Flexible Network Address Mapping for Container-based Clouds

Kyung-Hwa Kim, Jae Woo Lee, Michael Ben-Ami, Hyunwoo Nam, Jan Janak and Henning Schulzrinne
Columbia University, New York, NY, USA

***Abstract*—Container-based clouds have recently received great attention from the industry. However, we notice that this new type of cloud inevitably requires complex network setups and configurations from both providers and customers when deployed on an existing cloud system; Providers need to install additional network elements such as proxy servers and Network Address Translation (NAT), and customers need to use subdomain names and randomly assigned port numbers to access their services. Thus, we propose a new network architecture that performs M-to-N mapping between network addresses and containers in order to simplify the network setup and configuration. To achieve our goals, we adopt a software-defined networking (SDN) approach. We discuss the benefits and use cases of our approach, and present detailed designs and implementation.**

I. Introduction

Virtual machines (VMs) usually have been the only instances of computing resources that customers can purchase in the Infrastructure-as-a-service (IaaS) clouds. However, because VMs in the clouds are frequently used to perform only a single function (e.g., as Web server or database) [1], the question exists as to whether every cloud customer always needs to own VMs that contain a full operating system (OS) kernel and an entire set of system libraries.

To answer this question, and to provide alternatives to VMs, several technologies that enable lightweight cloud instances have been developed. First, in the *unikernel* [1] system, an application is compiled into a standalone bootable OS image that contains a small set of system libraries and kernel functionality that are used by the application. Second, container-based clouds have recently received great attention from the industry. A *container* is an instance of an OS-level virtualization environment that has its own isolated resources, but shares an OS kernel with other containers [2], [3]. Although the containers are lightweight, they provide isolated environments such as processes, system libraries, network namespaces, and root file systems [4]. Because of these benefits, the container-based IaaS cloud is rapidly growing in the market [5].

These approaches share the idea that thin instances are necessary in order to reflect the trend of running a single application on each cloud instance. Lightweight instances are beneficial to cloud providers because they can provision more number of instances on a host machine, and the migration time decreases owing to the reduced image size. It also enables providers to offer low-cost IaaS services to the customers who are not satisfied with Platform-as-a-service (PaaS) environments which do not allow OS-level access to the compute resources and provide limited functions to manage the life cycle of instances (e.g., launching and terminating).

978-1-4799-7899-1/15/$31.00 © 2015 IEEE

However, we notice that this new type of instance inevitably requires complex network configurations from both providers and customers when deployed on an existing cloud system. For example, a popular container-based cloud provider, *dotCloud*, runs multiple containers on VMs in the Amazon EC2, and sells the containers to their customers [6]. In *dotCloud* system, several network elements and technologies such as NAT, port mapping, and application-layer proxy servers are involved to correctly deliver incoming packets to the customers' applications that are running on the containers. This architecture enables containers to function properly as network servers; however, it contains several limitations that make configuration and management tasks confusing. For example, containers are addressed by particular subdomain names, rather than public IP addresses. In addition, applications on the containers are assigned random TCP/UDP ports instead of the regular port numbers used by the applications.

Our goal is to simplify the network setup and configuration by assigning public IP addresses and an appropriate port numbers to the applications. An application running on a container should be able to open any TCP/UDP port, regardless of other tenants residing on the same host. Also, the application could be routed directly through a public IP address, rather than through randomly assigned subdomains or private IP addresses.

Furthermore, we argue that decoupling of compute and network resources brings great convenience in terms of network management and configuration of applications. Instead of one-to-one mapping between IP addresses and containers, we propose to allow multiple containers to share the same IP address, and a container to have multiple addresses if needed. By doing this, we can avoid reconfiguring network addresses in the configuration files of applications when the instances are scaled up or down.

These goals cannot be achieved by using NATed networks and ordinary routing mechanisms. Therefore, we adopt a software-defined networking (SDN) approach using the OpenFlow [7] controller and virtual switches for packet routing. First, we explore the existing network architecture for a container-based cloud, then describe the detailed designs of our architecture and its benefits.

