OpenStack++ for Cloudlet Deployment

Kiryong Ha

krha@cmu.edu

July 15, 2015

PROJECT REPORT

Electrical and Computer Engineering Department

College of Engineering

Carnegie Mellon University

Pittsburgh, PA 15213

Reviewer:

Mahadev Satyanarayanan

Daniel P. Siewiorek

Copyright © 2015 Kiryong Ha

This research was supported by the National Science Foundation (NSF) under grant number IIS-1065336. Additional support was provided by the Intel Corporation, Google, Vodafone, and the Conklin Kistler family fund. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and should not be attributed to their employers or funding sources.

Keywords: Mobile Computing, Cloud Computing, Cloudlets, Virtual Machines, OpenStack, Virtual Machine, VM, VM Provisioining, VM Hand-off

Abstract

The notion of cloudlet is getting widely accepted from both academia and industry. However, the development of the cloudlet faces a classic bootstrapping problem. It needs practical applications to incentivize cloudlet deployment while developers cannot heavily rely on a cloudlet infrastructure until they are widely deployed. To provide a systematic way to expedite cloudlet deployment, I implemented *OpenStack++* that extends *OpenStack*, an open source ecosystem for cloud computing. In this work, I present design decisions to efficiently integrate cloudlets with OpenStack and shows implementation details for VM provisioning and handoff. Finally, I explain how I resolved practical challenges for the cloudlet porting.

Contents

1 Introduction **5**

2 Background **6**

2.1 Cloudlets 6

2.1.1 VM Synthesis 7

2.2 OpenStack 9

3 Design **10**

3.1 Modular Approach using OpenStack Extension 10

3.2 Support for both OpenStack and a Standalone Executable 12

4 Implementation **12**

4.1 Import Base VM 13

4.2 Resume Base VM 15

4.3 Create VM overlay 16

4.4 VM Provisioning 17

4.5 VM Handoff 19

5 Challenges **20**

5.1 Portability of the VM 20

5.2 Modification on hypervisor (QEMU/KVM) 22

6 Conclusion **23**

1 Introduction

A cloudlet is a small-scale cloud data center that is dispersed at the edge of the Internet [13]. It is aimed to support resource-intensive and interactive mobile applications by providing powerful computing resources to mobile devices with low latency. A cloudlet is a new architectural element that extends today's cloud computing infrastructure acting as the middle tier of a 3-tier hierarchy: mobile device — cloudlet — cloud. There are a few but important differences between cloudlets and the cloud.

1. *Rapid provisioning* [10]: The speed of provisioning matters on cloudlets. Today's cloud data centers are optimized for launching VM images that already exist in their storage tier. In contrast, cloudlets need to be much more agile in their provisioning because their association with mobile devices is highly dynamic with considerable churn due to user mobility.

2. *VM migration across cloudlets (Hand-off)*: To support user mobility, cloudlet seamlessly migrates the offloaded services on the first cloudlet to the second cloudlet as a user moves away from the currently associated cloudlet. This is referred to as *VM handoff*.

3. *Cloudlet discovery*: Since cloudlets are small data centers distributed geographically, a mobile device has to discover, select, and associate with the appropriate cloudlet among multiple candidates before provisioning.

In this work, I focus on the pragmatic aspect of the cloudlet research; deployment of a cloudlet infrastructure. Since a cloudlet model requires reconfiguration or additional deployment of hardware/software, it is important to provide a systematic way to incentivise the deployment. And here we are facing a classic bootstrapping problem. We need practical applications to incentivize cloudlet deployment. However, developers cannot heavily rely on a cloudlet infrastructure until it is widely deployed. How can we break this deadlock and bootstrap the cloudlet deployment?

The history of the Internet offers a hint. The Internet is an open ecosystem that uses a standard protocol suite (e.g. TCP/IP). Through this open standard, multiple vendors from low-level hardware companies to high-level services providers independently participate. However, no single vendor is bearing large risk for improving this ecosystem. Instead, they are creating synergy by investing in their own business. In this ecosystem, innovation in one layer can stimulate others, resulting in additional investment. For example, the wide use of the Internet services such as email and web searching has encouraged Internet service providers (ISPs) to invest in this infrastructure. Infrastructure advances have become a foundation for new Internet services like voice over IP (VoIP) and social networks.

