Design and Challenges of a Lightweight Container-Based Architecture for Multi-Clouds

Syed Ahmed and Muthucumaru Maheswaran
School of Computer Science, McGill University, Montreal, Canada
Email: syed.ahmed4@mail.mcgill.ca, maheswar@cs.mcgill.ca

***Abstract*—Cloud computing is a major part of the modern computing landscape. Although cloud computing systems already offer many advantages, there is a need to interconnect them such that resources from one provider can be substituted for resources from another provider. For existing cloud providers such a *marketplace* offers many advantages and disadvantages. In this paper, we outline an architecture for creating an multi-cloud without the full cooperation of the participating cloud providers. Our multi-cloud architecture leverages several new technologies that have started gaining a foothold in the industry such as Linux containers and software defined networking. We present a detailed description of the architecture and explain the challenges that need to be addressed in realizing it. In particular, we want an multi-cloud that is lightweight as possible and does not create a significant compute, memory, or storage footprint by itself.**

I. Introduction

Cloud computing is offered by large number of providers at this point in time: Amazon, Google, and Microsoft are few notable ones. Each cloud provider has its own conventions in terms of dimensioning resources like compute, storage and network. They also differ in terms of their pricing and policies for the resource units they offer to their customers. This creates a situation for the customer, where migrating a cloud deployment from one provider to another becomes a costly endeavor – this is referred to as the *vendor lockin*. To have a reliable and cheap cloud deployment the cloud customer needs to avoid vendor lockin so that the customer can migrate the deployment to benefit from the better offering.

Apart from the issue of vendor lockin, there is quite a bit of differences among the cloud offerings from the different vendors. For example, Amazon AWS offers GPU instances which are not offered by other major providers. Azure and AWS offer Windows OS images whereas Google Compute Engine (GCE) is still in beta (no SLA guarantees offered at this time). This leads to user preferring one cloud over the other and further complicates comparison of cloud instances across the different providers.

Further, the cloud providers differ in terms of how they price their instances. For example AWS offers three pricing schemes: on-demand – where the user pays an hourly fixed price, reserved – where a bulk price is paid to reserve an instance for a longer duration (an year or more) and spot instances – where the user can bid on instances using an auction mechanism. GCE on the other hand offers a minute-granular pricing. Depending on the type of workload, budget and complexity involved, a particular cloud provider may be preferred by the customer over others in terms of pricing.

Geographies also play an important role in limiting the choices customers have in terms of cloud providers. Not all providers are everywhere and depending on cloud customers' locations and from where their applications get most of the requests from, they may prefer certain providers with installations in given geographic localities. Amazon has the largest installation for a public cloud distributed over 9 regions across the Internet whereas Google has its GCE in only 3 regions. Apart from this, users might have sensitive data that they may not wish to share with the provider due to lack of trust or because some laws may prevent their data being stored on an overseas server.

Different models have been proposed to overcome the above mentioned issues. All of them try to combine various providers into a consistent single cloud layer which is agnostic of a provider. Depending on the level of engagement and agreement between the different providers, [1] broadly classifies the current initiatives into three different models.

- *Inter-Cloud:* This is an ideal collaboration between different cloud providers. In this type of a model, there are standard APIs for communication which are agreed upon or implemented by all providers. This could be considered as an internet of clouds.
- *Federated Cloud:* In this model, a group of providers mutually agree to collaborate and share resources among themselves. This is done using agents which translate requests from one cloud to another. To the end user, they expose a common API with vendor specific additions.
- *Multi-Cloud:* This model overlays a virtual cloud on top of disjoint clouds. In Multi-cloud, there is no collaboration between individual cloud providers. All the orchestration is done by a third party agent or a proxy. The agent exposes a standard interface to the user and translates it to provider specific calls transparently. It also creates an overlay network in which hosts running on different providers are seamlessly connected.

It is in the best interest of a cloud provider to maintain competitive advantage. In doing so they may not wish to share their unique offerings with others. Providers may also make migration inherently complex to retain customers. This makes the vision of global inter-cloud difficult to achieve. In fact, even getting a federated cloud of top providers can be difficult to realize. This leads us to focus on multi-clouds as they do not require collaboration with individual providers, can be implemented as a layer on top of existing infrastructure and can combine public and private clouds seamlessly.