II. Background: Container-Based Clouds

A *container* refers to an instance in an OS-level virtualization environment. Using the isolated namespace functions of the Linux kernel, containers provide isolated virtual environments that are composed of their own processes, system libraries, network namespaces, and root file systems [2], [3]. Therefore, from the user's perspective, containers can be thought of as lightweight VMs because they provide isolated environments and root permission [4]. The main disadvantage of the container-based cloud is that users cannot choose guest operating systems and must rely on the OS kernel installed on the host.

A number of container implementations have been developed in different Unix-like operating systems including LXC, OpenVZ and Linux-VServer (Linux), FreeBSD jails (FreeBSD), and Solaris Containers (Solaris). According to the measurements of Birke et al. [8], on average, about ten VMs are running on a single physical server in today's clouds. In addition, we expect greater number of containers will be running on a single VM. In our experiment, we could run up to three hundred containers on a single VM running on Amazon EC2. Therefore, in theory, hundreds or thousands of containers can run simultaneously on a single physical machine depending on its hardware specification. This implies that providers can offer lower-cost instances (containers) to customers who seek lightweight instances in order to run thin applications without buying an entire VM. Furthermore, since the containers can be launched and terminated in only about few seconds, the provisioning time decreases dramatically when the customers scale up or down the number of instances. Further, it is beneficial for providers because it facilitates increasing server utilization and supports faster migration compared to the case of VMs. Container-based clouds are different from existing Platform-as-a-service (PaaS) clouds that do not allow OS-level access to the compute resources and provide limited functions to manage the life cycle of instances (e.g., launching and terminating).

Recently, LXC [2] and its wrapper Docker [9] have received significant market attention. OpenStack [10], an open-source cloud management system, supports LXC and Docker. Also, many companies are attempting to run their applications on *dotCloud* [6], a cloud service based on Docker instances.

III. Problems and Challenges

In this paper, we focus on the network architecture of container-based clouds. In order to run a network service (e.g., a Web server) that receives incoming packets from outside networks, a container must have its own network interface and address. However, this is not trivial because containers share the host VM's network interface and IP address. We briefly describe several existing methods to configure a network for this environment.

A. Nested NAT

First, one can use NAT inside a VM in order to host multiple containers. This approach has the advantage that it does not require the cloud provider's support because the owner of

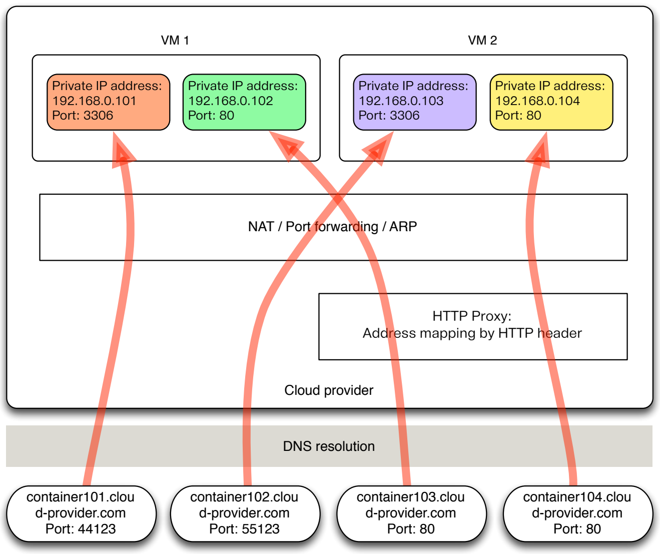
VM 1
VM 2
Private IP address: 192.168.0.101 Port: 3306
Private IP address: 192.168.0.102 Port: 80
Private IP address: 192.168.0.103 Port: 3306
Private IP address: 192.168.0.104 Port: 80
NAT / Port forwarding / ARP
HTTP Proxy: Address mapping by HTTP header
Cloud provider
DNS resolution
container101.cloud-provider.com Port: 44123
container102.cloud-provider.com Port: 55123
container103.cloud-provider.com Port: 80
container104.cloud-provider.com Port: 80