We will take a similar strategy with cloudlets. Many of today's server-based mobile applications use the cloud as a back-end server. We observe that the ecosystem in cloud computing is similar to that of the

Internet; hardware to software vendors are actively and independently participate for profit. For example, in hardware, network vendors such as Cisco and Jupiter are deploying Software Defined Network (SDN) routers/switches, and blade server vendors like IBM and HP are reshaping their products [11]. Similarly in software, multiple hypervisors are rapidly developed to compete with each other, and various Linux vendors like RedHat and Canonical propose their own solutions for cloud computing. *OpenStack*, which is a free and open-source cloud computing software platform, provides openness in the emerging cloud software ecosystem [12]. It offers a suite of standard APIs so vendors can independently contribute to different layers without breaking compatibility. As of 2015, more than 150 companies have contributed to OpenStack.

We will leverage this open platform to expedite cloudlet deployment. That is, we will make our system work as OpenStack extensions, so that any individual or any vendor who uses OpenStack for their cloud computing can easily use cloudlets. We refer to this Cloudlet-enabled OpenStack as *OpenStack++*. This project focuses on the design and implementation of cloudlet features of the OpenStack++ API. I will also provide a client library and web interface for the OpenStack users.

2 Background

2.1 Cloudlets

The convergence of cloud computing and mobile computing has begun. Apple's *Siri* [6], which performs compute-intensive speech recognition in the cloud, hints at the rich commercial opportunities in this emerging space. On the user end, mobile devices are becoming smaller and smaller as a form of wearable devices [8]. At the other end in the back-end server, cloud computing provides nearly infinite computing resources and scalability to mobile applications. Through context-aware real-time scene interpretation such as recognition of objects, faces, activities, signage text, and sounds, we can imagine new mobile applications that offer helpful guidance for everyday life much beyond what today's Siri can offer [9]. These interactive and resource-intensive applications will leverage the power of the cloud.

One of the critical challenges in cloud-backed mobile computing is the end-to-end network responsiveness between the mobile device and associated cloud. When the use of cloud resources is in the critical path of user interaction, operation latencies can be no more than a few tens of milliseconds. Violating this bound results in distraction and annoyance to a mobile user who is already attention-challenged [5, 7]. To support low-latency requirements for these applications, a new architectural element called *cloudlets*, was proposed [13]. Cloudlets represent the middle tier of a 3-tier hierarchy, mobile device — cloudlet — cloud, to achieve crisp response time.

There is a significant overlap between the requirements for the cloud and cloudlets. Both level need (a) strong isolation between untrusted user-level computations; (b) mechanisms for authentication, access control, and metering; (c) dynamic resource allocation for user-level computations; and, (d) the ability

to support a wide range of user-level computations with minimal restrictions on their process structure, programming languages or operating systems. Today's cloud data center meets these requirements with the virtual machine (VM) abstraction. We believe, for the same reasons they are used in cloud computing today, that VMs are the right level of abstraction for cloudlets. Meanwhile, there are a few but important differentiators between cloud and cloudlet.

1. *Rapid provisioning*: The speed of provisioning matters at cloudlets. Today, cloud data centers are optimized for launching VM images that already exist in their storage tier. They do not provide fast options for instantiating a new custom image. One must either launch an existing image and laboriously modify it, or suffer the long, tedious upload of the custom image over a WAN. In contrast, cloudlets need agile provisioning. Their association with mobile devices is highly dynamic, with considerable churn due to user mobility. A user from far away may unexpectedly show up at a cloudlet (e.g., if he just got off an international flight) and try to use it for an application such as a personalized language translator. For that user, the provisioning delay before he is able to use the application impacts usability.

2. *VM migration across cloudlets (Hand-off)*: Once a user successfully uses a provisioned cloudlet, the next question is "What happens if a mobile device user moves away from the cloudlet he is currently using?" As long as network connectivity is maintained, the applications should continue to work transparently. However, interactive response time will degrade as the logical network distance increases. To support this effect of user mobility, cloudlets seamlessly migrate the offloaded services on the first cloudlet to the second cloudlet, a capability termed *VM handoff*.

3. *Cloudlet discovery*: Dynamic discovery of a cloudlet by a mobile client is a unique problem to cloudlet. Because cloudlets are small data centers distributed geographically, a mobile device first has to discover, select, and associate with the appropriate cloudlet among multiple candidates before it starts provisioning. These steps are unnecessary with a cloud because it is centralized. But in cloudlets, discovery and selection have to be carefully managed because the choice of a cloudlet can directly affect the user's waiting time before starting offloading operation as well as future performance of the associated mobile application.

