A Walk in the Clouds: A New PaaS Model Leveraging Software-Defined Networks

Michael Ben-Ami
Internet Realtime Lab
Department of Computer Science
Columbia University
New York, NY
Email: mzb2106@columbia.edu

I. Introduction and Background

This project was started as a previous student semester project by Etan Zapinsky and Alex Merkulov[1]. In their paper, Zapinsky and Merkulov sought to simplify the interface between cloud PaaS providers and customers (developers) by minimizing the dependence on arbitrary provider-specific configuration, and also to streamline configuration within the provider's environment. They proposed a container-based cloud leveraging Software-Defined Networking (SDN) to create an M-to-N mapping between containers and IP addresses. In this model, applications are identified by a [public destination IP address, public TCP/UDP port] tuple, which can be associated with one or more containers, and IP addresses can be shared among containers running different applications. As part of their project, Zapinsky and Merkulov also implemented a Container Lifecycle Management system, with an HTTP API and a CLI utility.

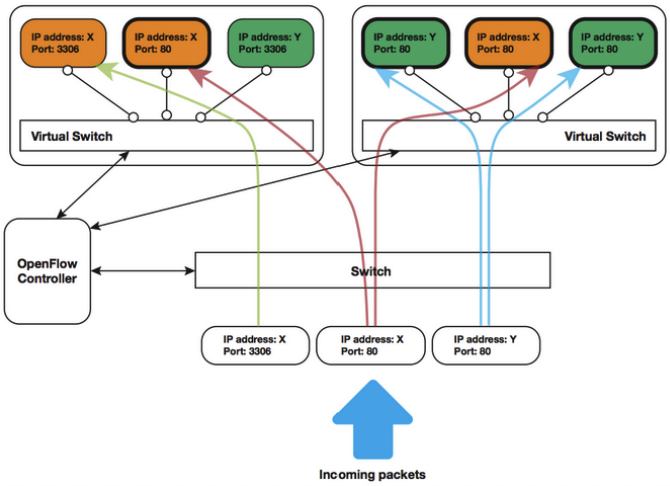

Fig. 1. Original Logical Goal

Zapinsky's architecture consisted of a single Linux VM, running an instance of Open vSwitch (OVS)[2], and one or more Docker[3] containers running inside of the VM, connected to the OVS. The OVS was controlled by a Pyretic[4] OpenFlow[5] controller, running inside another VM. When a packet came in to OVS from the outside world, the [public destination IP, port] tuple was mapped to a private IP address, and OVS performed a NAT. Containers in this scenario could serve clients from the outside world, but couldn't act as clients themselves or initiate connections. There were other components in the original architecture used for container lifecycle management, but those are not in focus here. My tasks in the continuation of the project included:

- to implement a new mapping: [public IP, port] to [destination MAC]
  - eliminates the NAT step
  - gives a container a single IP identifier, both "outside" and "inside"
- to implement container-originated communication
- to expand the architecture from single to multiple VMs
- to adapt the deployment to run on top of a public IaaS
- to focus on network operations, as opposed to container lifecycle management
- to contribute to an industry workshop paper

II. Architecture

A. Single VM - Phase 0

In the original (Zapinsky) architecture, a single Open vSwitch instance is controlled by a Pyretic controller programmed with with routing capabilities. The project modifications to the controller code introduce the NAT rules according to the original mapping. After the NAT, the OVS performs a normal layer 2 destination rewrite based on the new (private) destination IP address. If the destination container is on a local subnet, the destination MAC address will be that of the container. If the destination is on another subnet, the destination MAC address will be that of the appropriate next hop router.

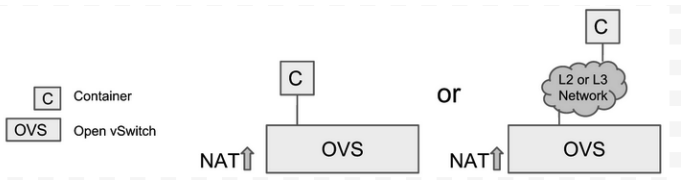

Fig. 2. Original Single VM Architecture

B. Single VM - Phase 1

In our new architecture, the single Open vSwitch instance is controlled by a POX controller programmed with layer 2 switching capabilities. The modifications to the controller code introduce layer 2 MAC rewrite rules according to the new mapping. This means that the destination containers must be in the same layer 2 domain as the OVS. The containers are programmed to have a default gateway residing on the OVS. Because our POX controller is layer 2 by nature, and doesn't recognize the concept of layer 3 interfaces, it is programmed with a "fake" default gateway. This "fakeway" replies to ARP requests from the containers with the MAC address of an interface on the host VM in a different subnet, but in the same layer 2 broadcast domain as the containers. Packets leaving the containers would then have this interface's MAC address as the destination, and OVS would forward them to that interface.