Fig. 1. Architecture of dotCloud

the VM can configure NAT inside the VM and associate it with the containers. Because common cloud systems already use NAT to map a public IP address to a private IP address of a VM, the address translation is carried out twice to address the containers (nested NAT). *dotCloud* runs their container-based cloud on Amazon EC2 using this method (Figure 1). However, we argue that NAT is not the best solution because containers behind the same NAT cannot listen on the same port because of TCP/UDP port-number conflicts. Although dotCloud uses port mapping for incoming and outgoing packets to support the containers, a randomly assigned port number is exposed to the external network instead of the application's original port number. Users who connect to a service via the Internet must know the assigned port number of the container. dotCloud uses a particular method to enable containers to use port 80 for Web servers. This is achieved by using HTTP proxies that distinguish incoming traffic by the subdomain name indicated in the HOST field in the HTTP header. However, this cannot be applied to other protocols that do not explicitly indicate the destination host name in their application headers.

B. Bridged network and public IP addresses

The second method is to configure a bridged network inside a VM. One can create a virtual bridge on the VM and connect the containers to that bridge. Then, the containers obtain their own IP addresses from the cloud provider's DHCP server. In this case, because the container receives an IP address within the same subnet of its host and other VMs, it will be considered simply as one of the VMs in the view of the network topology. This approach enables containers to run applications on any port without conflict because they have their own network addresses. However, this approach requires the provider to assign public IP addresses to every container as it does to VMs. This not only requires a large number of network addresses but also imposes additional overhead to the provider in translating the addresses because of the greater number of containers compared to VMs.

Although these two methods are plausible, we argue that neither of them is the best fit for containers because of their limitations discussed above. More importantly, we claim that it is not efficient that a container must have its own network namespace even though they run a single thin application which mostly uses a single port number.

For example, assume that a customer decides to purchase five containers instead of one VM for a Web service (We assume that five containers have the same amount of computing power as one VM, which is approximately calculated by the prices of dotCloud containers and Amazon EC2 `m.small` instances). In this case, the containers have the advantage of avoiding hot spots because they may be distributed over five different physical locations. However, the cost of management and configuration increases because the customer has five machines (containers) rather than a single machine (VM). When the customer scales up the number of containers to acquire more computing power, configurations of other applications that are linked with the Web service, such as load balancers, back-end servers, or databases must be updated to communicate with the new containers. In the NAT and bridged-network approaches, because of the one-to-one mapping between the IP addresses and the containers, the configurations of those applications have to be updated to include all the IP addresses of new containers. This management and configuration effort increases more sharply compared to the case of VMs, because a larger number of containers are needed to acquire the same amount of computing power.

IV. Flexible network address mapping

We assert that the limitations of these two methods and the management issue discussed above are caused by the tight coupling of computing units (containers) and its identifiers (IP addresses and port numbers). Thus, we propose a new network architecture that separates the network identifiers from the containers. The basic concept is that when a customer purchases public IP addresses, the provider assigns them to the customer rather than containers. Then, the IP addresses and port numbers can be mapped to any container owned by the customer in an M-to-N manner. In other words, multiple containers can share the same IP address, and even one container can have multiple IP addresses. For example, when a customer runs two containers, one for a Web server and the other for a database, these containers can be hosted on different virtual machines; however, they can share the same IP address. Because they have different port numbers, they can be distinguished by combining the IP address and port number. In other words, the identifier of the container is the combination of the IP address and port number. The gateway router forwards the incoming packets to different locations depending on this identifier. Therefore, the providers can place containers with the same public IP address on different VMs and place containers with different public IP addresses within the same VM. Customers can reuse their IP addresses on multiple containers (applications) without concern for their locations and port conflicts.

A. Benefits and use cases

The advantages of this approach are as follows.