2.1.1 VM Synthesis

Cloudlets use the notion of VM synthesis for rapid provisioning and VM handoff. The key idea behind VM synthesis that a large part of a VM image is devoted to the guest OS, software libraries, and supporting software packages. The customizations of a base system needed for a particular application are usually small. Therefore, if the *base VM* already exists on the cloudlet, only its difference relative to the desired

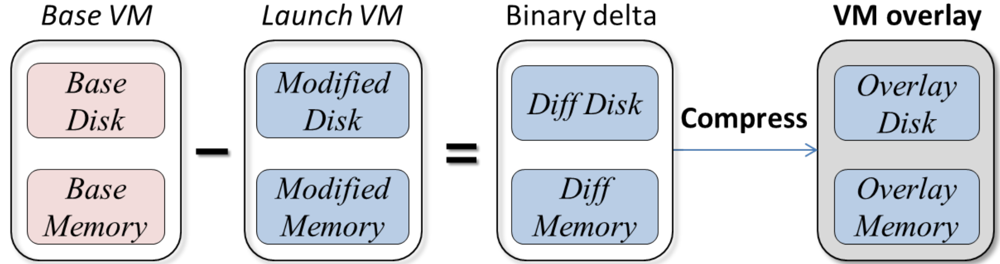


Figure 1: Creating *VM overlay* from *Base VM*

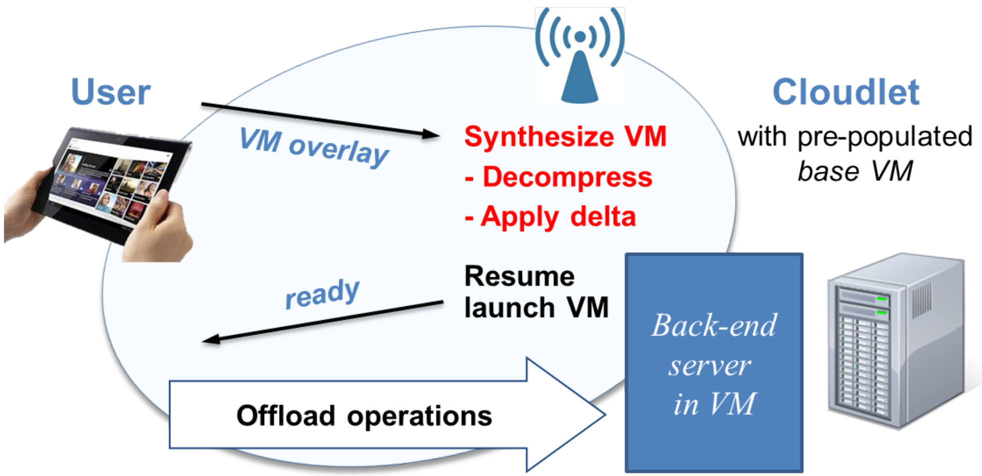


Figure 2: VM Synthesis from Mobile Device

custom VM, called a *VM overlay*, need to be transferred. This approach of using VM overlays to provision cloudlets is called *VM synthesis*. Rapid VM provisioning implementations in cloudlets show provisioning times of less than 10 seconds with a series of optimizations based on my VM synthesis approach [10].

Figure 1 shows the relationship between a *base VM* and a *VM overlay*. Base VM images are constructed from popular operating systems such as standard builds of Linux (Ubuntu 12.04 Server) and Windows 7. An instance of each image is booted and then paused; the resulting VM disk image and memory snapshot serve as the *base disk* and the *base memory*. To construct a *launch VM*, which is a VM image ready to serve mobile requests, a user resumes an instance of the appropriate base image, install and configure the application's back-end server binaries, and launch the back-end server. At that point, a user pauses the VM. The resulting disk image and memory snapshot constitute the launch VM image. As soon as an instance is resumed from this image, the application will be in a state ready to respond to offload requests from the mobile device — there will be no reboot delay. The overlay for each application is the compressed binary difference between the launch VM image and its base VM image, produced using `xdelta3` and LZMA compression.

