23:24:02.868635 IP 130.127.39.58 > 130.127.38.237: ICMP echo reply,id 1430,seq 11
23  23:24:03.876051 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1430,seq 12
    23:24:03.876127 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1430,seq 12
25  23:24:04.875417 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1430,seq 13
    23:24:04.875489 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1430,seq 13
27  23:24:05.873506 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1430,seq 14
    23:24:05.873590 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1430,seq 14
29  23:24:06.874735 IP 130.127.38.236 > 130.127.39.58: ICMP echo request,id 1430,seq 15
    23:24:06.874800 IP 130.127.39.58 > 130.127.38.236: ICMP echo reply,id 1430,seq 15
```

5.1.4 Summary

This section presented three scenarios where Elastic IP addresses were mapped to instances in the cloud. The most common scenario occurs when a user maps a new Elastic IP address to an instance without an Elastic IP address mapped to it already. Another common scenario occurs when a user must remap an Elastic IP to another instance in the cloud as in the case with failovers. The final scenario presented occurs when a user maps a new Elastic IP address to an instance that already has an Elastic IP address mapped to it. In this case, the old mapping is removed and the new mapping is installed by the OpenFlow controller.

Because our Elastic IP implementation uses the OpenFlow controller, there is minimal delay in associating an Elastic IP address with a running instance. The flow rules are installed on the cloud node's bridge interface and do not have to propagate throughout the cloud. Once the rules are installed, all packet modification occurs at line-rate on the networking hardware avoiding latency due to centralized processing. This behavior is seen in the observed packet delay in the `ping` output. Packets experience a brief spike in round trip time when new flow rules are installed. After the initial spike, the network stabilizes quickly.

5.2 Security Groups

In order to test our Security Groups implementation, we recorded network traffic between instances on Clemson's OneCloud and a client on the Clemson network. We used the `netcat` application to make connections to our instances from clients. `netcat` was chosen because of its ability to make connections using both TCP and UDP transport protocols without the overhead of any application protocols (SSH, HTTP, etc).

There are several scenarios of interest when testing the functionality of our Security Groups implementation. Since security groups allow users to specify the source of incoming traffic using a variety of formats, it is important to consider how these formats affect the resulting OpenFlow rules that are generated. Also, we must demonstrate that these rules do in fact prevent applications from connecting to an instance unless the security group permits it. Additionally, an instance can belong to multiple Security Groups. Therefore, it is important to observe how the rules from each group are aggregated together for an instance. This section presents the results of these scenarios in the Clemson OneCloud.

5.2.1 OpenFlow Rules Resulting from Various `source` Formats

As previously discussed, the `source` field of a Security Group rule has multiple valid formats. The first format allows network traffic from any source IP address, a value of 0.0.0.0. When 0.0.0.0 is given as the source, the OpenFlow rule's source IP field is set to a wildcard to match all IP addresses. Listing 5.10 shows the resulting rule when TCP traffic on port 22 is allowed from any IP address. The rule does not display the field `nw_src` because it is wildcarded.

Listing 5.10: Security Group Source Formats: Allow All

```
priority=32778,tcp,dl_dst=02:00:0a:0a:01:36,nw_dst=10.10.1.54,tp_dst=22 actions=
    output:5
```

Sometimes it is desirable to specify a range of IP addresses that are allowed access. For example, an administrator may want to restrict TCP traffic on port 22 to an instance by only allowing the traffic to originate from an IP address in the range 10.10.1.2 - 10.10.1.5. In this case, the OpenFlow controller must enumerate the range of IP addresses and create a rule for each address in the range. Listing 5.11 contains the four resulting OpenFlow rules if the `source` field is set to 10.10.1.2-10.10.1.5. The rules are identical except for the field `nw_src`.