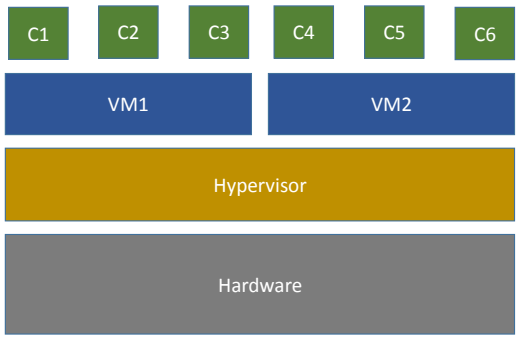

Fig. 1: Containers placement in a typical cloud

In this paper, we present a multi-cloud architecture using containers as the isolation principle. We implement an overlay network on top of existing cloud networks using VXLAN as the tunneling protocol to provide a virtual private cloud (VPC) for each cloud tenant. Finally, we explore optimizations that can be performed as third party service providers to optimize the workload performance of tenants and reduce storage and migration costs.

The proposed architecture addresses many of the concerns raised above. Firstly, it is not tied to a particular provider which prevents vendor-lockin. There is a consistent API which transparently allocates resources on the right cloud based on the requirements. And since this implements a seamless overlay network, cloud providers from any location can be leveraged.

The intelligent workload placement algorithm makes sure that each container on the host is not being affected by the operations of other containers on the same host. Finally, a minimal shared filesystem allows for lower footprint when migrating containers across different hosts.

II. Containers As Virtualization Platforms

Containers are a method of operating system-level virtualization which are lighter than full virtual machines (VMs). Unlike VMs where each instance runs its own kernel and provides complete isolation, containers share the same host kernel and provide isolation of resources like processes, filesystems, IPC, network, CPU, Memory and I/O based on kernel primitives (namespaces & cgropus in Linux) [2]. This enables them to have very low overhead on various operations when compared to traditional VMs [3].

Different operating systems offer their own version of containers, although not all of them implement the same isolation primitives. The popular ones include Freebsd jails, Solaris containers and Linux containers [4], [5], [6]. In this paper we focus on the containers in Linux. Various implementations of linux containers exist most popular are OpenVZ, Vserver and LXC. In our study we focus on LXC as it is the most popular implementation and does not require any patches to the kernel.

Due to many layers of abstraction, traditional VMs have lower performance than baremetal OS. Depending on the type of workload and resources utilized, this performance degradation can become significant [3]. Containers on the other hand share the same kernel and hence avoid the abstraction overhead. This leads to near-baremetal performance on many operations [7]. As there is no kernel to boot from, containers have faster boot times and use significantly lower CPU/memory when they are idle when compared to VMs (Because even in idle state, VMs run the kernel which occupies memory/CPU) . Containers have a higher density on a given hardware meaning, given the same hardware, we can run more containers than VMs [8].

Despite their advantages, containers still have some inherent issues – host and guest OS decoupling is one of the important problems. With full virtualization, the guest could be very different from the host such as non-native kernels (Windows on Linux) or multiple kernels with different versions. This is not possible using containers as each container is tied to the host kernel. Despite providing isolation, studies have shown that workloads on one container can affect the performance of another container despite them being completely interrelated [7]. This is not seen in traditional VMs as the underlying hypervisor offers full isolation. Since containers share the same kernel, a security exploit in one container can lead to the attacker being able to access every other container. For example, *CVE 2013-1858* exposed a bug in *clone()* using the user namespace in Linux which led to full root privilages on the base host [8].

Some container implementations like Docker enforce a strict workflow model [9]. This makes it difficult to port traditional applications to a container based environment. Also, containers even though they isolate resources, they do not actively communicate this to the application. For example in Linux based containers, if the container is allocated only one CPU, it can still see all CPUs and applications which rely on this metric, like deciding number of threads to spawn or how much memory to allocate, can have undefined behavior.

III. Container-Based Multi-Clouds

A. Overview

We aim to build a container based multi-cloud by implementing an overlay on top of existing public clouds. We use LXC as the container interface running on top of VMs provisioned in different clouds. To enable communication between containers in different clouds, we implement an overlay network based on VXLAN tunneling using Open vSwitch (OVS). We use a central SDN controller based on Opendaylight which configures and installs flows on the OVS [10]. The entire system is managed by a central orchestrator. The orchestrator exposes a public API and is the endpoint to which users connect and request for containers.