Figure 2 shows the relevant steps of VM synthesis. A mobile device delivers the VM overlay to a

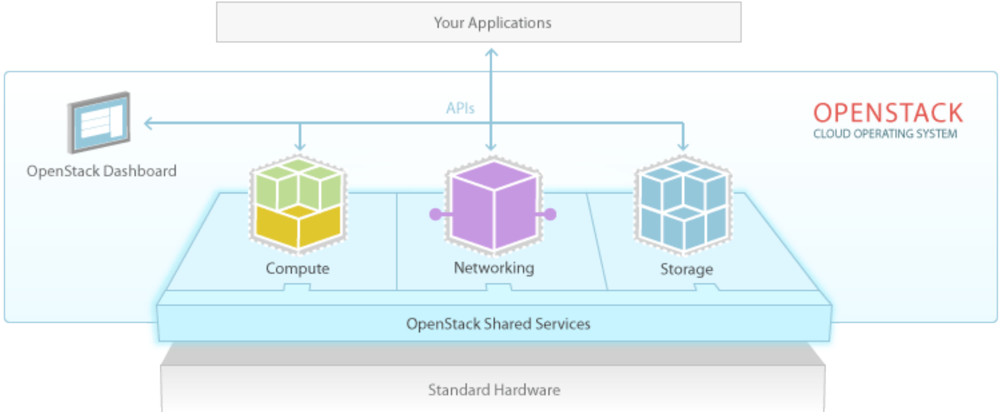


Figure 3: OpenStack Software Overview Diagram

Code name	Category	Description
Nova	Compute	Provision and manage large pools of on-demand computing resources
Swift	Object Storage	Petabytes of reliable storage on standard gear
Cinder	Block Storage	Persistent block-level storage devices for use with OpenStack compute instances
Neutron	Networking	A system for managing networks and IP addresses
Horizon	Dashboard	Self-service, role-based web interface for users and administrators
Keystone	Identity	Multi-tenant authentication system that ties to existing stores (e.g. LDAP) and Image Service
Glance	Image Service	Discovery, registration, and delivery services for disk and server images

Figure 4: OpenStack core modules

cloudlet that already possesses the base VM from which this overlay was derived (the delivery can be either from the cloud or from the storage of mobile device). The cloudlet decompresses the overlay, applies it to the base to derive the launch VM, and then creates a VM instance from it. The mobile device now can begin performing offload operations on this instance.

2.2 OpenStack

OpenStack is a open-source cloud computing software platform primarily focusing on IaaS (Infrastructure as a Service). It began in 2010 as a joint project of Rackspace Hosting and NASA. Currently it is managed by OpenStack Foundation, a non-profit corporate entity established in 2012. OpenStack has a time-based release cycle every six months and the latest version is *Kilo* (released on 2015 May).

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control. It has a modular architecture with code names for its components. The major modules and their description are listed at Figure 4.