1) Public IP address and port number: Each container can be accessed through its own public IP address rather than an arbitrary subdomain name. In addition, multiple containers can open the same port even when running on the same host. In Section IV-B, we describe an SDN approach that achieves this goal without additional elements such as NAT and port forwarding.

2) Simplifying static configurations: Static configuration files of applications that connect to other applications can be simplified since the code running on the containers would have the logical view of the network we provide. For instance, assume that a user plans to run a Web server and a database server on two different containers for resilience and fault tolerance. For simplicity in configurations, it would be much easier to think of the Web and database components as living on one logical network machine, as signified by having the same IP address but different ports.

3) Scaling and migration: When a customer scales up the number of instances of a stateless application, the newly created containers can simply have the same IP address that other containers use. In this case, SDN can be leveraged to provide round-robin load balancing or even more sophisticated load balancing algorithms based on customer demand without maintaining the list of every container's IP address. In addition, since the addresses of containers are independent of host VMs, the containers can be migrated to any locations without changing the network addresses and the port numbers.

4) Maximizing server utilization: Because of the separation of addresses and containers, providers are not required to place containers that have the same IP address within the same host. They can be located in any host in the cloud, thus increasing flexibility in cloud provisioning and maximizing utilization of the underlying physical machines.

B. Implementation

The proposed flexible address mapping system is difficult to implement with simple integration of NAT and port-mapping. Several complex and customized routing technologies might be required to support the M-to-N mapping scenario properly. However, our design goal is to build a transparent and controllable system without obscure network architectures.

Furthermore, because containers will be created, terminated, and migrated frequently owing to the business model (providing cheap and lightweight instances), the mapping tables are also expected to be updated very often. However, altering the NAT table and port mapping rules repeatedly can impose a significant overhead to the gateway router.

Therefore, in order to route packets correctly and support frequent changes of routing paths, we propose to adopt a software-defined networking (SDN) approach that uses a centralized controller connected to the switches. In this section, we elaborate on detailed designs of the proposed approach. First, we describe an architecture for cloud providers who own underlying infrastructure. Second, we propose two provider-agnostic architectures that can be built on existing public clouds operated by other companies.