Listing 5.11: Security Group Source Formats: Allow IP Range

```
priority=32778,tcp,dl_dst=02:00:0a:0a:01:36,nw_src=10.10.1.5,nw_dst=10.10.1.54,
    tp_dst=22 actions=output:5
priority=32778,tcp,dl_dst=02:00:0a:0a:01:36,nw_src=10.10.1.4,nw_dst=10.10.1.54,
    tp_dst=22 actions=output:5
priority=32778,tcp,dl_dst=02:00:0a:0a:01:36,nw_src=10.10.1.2,nw_dst=10.10.1.54,
    tp_dst=22 actions=output:5
priority=32778,tcp,dl_dst=02:00:0a:0a:01:36,nw_src=10.10.1.3,nw_dst=10.10.1.54,
    tp_dst=22 actions=output:5
```

In most cases, the network access is restricted to traffic originating from a specific subnet. A subnet can be specified using the full netmask, such as 10.10.1.0/255.255.255.0, or using CIDR notation such as 10.10.1.0/24. In the former case, the controller must convert the full netmask to CIDR notation. Both cases result in the same rule being generated, as seen in Listing 5.12. The OpenFlow protocol supports specifying subnets for source and destination IP addresses, so the controller does not need to enumerate the addresses in the subnet.

The final case for the `source` field is a single IP address. In this case, the controller simply sets the value of `nw_src` in the OpenFlow rule to the IP address specified. This can be seen in Listing 5.13. The Security Group rule created allows TCP traffic on port 22 from the IP address 10.10.1.32.

Listing 5.12: Security Group Source Formats: Allow IP Subnet

```
priority=32778,tcp,dl_dst=02:00:0a:0a:01:36,nw_src=10.10.1.0/24,nw_dst=10.10.1.54,
    tp_dst=22 actions=output:5
```

Allowing different formats for the `source` field of a Security Group rule provides greater flexibility when granting network access to an instance. The OneCloud OpenFlow controller can correctly generate OpenFlow rules for any of the valid `source` formats, as illustrated by the examples presented.

Listing 5.13: Security Group Source Formats: Allow Single IP

```
priority=32778,tcp,dl_dst=02:00:0a:0a:01:36,nw_src=10.10.1.32,nw_dst=10.10.1.54,
    tp_dst=22 actions=output:5
```

5.2.2 Enabling Access to an Instance by Adding Security Group Rules

This scenario addresses the use case where an instance has been started in the cloud and belongs to one or more Security Groups, none of which allow access for a particular application. To simplify this example, the instance will belong to a single Security Group, however, the process is identical for membership in multiple Security Groups.

Listing 5.14: Security Group Scenario 1: OpenFlow Rules

```
# Before enabling 8080
priority=32773,ip,dl_dst=02:00:0a:0a:01:34,nw_dst=10.10.1.52 actions=drop
priority=32778,tcp,dl_dst=02:00:0a:0a:01:34,nw_dst=10.10.1.52,tp_dst=80 actions=
    output:10

# After enabling 8080
priority=32773,ip,dl_dst=02:00:0a:0a:01:34,nw_dst=10.10.1.52 actions=drop
priority=32778,tcp,dl_dst=02:00:0a:0a:01:34,nw_dst=10.10.1.52,tp_dst=8080 actions=
    output:10
priority=32778,tcp,dl_dst=02:00:0a:0a:01:34,nw_dst=10.10.1.52,tp_dst=80 actions=
    output:10
```

Suppose a network administrator has created a Security Group to protect a web server. This group only allows HTTP traffic on TCP port 80 to reach the virtual machine. One of the web developers requests access to TCP port 8080 for development work on the web server. The administrator must now modify the rules for the Security Group to allow the new traffic.

Listing 5.14 shows the OpenFlow rules installed by the controller when the instance is first launched in the cloud (lines 2-3) and after the new rule has been added (lines 6-8). The first rule on line 2 is the DROP rule that blocks all incoming traffic to the instance. This rule has an elevated priority, 32773, so that it takes precedence over the rules installed by the controller's default behavior, a learning switch. The next rule on line 3 allows TCP traffic on port 80 to reach the instance. Any TCP traffic destined for the instance's IP and MAC address on port 80 will be forwarded by the switch out port 10, where the instance resides. This rule has a priority of 32778, which is higher than the DROP rule. This ensures packets that match this rule are forwarded by the switch and not dropped. After the administrator enables traffic for TCP port 8080, a new flow rule is installed (see line 7). This rule is identical to the rule on line 8, except the TCP port is now 8080.

