An Empirical Study of OpenStack++ in edge computing

Xin(Adam) Chen

University of Oregon
achen@uoregon.edu

Abstract

The emergence of cloud computing leads the variety of computer branches and technologies. As a branch of cloud computing, edge computing is a distributed computing diagram which allows edge devices as edge nodes to provide computation. To achieve this, the OpenStack++, as known as cloudlet, is introduced to represent a new architectural element in the 3-tier hierarchy: device - cloudlet - cloud. This empirical study will briefly introduce the OpenStack++, including the code structure, implementation, and features. Then, the thesis will examine the performance of OpenStack++ and compare it with Amazon EC2 instances and Microsoft Azure Virtual Machines.

I. Introduction

OpenStack++ is developed by Elijah project at Carnegie Mellon University from 2012 to 2017. It is built based on 4 different Github repositories: *Elijah-OpenStack, Elijah-provisioning, Elijah-QEMU, Elijah-discovery-basic*. It was successfully implemented and it leads the development of more practical applications. This thesis mainly focus on Elijah-OpenStack, which integrates cloudlet functionality and provides customized graphical user interface in the OpenStack [1].

i. OpenStack++ features

The important attribute of OpenStack++ is that it only has the **soft state**. It does not have any hard state. The avoidance of hard state means that each cloudlet adds close to zero management burden after installation: it is entirely self-managing [2]. To support the soft state in OpenStack++, it has the following features.

Import Base VM. This feature allows developers to import VM images to OpenStack++. The VM doesn't have to be running a Linux distribution. It even could be a Windows 7/10 VM that users use as virtual desktops. Certainly, users can import a customized VM which includes all the libraries and packages. However, due to OpenStack++ uses modified QEMU fo visulization, before importing, the VM image should be obtained by using the export-base command of clouldet via terminal.

Resume VM. After importing the base VM, this feature allows initializing a VM instance from the base VM. It will restore the state from the memory snapshot and disk snapshot provided by the base VM. It is necessary to soft-reboot the new VM instance because the network interface the base VM has may be different from the one which is used by current machine and then a soft-reboot will restart Nova service to resolve the network interface configuration.

Create VM Overlay. This feature will analyze and compare the current VM and its base VM, and then extract the compressed differences of the disk and memory snapshots with the base VM [3]. The operation will apply optimizations to generate a minial VM overaly and finally save the VM overlay in Glance storage [4]. The VM Overaly also contains some metadata associated with the base VM into JSON format and pass it as a URL. By performing 'Create VM Overlay', developers can get the VM overlay which could be used for VM Synthesis later.

Perform VM Synthesis. VM Synthesis al-

lows developers can obtain the customized VM by synthesizing the VM overlay and the base VM during the run-time. A back-end server of a nearby OpenStack++ will be launched based on users' requests so that it can receive the VM overlay information via an HTTP POST message. By extracting the metadata about the base VM from the JSON, the OpenStack++ will initialize the base VM first, and then the VM Overlay will be applied into the instance too.

VM handoff. VM handoff will be triggered When users have physical movements between different OpenStack++ clusters. To provide smooth and same experience among different OpenStack++ clusters, the current OpenStack++ will execute the VM handoff which will migrate a running VM instance to another OpenStack++ as needed. First, it will authenticate the credential information of other OpenStack++ clusters to ensure that the another OpenStack++ is trustable. Then it will generate the VM Overlay and send it to another OpenStack++ by HTTP POST request. Similarly, like the process of performing VM Synthesis, once the new OpenStack++ receive and extract the configuration from the URL, it will automatically perform VM Synthesis to build the same VM instance users had before.

ii. OpenStack++ code structure and implementation

The another important attribute of OpenStack++ is that it was built based on standard cloud technology: It encapsulates offload code from mobile devices in virtual machines (VMs), and thus resembles classic cloud infrastructure such as Amazon EC2 and OpenStack. In addition, each cloudlet has functionality that is specific to its cloudlet role [2]. Technically, the OpenStack++ heavily re-use some OpenStack APIs so that the OpenStack++ will always be compatible with the OpenStack. In the *elijah-openstack* repository, the whole code structure can be divided into three parts: installation scripts, OpenStack++ GUI codes, and cloudlet APIs implementation. It should be pointed it out that cloudlet APIs belongs to high-level design since more infrastructural implementations are done in *Elijah-provisioning and Elijah-QEMU*, not in *Elijah-OpenStack++*.