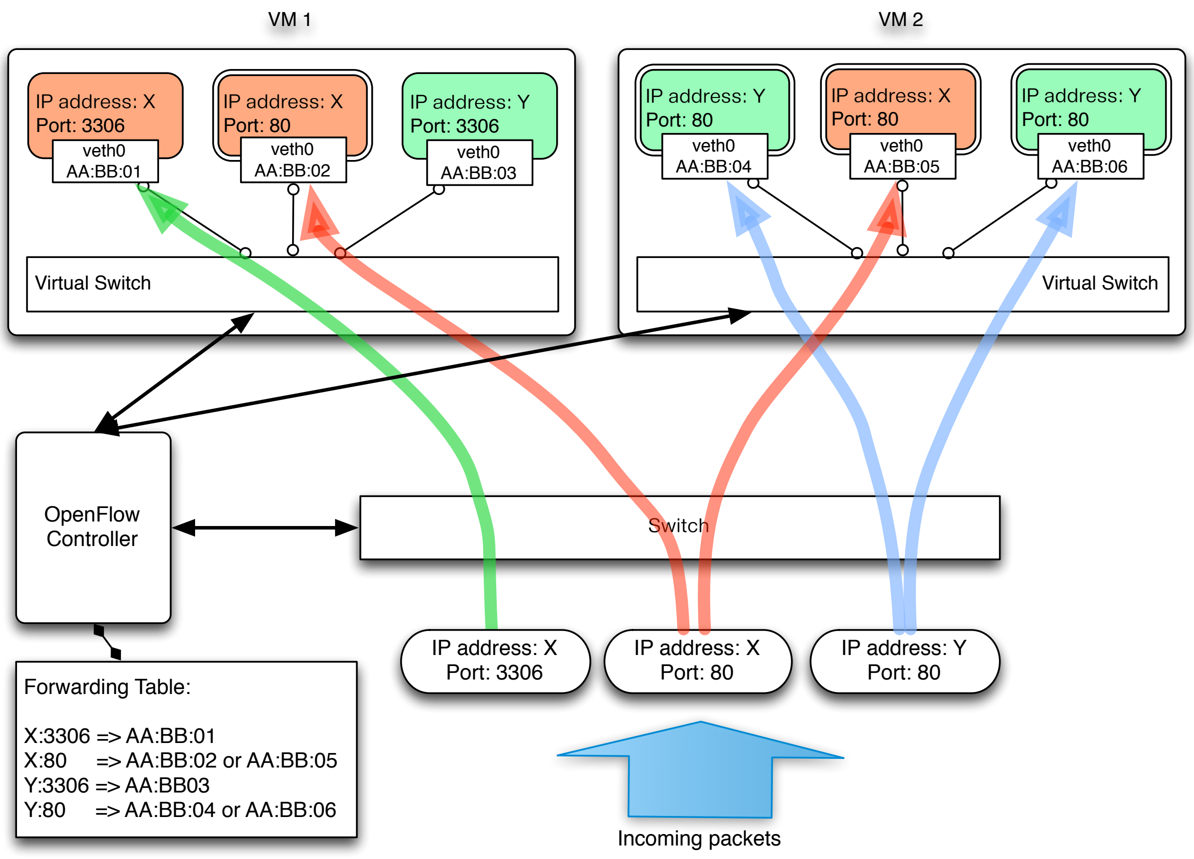

Fig. 2. Mapping IP addresses and TCP/UDP ports to containers in an M-to-N manner. A customer who purchased two IP addresses, X and Y, is running two applications (on port 80 and 3306). IP addresses and port numbers can be assigned to the containers regardless of their locations.

1) A provider-supported architecture: We place OpenFlow-supported switches between VMs and layer-2 devices (Figure 2). In addition, we install a virtual switch on each VM that connects containers to the network. When a new container is created, a software controller will be notified by our management agents and the switches in the data center will be updated to forward packets to the correct location. In our prototype, we manipulate ARP tables in layer-3 routers in order to fill the MAC address field of Ethernet packets destined to the containers with a pre-defined special MAC address (ARP spoofing can be also used on the switch connected to the router in case we do not have access of the layer-3 routers. The controller responds to ARP packets with the pre-defined MAC address if the queried IP address belongs to the containers).

When a new packet flow arrives, the switches forward the first packet to the controller to ask where to send the packets in this flow. The controller can distinguish the packets destined to the containers from other packets using the pre-defined MAC address. Then, the controller queries the database using the pair of destination IP address and port number of the packet to determine the MAC address of the corresponding container. Finally, the controller injects an OpenFlow rule to the switch. The rule instructs to replace the following packet's MAC address with the container's MAC address if the destination IP address and port number match. By doing this, we achieve M-to-N mapping between IP addresses and containers without a proxy server and a NAT process.

Centralized control of address mapping in a data center is not new. For example, several studies attempt to separate the network address and the locations of the VMs in a data center. PortLand [11] intercepts the ARP packets at the switches, and responds with a *pseudo* MAC address that represents the topological location of the queried IP address. VL2 [12] uses IP-in-IP encapsulation to decouple the actual IP addresses and logical IP addresses of the VMs to locate them at any place within a data center. However, our goal is different from these studies. We focus on the flexible assignment of public IP addresses and port numbers to containers.

2) Provider-independent architectures: If the VMs are provided by an existing cloud system (e.g., Amazon EC2), the network infrastructure between VMs will not be part of our SDN control plane. Because the proposed approach involves sharing IP addresses between containers, and Amazon uses standard non-SDN layer-2 and layer-3 forwarding mechanisms, a scenario could easily arise whereby Amazon will forward a packet to VM A based on the destination IP address only, while the real destination is on VM B. The SDN-controlled virtual switch in VM A may still perform the lookup to find the appropriate MAC address of the destination container, but will not be able to use layer-2 forwarding because VM B is in another place in Amazon's network.