Listing 5.15: Security Group Scenario 1: `tcpdump` Output

```
   00:34:15.362712 IP 10.10.1.1.52380 > 10.10.1.52.80: Flags[S],seq 2072315589
 2 00:34:15.377504 IP 10.10.1.52.80 > 10.10.1.1.52380: Flags[S.],seq 2695841706,ack
       2072315590
   00:34:15.377531 IP 10.10.1.1.52380 > 10.10.1.52.80: Flags[.],ack 1
 4 00:34:18.161429 IP 10.10.1.1.52380 > 10.10.1.52.80: Flags[P.],seq 1:7,ack 1
   00:34:18.161683 IP 10.10.1.52.80 > 10.10.1.1.52380: Flags[.],ack 7
 6 00:34:20.305407 IP 10.10.1.1.52380 > 10.10.1.52.80: Flags[F.],seq 7,ack 1
   00:34:20.305747 IP 10.10.1.52.80 > 10.10.1.1.52380: Flags[F.],seq 1,ack 8
 8 00:34:20.305779 IP 10.10.1.1.52380 > 10.10.1.52.80: Flags[.],ack 2
   00:34:28.962676 IP 10.10.1.1.56313 > 10.10.1.52.8080: Flags[S],seq 2292193873
10 00:34:31.962320 IP 10.10.1.1.56313 > 10.10.1.52.8080: Flags[S],seq 2292193873
   00:34:37.962324 IP 10.10.1.1.56313 > 10.10.1.52.8080: Flags[S],seq 2292193873
12 00:35:27.778677 IP 10.10.1.1.56315 > 10.10.1.52.8080: Flags[S],seq 3209508960
   00:35:27.792671 IP 10.10.1.52.8080 > 10.10.1.1.56315: Flags[S.],seq 3841054552,ack
       3209508961
14 00:35:27.792697 IP 10.10.1.1.56315 > 10.10.1.52.8080: Flags[.],ack 1
   00:35:30.817437 IP 10.10.1.1.56315 > 10.10.1.52.8080: Flags[P.],seq 1:7,ack 1
16 00:35:30.817716 IP 10.10.1.52.8080 > 10.10.1.1.56315: Flags[.],ack 7
   00:35:34.481346 IP 10.10.1.1.56315 > 10.10.1.52.8080: Flags[F.],seq 7,ack 1
18 00:35:34.481672 IP 10.10.1.52.8080 > 10.10.1.1.56315: Flags[F.],seq 1,ack 8
   00:35:34.481708 IP 10.10.1.1.56315 > 10.10.1.52.8080: Flags[.],ack 2
```

A `tcpdump` trace of network traffic is shown in Listing 5.15. The `tcpdump` trace was recorded on the cloud node's physical ethernet port so that all packets could be observed before the switch dropped any of them. Lines 1-8 show a session between the client and the instance using TCP port 80. The client connected to the instance on port 80 using `netcat`, sent a short message, and then closed the TCP connection. The client then attempts to connect to the instance on port 8080, which is currently blocked. Lines 9-11 show three TCP SYN packets sent to the instance attempting to establish a TCP connection port 8080. The client never receives an ACK packet from the instance indicating it wants to complete the TCP handshake. The administrator then adds the new rule allowing TCP traffic on port 8080 to reach the instance. The controller installs the new rule on the switch and the client tries to connect again. Lines 12-14 indicate that the client sent a TCP SYN packet on port 8080, received an ACK packet from the instance, and the completed the TCP handshake, establishing a TCP connection. Lines 15-19 show the client sending a brief message to the instance and then terminating the TCP connection.