Installation Scripts. The OpenStack++ utilizes Ansible Playbook to install and configure OpenStack Kilo release and OpenStack++. Before running these scripts, developers have to configure the public network interface and the flat network interface to configure Nova service of OpenStack work properly during the setup. First, the Ansible script will check and install any required dependencies, such fabric, openSSH and Git. Then, it will modify the kernel modules and reload them to meet some necessary installation requirements. After that, it will begin to install OpenStack Kilo release and add OpenStack++ module into it. The next thing is replacing the nova config files so that it can accept the base VM generated by OpenStack++. After installation, developers are able to use the functionality either by terminal or dashboard of OpenStack. However, the virtual interface is not persistent and will be lost after reboot [3], so it is necessary to re-run the whole Ansible script to re-create the interface and restart nova service.

OpenStack++ GUI. OpenStack++ implements the GUI by encapsulating Horizon and Glance APIs of OpenStack and rendering Django template HTML pages to customize a new tab under Project section. It defines several GUI modules, such as forms, panels, views. These modules are used in Model-View-Controlle pattern. When GUI widgets are triggered, the corresponding APIs will be invoked.

Cloudlet APIs implementation. As I introduced at first, OpenStack++ encapsulates major part of cloudlet infrastructure which is developed in *Elijah-provisioning, Elijah-QEMU, Elijah-discovery-basic* into some high-level APIs. Since OpenStack++ is the integration of cloudlet library functionality , it will be more clear to describe the APIs by interpreting the corresponding feature.

Import Base VM. the OpenStack++ API for cloudlet base creation actually invokes some existing Glance storage APIs from OpenStack, because essentially the process of importing a

base VM can be considered as saving its disk image, memory snapshot and their hash value lists in Glance. However, to distinguish the Cloudlet's base VM, some unique markers are necessary. When 'export-base' command of cloudlet is executed via terminal, the image will be tagged by a special keyword: cloudlet-type. Meanwhile, for further implementation, some other properties are kept in the metadata as well, such as UUIDs of all other associated files, VM's libvirt configuration and so on.

Resume VM. Resuming VM is similar to instantiating a new VM instance using a VM snapshot [4]. The command will be passed to the nova-compute node for launching VM. The OpenStack++ builds a customized driver: cloudletDriver which inherited from *Libvirt-Driver* from OpenStack. The driver will check whether the VM imported has the tag 'cloudlet-type' or not. If it has, then it will get the VM's libvirt configuration from its metadata and use them to do the VM visualization. The important difference between traditional OpenStack and OpenStack++ is OpenStack will initialize a VM instance from booting but OpenStack++ will resume the VM from its original status such that OpenStack++ is able to switch instances among different OpenStack++ without losing any configuration and data.

Create VM Overlay.: Generating a VM Overlay is a process that the compressed differences of the disk and memory snapshots with the base VM, so actually it is also a process of resizing and rebooting itself. Hence, when 'Create VM Overlay' command is conveyed, it will be passed to the visualization driver(cloudletDriver). Then a customized API which inherits the nova RPC is invoked to generate VM Overlay. Finally, the VM Overlay will be store in the Glance. Until now, we are able to have a perspective about how cloudlet service providers interact with developers and how developers deploy their applications within cloudlet nodes. Specifically, cloudlet service providers could build and deploy cloudlet nodes around customers. To keep it light and usable, the cloudlet node should have a base VM which contains minimal common libraries for application developments. Then, developers could just deliver their VM overlays to cloudlet, which saves more time and resources.

Perform VM Synthesis. While performing VM synthesis, the API will access the VM Overlay URL and check the metadata of VM overlay. First, it reads the information about the associated base VM first so that OpenStack++ can select the instance flavor and start initializing a VM instance automatically. Then it is similar as 'Resume VM', the OpenStack++ will invoke APIs to let CloudletDriver perform VM visualization. However, the VM spawning methods are overridden so that OpenStack++ checks overlay URL first and then perform VM synthesis to ensure the request is differentiated from the normal request of VM creation.