In this scenario, we first propose to use a layer-2 tunneling between virtual switches running on the two VMs. We can use VXLAN [13], which encapsulates the entire layer-2 packet with layer-2 and layer-3 headers routable over the underlying network. In this way, we can have one VM act as VXLAN gateway, interfacing with the outside world, forwarding packets to the appropriate VM's virtual switch based on layer-2 MAC address. VXLAN's main use case is combating VLAN exhaustion in large multi-tenant data centers, but has an added benefit of providing layer-2 overlay tunnels at the hypervisor level.

An alternative approach is to have dedicated VMs with virtual switches acting as gateways between the outside world and our network. These virtual switches would not have any directly connected containers, but would share the container location information with the rest of the SDN control plane. All 'access layer' virtual switches could connect to one or more gateways in a star topology. The dedicated gateways also provide deterministic entry points into the network, advertising customer routes to the Internet. If our system is riding on top of one or more other cloud providers, it would be difficult to have the providers advertise our routes to the Internet. They may only advertise IP addresses that they provide, and may have rules about how their public addresses are used in the VMs. They may, for example, dictate that public IP addresses can only be assigned to VM-level interfaces and not to containers within the VM. In the 'dedicated gateway' scenario, traffic is immediately encapsulated in VXLAN and can be forwarded to the correct destination virtual switch, whether that destination is in the same cloud provider or in another.

V. Conclusion and future work

We investigated the network issues in the container-based clouds, and proposed a new network architecture that avoids complex network settings and reduces configuration tasks. Although we focused on the container-based clouds to describe our approach, we believe the proposed approach is applicable to any system that leverages machine-inside-machine models such as *Inception* cloud [14], which also adopts nested NAT to build a private IaaS cloud on top of existing public clouds.

We have implemented a prototype of the provider-supported architecture described in Section IV-B1 using OpenFlow and Open vSwitch. In our prototype, container management and network management are performed by separate custom built systems. It is useful to explore a unified cloud platform like OpenStack, supporting Docker through the Nova and Glance modules [15], and Open vSwitch through the Neutron module. We also suggested two provider-independent designs. To the best of our knowledge, this is the first study of network architecture for a container-based cloud. Thus, we plan to conduct an extensive evaluation in terms of performance and usability.

References

[1] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library Operating Systems for the Cloud," in *Proc. of ASPLOS*, Houston, Texas, USA, Mar. 2013.

[2] "LXC," http://linuxcontainers.org/, [Online; accessed Dec 2013].

[3] "Linux Containers," http://goo.gl/NzMbpU, [Online; accessed Dec 2013].

[4] "Linux Containers," http://wiki.gentoo.org/wiki/LXC, [Online; accessed Dec 2013].

[5] "Docker Challenges Virtualization Market with Containers," http://goo.gl/IsOiBt, [Online; accessed Feb. 2014].

[6] "dotCloud," https://www.docker.io/, [Online; accessed Jan 2014].

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

[8] R. Birke, A. Podzimek, L. Y. Chen, and E. Smirni, "State-of-the-practice in data center virtualization: Toward a better understanding of VM usage," in *Proc. of IEEE DSN*, Budapest, Hungary, June 2013.

[9] "Docker," https://www.dotcloud.com/, [Online; accessed Jan 2014].

[10] "OpenStack," http://www.openstack.org/, [Online; accessed Jan 2014].

[11] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric," in *Proc. of SIGCOMM*, Barcelona, Spain, Aug. 2009.

[12] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *Proc. of SIGCOMM*, Barcelona, Spain, Aug. 2009.

[13] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," Working Draft, Internet-Draft draft-mahalingam-dutt-dcops-vxlan-07, Jan 2014.

[14] C. Liu and Y. Mao, "Inception: Towards a Nested Cloud Architecture," in *Proc. of HotCloud*, San Jose, CA, June 2013.

[15] "OpenStack and Docker," https://wiki.openstack.org/wiki/Docker, [Online; accessed Feb. 2014].