Current multicloud offerings focus on a broker based architecture where users are given an interface to manage multiple clouds from a single console. This offers a mere convenience to the user and does not abstract the underlying complexity from the customer [11]. Our primary goal for building this system is to ensure a consistent interface across multiple cloud with minimal or no cooperation between the underlying cloud providers. This would prevent vendor-lockin and allows customers to optimize their deployments across multiple providers. The proposed architecture has a relatively

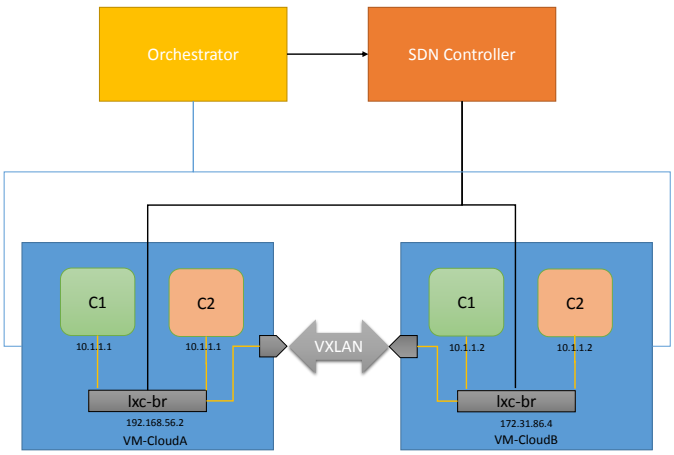

Fig. 2: Multicloud Architecture: Two VMs on different clouds A & B host containers which belong to different isolated guest networks sharing the same IP address. The OVS bridge $lxc - br$ maintains isolation and routes between different hosts, maintaining an overlay. An SDN controller installs flows on the OVS and the host itself is managed by a central orchestrator.

small footprint which leads to saving in terms of hardware and VMs allocated on remote clouds. The small footprint also ensures ease of migration if the user wants to migrate to a different provider. Finally, SDN enables seamless networking across multiple clouds and reconfigurations happen transparently of container migrations.

B. Container-Based Multi-Clouds Architecture

Figure 2 shows the basic architecture of the container-based multi-cloud. Each cloud provider hosts the base VMs which are responsible for running the containers. By default, these hosts are assigned a public IP which is managed by the cloud provider. The base VMs are allocated based on the capacity requirements which are decided by the orchestrator. If there is a need to spin up a VM to host more containers, the orchestrator decides which is the best cloud to allocate the base VM on. Once a base VM is allocated, the orchestrator installs an OVS bridge on the host and reassigns the public IP to the bridge. The external interface is reconfigured as a port on the OVS bridge. Finally, the bridge is configured to connect to the SDN controller. The OVS is responsible for encapsulating and routing traffic from one host to another.

We use LXC as our container implementation as it allows us to achieve near VM experience in that it gives us complete shell access making the container behave like a self-contained operating system instead of an application isolation mechanism (e.g., Docker, systemd-nspawn, lmctfy) [9], [12], [13].

When a container comes up, a virtual ethernet pair is created. One end of the pair is attached to the container and the other end is attached to the OVS bridge. Both interfaces act like a pipe. Packets sent to one interface comes out of the other. OVS also associates this interface with an Openflow port which is used to configure flows. The OVS acts as the default gateway for every container. A packet destined to another container is forwarded to the local OVS via the virtual ethernet pair. OVS looks up its flow table to decide the destination container. If the destination container resides on another host, OVS encapsulates and tunnels the packet using VXLAN to the destination host using the public interface. Once the packet reaches the destination OVS, it is decapsulated and the VXLAN header is removed to get the original packet. The OVS looks at the flow table to decide which destination container the packet is to be sent and it forwards it to the correct virtual interface.

An orchestrator is responsible for managing containers on all the hosts. The orchestrator keeps track of the current state of each container and keeps the metadata of each of the container's properties like the host; IP address; image; CPU and memory restrictions. Any operation on the container (start, stop, destroy) has to go through the orchestrator. In addition to this, the orchestrator is also responsible for collecting monitoring information from the host and the containers which will help in deciding optimal placement of workloads and if new base hosts need to be spawned. The orchestrator is also responsible for maintaining a global view of the cloud. This enables it to control the networking of all hosts via SDN. In our architecture, we use Opendaylight as the SDN controller with Openflow to communicate between the controller and the OVS. In essence, we are creating an overlay network on top of the existing cloud-network and control it using SDN via a central orchestrator.