VM handoff. To switch to another OpenStack++, it is no doubt that the original OpenStack++ has to have permissions to let another OpenStack++ perform VM Synthesis. The auto-token will be sent via a client program so that the destination keeps the credential information. After authentication for permissions, the original OpenStack++ cluster will perform 'Generate VM Overlay' and send the JSON payload which contains some metadata via an HTTP POST request to its destination. When the destination OpenStack++ successfully receive them, it will perform 'VM synthesis' to obtain the customized VM instance for users.

II. Performance comparison under the real application

The thesis designs a simple full-stack video streaming project to examine the throughput of the application and compare them by deploying the back-end server at different locations. The front-end is using the optimized product build of React.JS, and the back-end is using standard Express framework based on Node.JS. The back-end server will transport the video stream chunk by chunk to the front-end server. The network performance is captured and analyzed by Chrome DevTools Network panel. All tests are performed under a stable

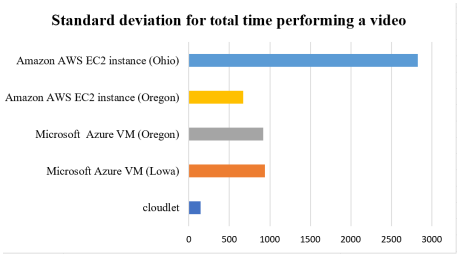

Figure 1: *Standard deviation comparison*

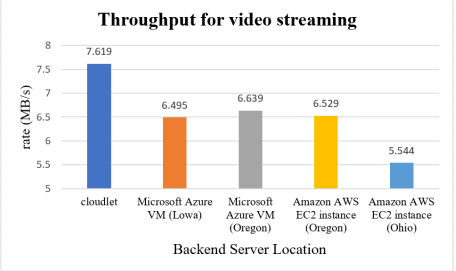

Figure 2: *Throughput comparison*

and 100 Mbps network connection. For a video streaming project, a smooth and low-latency experience is most important for users. So, by examining the time the whole time spent to deliver the video and instantaneous network speed, we can have an insight into the throughput and stability. The figures above illustrate the performance comparison among different locations.

These diagrams clearly state that cloudlet has better throughput and more stable during the tests. Also, if we only compare the performance among Amazon EC2 instances or Microsoft Azure Virtual Machines, it is obvious that the distance between cloud and users has a significant influence on the performance of applications. The data may lose or takes longer time to travel, as the distance increases. Since the cloud and the cloudlet as being identical except for the proximity to the user, when applications run on the cloudlet, it has reduced latency and more stable because the computation is placed at the edge which is closer to the mobile device. More importantly, it really inspires us that cloudlet could possibly resolve some last-mile problems in cloud computing. After deploying enough cloudlet nodes around end-users, each individual will be empowered and achieve more personal computing.

III. Implementation flaws in OpenStack++

Although OpenStack++ provides such an amazing performance and economic benefits compared to some traditional and popular Cloud service in the current market, it is still noticeable that OpenStack++ has some implementation flaws which should be resolved before extensively used by companies which care more about stability and maintainability. During the development and utilization of OpenStack++, I observed these following problems.

Non-persistent Network Interface. The current implementation of OpenStack++ will lost its virtual network interface after a reboot. To remedy this situation, developers have to rerun the whole Ansible scripts to recreate the dummy network interface and restart some necessary Nova service [3]. First of all, We can image that, due to the power outage, periodic maintenance or some inappropriate operations which may result in shutdown or reboot in OpenStack++ clusters, it may cause fatal problems that all VM instances running on the OpenStacks++ cannot continue to work. Secondly, the executing time for Ansible script takes least a few minutes for a single machine. For a larger cluster, it may even take longer. During that time, services of OpenStack++ are not available and users may lose their important data. Hence, the non-persistent network interface can bring these serious problems which should be completely resolved before using by any applications in a production environment.