5.2.3 An Instance Belongs to Multiple Security Groups

In most cases, an instance will belong to multiple Security Groups, each group allowing a particular type of access (i.e. ICMP, HTTP, SSH, etc). In these cases, the OpenFlow controller must aggregate all of the Security Group rules for an instance and install them in the switch connected to the instance.

Suppose we have three different Security Groups named *pingGroup*, *httpGroup*, and *sshGroup*. The pingGroup allows ICMP Echo Request and ICMP Echo Reply packets to reach the instance so you can check for network connectivity using `ping`. The httpGroup allows users to connect to the instance over TCP port 80 from any source IP address. The sshGroup allows users to login to the instance using the SSH protocol on TCP port 22 from any source IP address. The pingGroup consists of two rules, one for requests and another for replies, and the other two groups each have a single rule.

To test the rule aggregation, we start an instance in the cloud that belongs to all three Security Groups. As seen in Listing 5.16, the OpenFlow controller has installed five OpenFlow rules governing traffic to our instance. The first rule is the DROP rule that prevents any network traffic from reaching the instance. The rule on line 2 is the result of the httpGroup. It allows TCP traffic on port 80 to reach the instance. The sshGroup's rule allowing TCP traffic on port 22 is rule seen on line 3 in the listing. The last two rules correspond to the pingGroup's rules: one allowing Echo Request packets and the other Echo Reply packets.

Anyone who sees the result of the aggregated rules would assume the instance belongs to a single Security Group that allows HTTP, SSH, and ICMP traffic to the instance. However, the Security Groups feature allows administrators to create groups with a single purpose (web server, remote login, etc) and combine the required groups for a particular instance. This simplifies the management of network security in the cloud and allows users to adjust security requirements for an instance when it is launched.

Listing 5.16: Security Group Scenario 2: OpenFlow Rules

```
priority=32773,ip,dl_dst=02:00:0a:0a:01:35,nw_dst=10.10.1.53 actions=drop
priority=32778,tcp,dl_dst=02:00:0a:0a:01:35,nw_dst=10.10.1.53,tp_dst=80 actions=
    output:4
priority=32778,tcp,dl_dst=02:00:0a:0a:01:35,nw_dst=10.10.1.53,tp_dst=22 actions=
    output:4
priority=32778,icmp,dl_dst=02:00:0a:0a:01:35,nw_dst=10.10.1.53,icmp_type=8,
    icmp_code=0 actions=output:4
priority=32778,icmp,dl_dst=02:00:0a:0a:01:35,nw_dst=10.10.1.53,icmp_type=0,
    icmp_code=0 actions=output:4
```

5.2.4 Summary

This section presented two scenarios where instances were placed in Security Groups in the cloud. In the first scenario, an instance was started in a Security Group. The group was modified to allow incoming traffic for an application after the instance was already started. The second scenario demonstrated how the OpenFlow controller aggregates rules from multiple Security Groups for a single instance in the cloud. This section also shows the resulting OpenFlow rules generated from the various formats of the `source` field in a Security Group rule.

Because our Security Groups implementation uses the OpenFlow controller, there is minimal delay when adding or removing rules from a Security Group. Firewall services do not have to be restarted and the network connectivity of the virtual machine remains available the entire time. Also, since the network hardware manages the security, there is no longer a bottleneck at a single firewall appliance. The firewall functionality is distributed across the network devices.

Chapter 6

Conclusions and Future Work

We presented a virtualized networking solution for the cloud using the OpenFlow protocol on Clemson's OneCloud. This work analyzes the potential of OpenFlow to enable dynamic networking in cloud computing and presents reference implementations of Amazon EC2's Elastic IP Addresses and Security Groups using the NOX OpenFlow controller and the OpenNebula cloud provisioning engine. It differs from current research in that it does not use network overlays or route all packets through a single server, which can lead to degraded network performance. While other cloud offerings, such as OpenStack and Eucalyptus, provide Elastic IP and Security Group services, OneCloud uses the network hardware to enable such services instead of a centralized network server. CloudNaaS and other cloud implementations have used OpenFlow in the cloud, but its use is limited to isolating networks with VLAN tagging. OneCloud uses OpenFlow to control network traffic routing and enforce higher-level security rules that cannot be implemented with VLAN tagging.