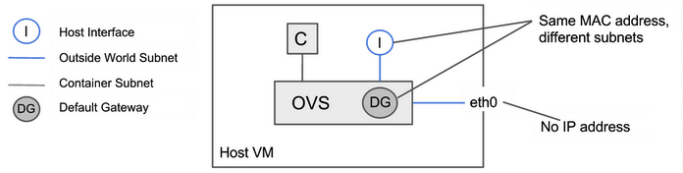

Fig. 3. Single VM Phase 1 Architecture

C. Single VM - Phase 2

As the project progressed, a more elegant architecture emerged in which the containers' default gateway is simply provided by an interface on the host VM that is on the same layer 3 subnet and layer 2 broadcast domain as the containers. This interface routes traffic received from containers to the outside world. ARP requests from the containers and responses from the default gateway are allowed to pass through OVS. OVS responds to ARP requests from the default gateway with an arbitrary MAC address so that if two containers were to have the same IP address, there would be no flapping in the host's ARP table.

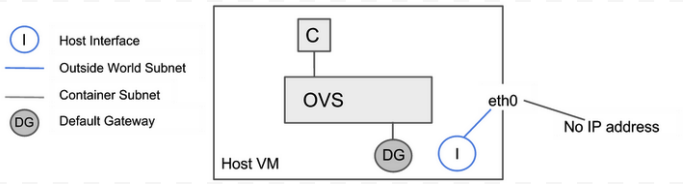

Fig. 4. Single VM Phase 2 Architecture

D. Mutliple VMs, Multiple Virtual Switches

While testing on a single VM was sufficient for a basic proof-of-concept, we decided that implementing a multiple-VM environment was necessary for real-world viability. In this new design, each host VM runs a single instance of Open vSwitch. Each instance of OVS can have one or more containers attached to it. As in the other phases, all OVS instances are controlled by a centralized OpenFlow controller through an out-of-band management network.

Expanding the design to multiple Virtual Machines presented new challenges. If we think about the OVS instances as the "access layer" networking for containers, questions arise as to the "fabric" that interconnects the access layer. Should this fabric be layer 2 or layer 3 in nature? Is it necessary for every hop along the way to be participate in our SDN control plane? The answer to the first question is that, yes, it should be a layer 2 fabric that forwards packets based on destination MAC address. This is because when an incoming packet hits one of the SDN-controlled switches, the MAC address will be re-written based on our special mapping. If the packet were then to reach a layer 3 boundary on the way to its destination, this new MAC address would be erased and rewritten again. Furthermore, forwarding at layer 3 would complicate the original goal of allowing containers to share IP addresses. The answer to the second question is more undecided. On the surface it would seem that a normal (non-SDN-enabled) layer 2 fabric would have no problem forwarding based on MAC address after our special rewrite, since MAC addresses of containers are still unique. However, some non-SDN network equipment may implement mechanisms that thwart our goals. For example, some layer 2 switches implement security mechanisms enforcing strict IP-to-MAC 1-to-1 mappings to prevent spoofing[6]. So, ideally all nodes that make up the fabric should participate in the SDN control plane.

E. Moving to a Public Cloud

Because so much of the original work was a response to the weaknesses of dotCloud, and how it functioned on top of Amazon's AWS, we wanted to deploy our multiple-VM design on top of a public IaaS. Furthermore, a proven deployment on top of a public IaaS would provide a repeatable model for others to adopt with little capital expenditure.

We chose to deploy on Amazon AWS. In this scenario, each instance of a VM (and corresponding OVS switch) maps to an Amazon EC2 instance. The fabric connecting the VMs is Amazon's internal networking infrastructure, which we have no control over, or visibility into. Although it seems this infrastructure can act as a basic layer 2 switched network, and forward our packets with re-written destination MAC addresses, it doesn't. Instead, Amazon will drop packets if the destination MAC address and IP address don't match the user-supplied configuration. Since our deployment involves an IP address being shared among multiple MAC addresses, we were forced to find alternatives.