C. Secondary Storage Mechanisms

Containers implementations such Docker and LXC support the use of an overlay filesystem in which there is a base image which is read-only and any changes that are done go to a different place transparent to the user. However, the base image which is read-only is replicated across containers. We intend to develop a base layer which can be shared among containers running on the same host. This image will be the same for any host in the multi-cloud system. This would make the migration footprint very small as only the non-unique files need to be replicated.

D. Workload Adaptation

Studies show that unlike complete VMs, containers have an inherent problem with resource isolation. Though the containers are good at isolating CPU resources, [7] shows that there is a 50% degradation in performance of a container on memory heavy workloads when other containers on the same host are also performing memory heavy workloads. This is partly due to the fact that they share the same underlying kernel and operations of all containers are ultimately handled by the same kernel. This could lead to situations where one container can overwhelm the kernel indirectly causing other containers to have degraded performance. We propose an intelligent workload migration algorithm which continuously monitors the VMs and decides where the next workload should run or if containers-to-hosts assignments need to be rebalanced.

Each container and the hosts on which these containers are instantiated, periodically report their CPU, memory and I/O usage to the central orchestrator. The orchestrator balances the workload based on usage patterns. For example, it will try to allocate two containers executing memory intensive jobs

on separate hosts. It may also dynamically decide to migrate containers from one host to another to balance the main hosts usage and reduce I/O between different containers.

Current live migration techniques of VMs rely on underlying hardware support to make the migration possible. Moreover, migration can only be done between same hypervisors. i.e., one cannot live migrate from Xen to VMWare or vice-versa. In a multi-cloud scenario, both of the above conditions are not possible as the hardware is different on different clouds and users don't have access to the cloud hardware. Implementing migration on a container level is possible as the underlying hardware is abstracted away by the host VM.

To migrate a container, three entities need to be transferred to the remote host. Memory, filesystem and process state. The first state of migration is the bulk transfer phase where the memory pages belonging to that container are transfered to the remote host. After the memory transfer is complete, the filesystem is copied. Finally, the process state for each process (registers, open files, sockets) is transferred to the remote host. Note that during the copy, there may be active changes happening on both memory and filesystem that will make the transferred pages outdated. The process can also open and close files and socket which will make its state inconsistent with the remote state hence preventing migration. To overcome this, we will keep a diff of all the changes that are happening in the container (memory, filesystem and state) and keep updating the remote host. Eventually, the container will have a pause time. i.e time where there is no activity greater than the amount of time taken to transfer the difference. This means that during that time, both the remote and local containers are in sync. Once this event triggers, the local container is paused and the remote container is started completing the migration.

Finally, the SDN controller is informed about the change and new flows are installed on the respective OVS bridges. The network is reconfigured so that that container gets the same private IP address in the remote location. If the container had a public IP address, a redirector is installed at the initial location and old connections are forwarded to the new location whereas the new connections are made to the new IP.

Once the migration is completed and the new container is running, the old container is destroyed by the orchestrator and metadata for that container is updated with new values.

IV. Implementation Challenges

Currently checkpointing & migration of containers is not supported by the Linux kernel. Memory and process migration require changes at the kernel level which can be non-trivial. The architecture relies on tunneling between different clouds. This can become a bottleneck as we don't have control over the underlying network. Depending on the number of flows and traffic, the OVS itself can become a bottleneck as it is running on top of a VM instead of a dedicated host and it is responsible for routing the traffic of all the containers in the host. OVS could also become a single point of failure for a host as a crash would stop connectivity of all the containers in that particular host. This could however be prevented by using a redundant OVS with a trade-off of increasing the complexity of the system.

V. Related Work

[1] discusses various cloud architectures extensively with multi-cloud being one of them. mOSAIC built an opensource API and platform for multiple clouds [14]. They introduce the concept of a component as the resource abstraction for a cloud. Applications use components which are later translated to actual cloud components like VMs using different plugins for different clouds. Although their platform has APIs for multiple clouds, Networking and management of VMs is left up to the user.

Apache JClouds is a multi-cloud toolkit written in Java [15]. It exposes a standard API for managing compute services (VMs) and storage services. The management of these resources is left to the user. This also does not offer a network overlay which is essential for seamless communication between different clouds.

Apache Libcloud is similar to JCloud except that it is written in Python [16]. Although it supports fewer clouds than JCloud, the core concepts remain the same. i.e providing a consistent API to end users for each cloud.