The solution presented requires some modifications to improve performance and reliability and be fully compliant with the EC2 specification. The current Elastic IP implementation terminates existing TCP connections when a mapping is established. When the mapping for an Elastic IP is removed in the current implementation, all flow rules are removed from the switch immediately. Likewise, if the user is assigning a new Elastic IP to an instance, all existing flow rules for the instance are removed immediately, causing existing connections to terminate abruptly. Extensions to the current implementation will include the installation of microflow rules for existing flows that timeout once the flow has stopped generating network traffic. This would allow current connections to remain alive until the user has closed the network connection. Amazon's EC2 Security Groups allows users to specify another security group as the source of network traffic. Our current implementation only allows users to specify IP addresses as the source of incoming traffic. Future work will investigate adding these missing features to the Security Groups implementation in OneCloud.

Currently, work is being done to analyze the performance of our reference implementations and measure their scalability. Initial scalability testing will be conducted using Clemson's Palmetto Cluster, a high-performance computing cluster. Tests will measure the performance of our OpenFlow controller and the networking overhead of our solution. Future scalability testing will investigate deploying Clemson OneCloud to the GENI research infrastructure using GENI Racks. GENI Racks are self-contained racks of compute, storage, and networking hardware accompanied by GENI management software. These racks enable organizations to easily join the GENI infrastructure and deploy meso-scale experiments on distributed, heterogeneous resources. OneCloud will provision resources across the GENI infrastructure to conduct at-scale cloud networking research.

Large-scale cloud environments consist of multiple data centers, known as regions or zones, to minimize network latency and maximize reliability by isolating failures to specific data centers. However, this creates barriers for cloud users' resources. Instances launched within one region do not have layer 2 connectivity to instances launched in a separate region. This also presents problems if users wish to live migrate a virtual machine instance to another node because the node must be located in the same data center. OpenNebula and KVM support live virtual machine migration, however, the disk image must be on a shared file system that can be accessed by all of the cloud nodes. Future research will focus on the ability to perform live migrations between separate physical data centers. For OpenNebula, this means migrating instances between Virtual Data Centers (VDCs) by extending the OpenNebula Zones Server. We will also investigate re-routing live network traffic via OpenFlow to fail-over instances located in different data centers.

Bibliography

[1] Flowvisor. http://flowvisor.org, Nov 2011.

[2] Geni: Exploring networks of the future. http://www.geni.net, Nov 2011.

[3] Nox. http://www.noxrepo.org/, 9 2011.

[4] Openstack open source cloud computing software. http://www.openstack.org, Nov 2011.

[5] Open vswitch. http://openvswitch.org/, February 2012.

[6] Opennebula home page. http://www.opennebula.org, January 2012.

[7] Openstack quantum. http://wiki.openstack.org/Quantum, 3 2012.

[8] Amazon. Overview of amazon web services. White paper, December 2010.

[9] Inc. Amazon. User guide for amazon elastic compute cloud. http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/, Feb 2012.

[10] Inc. Amazon. What is aws? http://aws.amazon.com/what-is-aws/, March 2012.

[11] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. *SIGCOMM Comput. Commun. Rev.*, 32(1):66–66, January 2002.

[12] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, 2009.

[13] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 8. ACM, 2011.

[14] B.J. Brodkin, C. Computing, et al. Gartner: Seven cloud-computing security risks. *Infoworld*, pages 2–3, 2008.

[15] Inc. Eucalyptus Systems. Cloud computing software from eucalyptus. http://www.eucalyptus.com/, 3 2012.

[16] A. Ganguly, A. Agrawal, P.O. Boykin, and R. Figueiredo. Ip over p2p: enabling self-configuring virtual ip networks for grid computing. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 10 pp., april 2006.

[17] A. Ganguly, A. Agrawal, P.O. Boykin, and R. Figueiredo. Wow: Self-organizing wide area overlay networks of virtual workstations. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 30–42. IEEE, 2006.