Inconsistent building process. Even though the probability of causing a different build of OpenStack++ is low, it still exists, due to a variety of possible reasons. The most common error I experienced when I build OpenStack++ is "No valid host was found. There are not enough hosts available" appearing on the OpenStack++ dashboard. According to my tests, it occasionally happens when I build

the OpenStack++ by using the Ansible scripts, however, it will never happen on my another machine where I also build OpenStack++ several times. In the architecture of OpenStack, also in OpenStack++, the Nova service is the part of OpenStack that provides a way to provision compute instances. The reason about why OpenStack++ cannot find the valid host is that OpenStack++ couldn't find a compute node to launch the VM. In this case, no available compute node usually indicates that either some of Nova services are down or configuration for Nova services is wrong. However, the deeper reason is that, when developers build OpenStack++, the Ansible script may not work properly such that some config files are still not up to date or services still rely on the original configuration since config files changed. Because of it, Nova service is referencing the older configuration which causes the hosting error above or some networking errors between host machines and running instances on OpenStack++.

IV. Possible future work

This research presented in this thesis briefly discuss the features and code implementation of OpenStack++ but it also points out some flaws of OpenStack++ which could work further.

Firstly, to address the issue about the non-persistent network, this thesis hasn't located the root of the problem yet, but the possible reason why it happens is that for single network interface card user, OpenStack++ install a dummy kernel module to create the virtual network interface card for OpenStack++. After reboot each time, the dummy kernel will be lost so that the dummy network interface will be lost as well. According to this information, developers may have to think about a way to make the kernel be persistent so that the non-persistent network could be resolved.

Secondly, to deal with another issue about the inconsistent building process, as the thesis introduced earlier, it is related to the Nova service of OpenStack++ because it always read the old configuration files rather than forcibly read the latest version of config files. Definitely, developers have to append some new Ansible scripts to force Nova always read the new files. Also, it is important to add some additional logic to make the building process be more robust so that it could generate more useful warning logs and the error should be handled by OpenStack++ itself.

Besides the implementation flaws of OpenStack++, the performance test is limited to the video streaming in this empirical study. To apply OpenStack++ into real-world production, more performance test under different situations based on lots of running environment are necessary to work in the future. For instance, since OpenStack++ provides better performance and more stable than traditional heavy cloud services, to keep it close to the end users, mobile applications should be ideal for OpenStack++. Developers may have to develop more mobile applications, such as wearable cognitive assistants, a video live streaming project and so on. There are many possibilities and it is worthy to be explored.

V. Conclusion

Cloud computing is growing even faster than expected in these decades. More and more companies are using cloud computing technologies to develop and deploy their products. Meanwhile, there are more old brand technology companies, such as *Oracle, Microsoft, Google* etc. , have invested a lot of resources in cloud computing. However, as people may notice, the cloud still has some limits, due to cloud service locations, cost of developing and maintaining and more economic factors. The emergence of cloudlet can help us to resolve these issues and its market still has many potentials to be discovered. A cloudlet is a trusted, resource-rich computer or cluster of computers that are well-connected to the Internet and available for use by nearby mobile devices [5]. As this thesis demonstrated, it provides better performance than traditional cloud service. If the market of cloudlet is developed well, we can

image that, in the future, any computation resources are easy to be accessed. People can achieve more personal computation for different fields, such as wearable cognitive assistants for medical purpose, video lives streaming for entertainment. Cloudlet opens the door for a new diagram of cloud computing.

References

[1] Open Edge Computing, `http://openedgecomputing.org/developers.html`

[2] Elijah Project Home Page `http://elijah.cs.cmu.edu/index.html`

[3] Elijah OpenStack GitHub Repository `https://github.com/OpenEdgeComputing/elijah-openstack`

[4] Ha, Kiryong and Mahadev Satyanarayanan. "OpenStack++ for Cloudlet Deployment." (2015)

[5] Satyanarayanan, M., Bahl, V., Caceres, R., Davies, N. (2011). The Case for VM-based Cloudlets in Mobile Computing. IEEE Pervasive Computing. doi:10.1109/mprv.2009.64