All the above approaches still work on the resources provided by the respective clouds i.e., VMs. They do not however tackle networking between different clouds.

[17] proposes a publish-subscribe mechanism for routing messages between different clouds. Each client (VM) can be both a publisher and subscriber. Brokers are the entitles that perform matching and forwarding of messages. Each client sends its publications to its respective broker which is in-turn connected to other Brokers creating a virtual network on top of a physical network.

[18] uses user level virtual routers to create an overlay network. One VR is deployed per broadcast domain and/or site and a module on each host intercepts packets and tunnels it to the VR. Once the packet reacher VR, it routes the packet to the destination VR where the packet is decapsulated and sent to the correct host. This system relies on having additional nodes just as routers and is not scalable as the VR becomes a single point of failure for the entire site. Also, all the VRs are assumed to be known to other VRs.

[19] proposes a testbed for prototyping Information-Centric Networking (ICN) testbeds using containers. However, in this deployment, the primary goal is to prototype and test various ICN protocols instead of developing a generic multi-cloud where end users can run their applications.

To the best of our knowledge, Rancher is the only multi-cloud container based platform [20]. The system built keeping Docker as the base container implementation mechanism. It creates an overlay using IPSec tunnels for interconnection between different clouds. However, Docker does not support generic applications as docker philosophy promotes running only one application in a container this means that applications need to be modified in order for them to run on Rancher. Moreover, Rancher does not support container migration and workload adoption which makes it prone to the inherent problems that containers have.

VI. Concluding Remarks and Future Work

We propose an overlay-based architecture for multiclouds using containers and SDN. We highlight current challenges in developing an inter-cloud system and our proposed architecture address some of those points. We also discuss intelligent workload placement and migration techniques. There is a lot of work to be done in ensuring effective isolation among containers. Kernel support for migration and checkpointing needs to be developed for the workload allocation algorithm to manage containers more efficiently.

References

[1] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience*, vol. 44, no. 3, pp. 369–390, 2014.

[2] R. Rosen, "Resource management: Linux kernel namespaces and cgroups," [Online; accessed 2-Feb-2014].

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 28, p. 32, 2014.

[4] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, vol. 43, 2000, p. 116.

[5] D. Price and A. Tucker, "Solaris zones: Operating system support for consolidating commercial workloads." in *LISA*, vol. 4, 2004, pp. 241–254.

[6] M. Helsley, "Lxc: Linux container tools," *IBM devloperWorks Technical Library*, 2009.

[7] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, 2013, pp. 233–240.

[8] R. Rosen, "Linux containers and the future cloud."

[9] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[10] A. OpenDaylight, "Linux foundation collaborative project."

[11] Rightscale, "Rightscale hybrid cloud," [Online; accessed 2-Feb-2014].

[12] GNU, "systemd-nspawn lightweight namespace container," http://www.freedesktop.org/software/systemd/man/systemd-nspawn.html, [Online; accessed 2-Feb-2014].

[13] Google, "LMCTFY Google container stack," https://github.com/google/lmctfy, [Online; accessed 2-Feb-2014].

[14] D. Petcu, B. Di Martino, S. Venticinque, M. Rak, T. Máhr, G. E. Lopez, F. Brito, R. Cossu, M. Stopar, S. Šperka *et al.*, "Experiences in building a mosaic of clouds," *Journal of Cloud Computing*, vol. 2, no. 1, pp. 1–22, 2013.

[15] A. S. Foundation, "Apache jcluods: The java multi-cloud toolkit," [Online; accessed 2-Feb-2014].

[16] ——, "Apache libcloud: Python library for interacting with cloud service providors," [Online; accessed 2-Feb-2014].

[17] M. Ficco, L. Tasquier, and B. Di Martino, "Interconnection of federated clouds," in *Intelligent Distributed Computing VII*. Springer, 2014, pp. 243–248.

[18] M. Tsugawa, A. Matsunaga, and J. Fortes, "User-level virtual network support for sky computing," in *e-Science, 2009. e-Science'09. Fifth IEEE International Conference on*. IEEE, 2009, pp. 72–79.

[19] H. Asaeda, R. Li, and N. Choi, "Container-based unified testbed for information-centric networking," *Network, IEEE*, vol. 28, no. 6, pp. 60–66, 2014.

[20] Rancher, "Rancher: Portable aws-style infrastructure service for docker," [Online; accessed 2-Feb-2014].