[18] A. Ganguly, D. Wolinsky, P.O. Boykin, and R. Figueiredo. Decentralized dynamic host configuration in wide-area overlays of virtual workstations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8, march 2007.

[19] Dell Inc. Dynamic insertion of services in a multi-tenant virtual data center. http://opennetsummit.org/demonstrations.html, Oct 2011.

[20] X. Jiang and D. Xu. Violin: Virtual internetworking on overlay infrastructure. *Parallel and Distributed Processing and Applications*, pages 937–946, 2005.

[21] E. Keller and J. Rexford. The platform as a service model for networking. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 4–4. USENIX Association, 2010.

[22] Radware Ltd. Scalable dos attack detection and mitigation. http://opennetsummit.org/demonstrations.html, Oct 2011.

[23] P. Marshall, K. Keahey, and T. Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52. IEEE Computer Society, 2010.

[24] J. Matias, E. Jacob, D. Sanchez, and Y. Demchenko. An openflow based network virtualization framework for the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 672 –678, 29 2011-dec. 1 2011.

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[26] P. Mell and T. Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800:145, 2011.

[27] Dejan Miloji andi and, Ignacio M. Llorente, and Ruben S. Montero. Opennebula: A cloud management tool. *Internet Computing, IEEE*, 15(2):11 –14, march-april 2011.

[28] M.A. Murphy, L. Abraham, M. Fenn, and S. Goasguen. Autonomic clouds on the grid. *Journal of Grid Computing*, 8(1):1–18, 2010.

[29] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pages 124–131. IEEE, 2009.

[30] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. *Proc. HotNets (October 2009)*, 2009.

[31] Pedro Pisa, Natalia Fernandes, Hugo Carvalho, Marcelo Moreira, Miguel Campista, Lus Costa, and Otto Duarte. Openflow and xen-based virtual network migration. In Ana Pont, Guy Pujolle, and S. Raghavan, editors, *Communications: Wireless in Developing Countries and Networks of the Future*, volume 327 of *IFIP Advances in Information and Communication Technology*, pages 170–181. Springer Boston, 2010. 10.1007/978-3-642-15476-8-17.

[32] RackSpace. Cloud computing, managed hosting, dedicated server hosting by rackspace. http://www.rackspace.com/, 3 2012.

[33] S. Ramgovind, M.M. Eloff, and E. Smith. The management of security in cloud computing. In *Information Security for South Africa (ISSA), 2010*, pages 1 –7, aug. 2010.

[34] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual distributed environments in a shared infrastructure. *Computer*, 38(5):63–69, 2005.

[35] Santiago@AWS. Feature guide: Amazon ec2 elastic ip addresses. http://aws.amazon.com/articles/1346, July 2010.

[36] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14–22, 2009.

[37] Ananth I. Sundararaj and Peter A. Dinda. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[38] Ananth I. Sundararaj, Ashish Gupta, and Peter A. Dinda. Dynamic topology adaptation of virtual networks of virtual machines. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, LCR '04, pages 1–8, New York, NY, USA, 2004. ACM.

[39] M. Tsugawa and J.A.B. Fortes. A virtual network (vine) architecture for grid computing. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[40] M. Tsugawa and J.A.B. Fortes. Characterizing user-level network virtualization: performance, overheads and limits. *International Journal of Network Management*, 20(3):149–166, 2010.

[41] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[42] David Isaac Wolinsky, Yonggang Liu, and Renato Figueiredo. Towards a uniform self-configuring virtual private network for workstations and clusters in grid computing. In *Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, VTDC '09, pages 19–26, New York, NY, USA, 2009. ACM.

[43] Y. Xin, I. Baldine, A. Mandal, C. Heermann, J. Chase, and A. Yumerefendi. Embedding virtual topologies in networked clouds. In *Proceedings of the 6th International Conference on Future Internet Technologies*, pages 26–29. ACM, 2011.