Our solution was to use layer 2 overlay tunnels to connect our VMs through the Amazon network, specifically VXLAN[15], as it and GRE are the only options supported on Open vSwitch. In this way, all layer 2 frames to or from our VMs are encapsulated with new layer-3 and layer-2 headers routable over the Amazon network. This design has the added benefit of providing isolation of our addresses from Amazon's

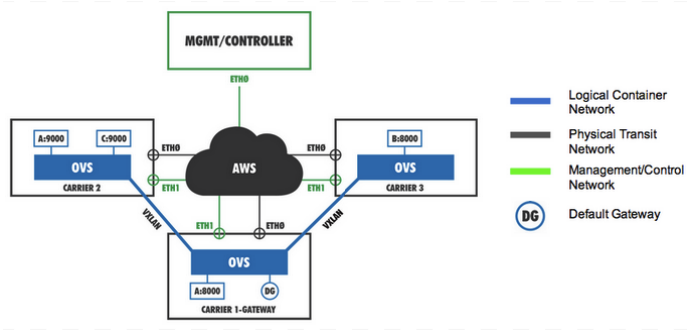

Fig. 5. Architecture over Amazon AWS

addresses, as the inner IP and MAC need not be examined by Amazon. It also allows for the underlying network to cross layer 3 boundaries. Two containers can be geographically distant, in different Amazon zones and subnets, and still be logically layer 2 adjacent to one another.

The "Amazon addresses" mentioned above refer to private IP addresses assigned to EC2 instances. Each of these addresses can be, in turn, mapped to an an Amazon public address routable over the Internet. Amazon provides the NAT upon entry into their network, and we have no control over that process. Because one of the original project goals was for the containers' IP addresses to be real public IP addresses, we dedicated a "gateway" EC2 instance to NAT private addresses back to their original public addresses, before being directed to the correct container on the appropriate OVS. This gateway EC2 instance can also host OVS and containers, and it acts as the single point of entry from the outside world into our environment. From Amazon's point of view, all destination hosts actually live on this gateway. Amazon is unaware that once traffic hits the gateway, it is NAT'd back to its original public IP address, and might then be VXLAN-encapsulated and sent to a different EC2 instance. In a real-world scenario, public IP addresses can be obtained from a third-party source, and NAT on the gateway wouldn't be necessary. There would still need to be a gateway, or multiple gateways, between the outside world and our environment, the main purpose of which would be to advertise container routes to the Internet. Independence from Amazon-supplied public IPs also allow for future IPv6 implementation and scalability. It should be noted that the NAT that takes place here on the gateway is a workaround specifically for deploying on top of Amazon, and not something inherent to the core solution, of which one of the original goals was to eliminate the NAT step from the previous semester's implementation.

III. Implementation[12]

A. Setup - Initial Testbed

1) Equipment Details: The initial testbed setup consisted of an Apple MacBook, running OSX 10.6.8 and three VMs through VirtualBox 4.3.6[7]. Each VM ran Ubuntu 13.10 (64-bit)[8]. The three VMs consisted of two "carrier" machines and one "controller". The carriers hosted instances of Docker version 0.8.1, and Open vSwitch 1.10.2, while the controller VM hosted the POX OpenFlow controller version 0.1.0, which used Python 2.7 as a runtime environment. Docker is software used for the creation and management of Linux containers.

2) Networking Configuration: The carrier VMs were connected to each other, the POX OpenFlow controller, and the host Macbook through VirtualBox networking. Both carrier VMs had three interfaces on different subnets: a "host-only"[9] network for data plane and control plane communication between the other carrier and the controller, another "host-only" network for SSH access from the host, and a third "NAT" network for initiating connections to the outside world. The "host-only" networks also had interfaces on the host Macbook. Routing was configured on the host to forward traffic to container subnets over these interfaces. The containers themselves lived on a separated subnet, and were connected to the data plane in different ways as described in the above "Architecture" section.

On the containers themselves, the Linux policy routing database[10] was altered to force packets "onto the wire" even when the destination IP address resided on the container itself. This was necessary because the containers were not part of our SDN control plane.

3) OpenFlow Controller: The POX[11] OpenFlow controller provided core software and libraries for conforming to the OpenFlow standard, and allowed for the custom user logic to be loaded as a single Python module conforming to the POX API. The module used in this project was created from scratch, but influenced by the example POX module "l2-learning", which comes with the POX installation.

The Open vSwitch instances on the carrier VMs were configured to use this POX controller. In OpenFlow operations, when a packet enters the switch, the switch looks for a rule in its flow table that might match packet information. If a match is found, the switch can take actions. Some actions include forwarding out a certain port, flooding, or rewriting the destination MAC address. If no match is found, the packet is sent to the controller. The controller can then asynchronously instruct the switch on what actions to take for future similar packets. This is called "installing a flow" on the switch. In our implementation, on a received packet at the controller, the controller installs a flow on the switch instructing it to, if the packet's destination IP and (layer 4) port match our static predefined mapping, re-write the MAC address and forward out a certain (layer 1) port.

Flows are also dynamically installed when a container initiates a connection, whether to the outside world or to another container, so that return traffic can reach the appropriate container, even if that container shares an IP address with another container.

Source code and installation instructions can be found at [12].

B. Setup - Amazon

The setup on Amazon was similar to that of the initial testbed except that the VMs were provided by Amazon EC2 as

opposed to VirtualBox, and the networking was supported by VXLAN encapsulation. A VXLAN tunnel between two OVS instances can be thought of as pseudo-wire, a virtual ethernet cable connecting two switches.

The "gateway" VM needed some special setup. It provided NAT functionality to convert Amazon's private IP addresses back to their original public IP addresses through the use of Linux iptables[13]. These public addresses were ultimately assigned to containers. Because from Amazon's point of view, this gateway VM hosted all the IP addresses that were actually used for containers, it needed to be a medium[14] instance. The other instances were micro. It was important that the gateway instance be configured with ip-forwarding enabled, and that the other instances had it disabled.

The controller VM doubled as a management jumpbox, and needed only one interface, associated with an Amazon public IP address. When we would configure any VM, we would first SSH into the controller VM, and then SSH onto the management interface of whatever VM we were interested in configuring. In this way we could make changes to the data plane interfaces on our carrier VMs without being disconnected and/or locked out. It also allowed us to not waste public IP addresses. In the entire setup, one public IP address was used for the controller/management VM, and the others we reserved for containers.

Beside the management interface on the carrier VMs mentioned above, the carrier VMs also had a data plane interface. These interfaces were assigned private Amazon addresses, were not associated with public addresses, and were used for VXLAN tunnel endpointing. The exception was the data plane interface on the gateway VM, which had one sub-interface for VXLAN tunnel endpointing (with no associated public IP address), and several other sub-interfaces configured with the pre-NAT Amazon private addresses (associated with Amazon public IP addresses assigned to containers). Because of the NAT at the gateway, these interfaces were never really "hit", besides to respond to ARP requests from Amazon.

C. Experimentation

We tested both implementations by assigning overlapping and non-overlapping IP addresses to containers on the same and on different carrier machines, and running netcat sessions between container and container, and between container and the outside world. All tests were successful. Variations included:

- host/outside world to container (including concurrent sessions to different containers with the same IP address)
- container to container, same VM, different IP address
- container to container, same VM, same IP address (different layer 4 port)
- container to container, different VM, different IP address
- container to container, different VM, same IP address
- container to host/outside world

IV. Conclusion and Future Work

One of my tasks this semester was to contribute to a paper to be submitted to the HotCloud workshop[16]. In additional to general editing, I wrote the sections about how one would deploy our system on top of a Amazon. Our paper was rejected. There were many comments, but the general theme was that the benefits we were claiming weren't worth the engineering effort. One drawback that I noticed is that the solution only works for applications that use TCP or UDP port numbers. For applications that depend only on IP address like ARP and ICMP, workarounds must be invented. We presented workarounds for ARP in this paper, but ICMP pings pose a challenge when multiple destinations can share an IP address. I believe work should continue. There is an opportunity to integrate the system with OpenStack or other Cloud Management systems, and to develop a more robust customer-facing frontend. But, first and foremost, more work needs to be done to uncover benefits for specific applications and use cases, and focus research around those use cases.

References

[1] Etan Zapinsky and Alex Merkulov, *Barge: Simple Container Network Management*, 2013.
[2] Open vSwitch. http://openvswitch.org/
[3] Docker. http://docker.io/
[4] Pyretic. http://http://frenetic-lang.org/pyretic/
[5] OpenFlow. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow
[6] An example of layer 2 security mechanisms from Cisco Systems. http://www.cisco.com/c/en/us/support/docs/switches/catalyst-3750-series-switches/72846-layer2-secftrs-catl3fixed.html
[7] Oracle Virtual Box. https://www.virtualbox.org/
[8] Ubuntu. http://www.ubuntu.com
[9] Virtual Box Networking. https://www.virtualbox.org/manual/ch06.html
[10] Linux Policy Routing. http://linux-ip.net/html/tools-ip-rule.html
[11] POX OpenFlow Controller. http://www.noxrepo.org/pox/about-pox/
[12] Code Repository and Setup Instructions. http://github.com/mzbenami/awic/
[13] Linux Netfilter and iptables. http://www.netfilter.org/
[14] Amazon EC Instance Types. http://aws.amazon.com/ec2/instance-types/
[15] VXLAN Standard. https://datatracker.ietf.org/doc/draft-mahalingam-dutt-dcops-vxlan/
[16] HotCloud Industry Workshop. https://www.usenix.org/conference/hotcloud14