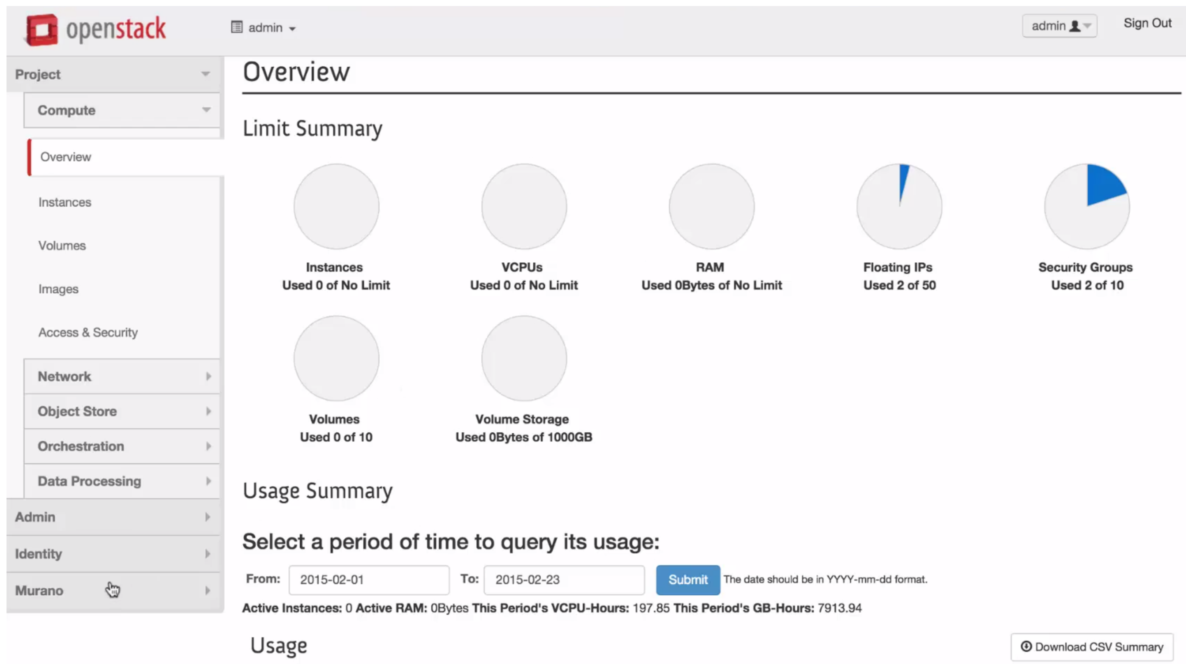


Figure 5: OpenStack Dashboard Example

As of 2015, more than 500 companies have joined the project, including AT&T, AMD, Canonical (Ubuntu), Cisco, Citrix, Dell, EMC, HP, Huawei, IBM, Intel, Red hat, and Yahoo [4]. In every single business category, the top three vendors support or participate in OpenStack. As this large number of participants indicates, OpenStack is becoming the *de facto* standard in open-source cloud computing platform.

3 Design

OpenStack has a 6-month time-based release cycle, which is a fairly short period, and new release introduces significant changes in internal APIs. Therefore, in order to keep track of their release cycle with minimal effort on the cloudlet binding code, I am taking a modular approach for the OpenStack integration by extending the original code rather than modifying the code directly. In addition, I maintain a standalone cloudlet executable, which runs without an OpenStack cluster along with OpenStack++, as explain in Section 3.2.

3.1 Modular Approach using OpenStack Extension

OpenStack provides an extension mechanism to add new features to support innovative approaches. This allows developers to experiment and develop new features without worrying about the implications to the standard APIs. Since the extension is queryable, a user can first send a query to a particular OpenStack cluster to check the availability of the cloudlet features. Figure 6 shows the OpenStack API call hierarchy. APIs for extensions are provided to the users by implementing an Extension class. API request from the user

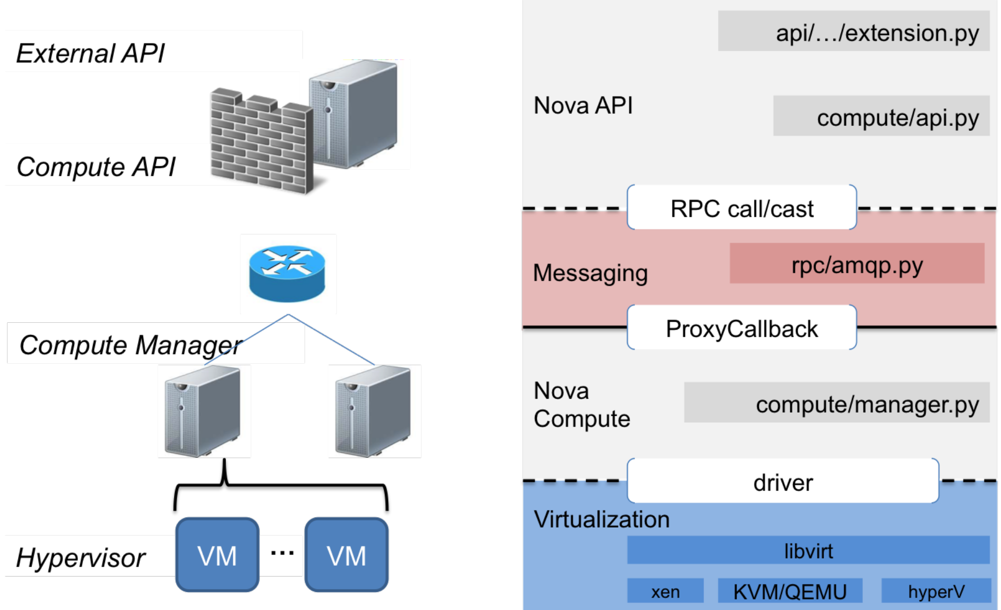


Figure 6: OpenStack API call hierarchy

will arrive at the extension class and a set of internal APIs will be called to accomplish desired functionality. Some of the internal API calls will be passed to a corresponding compute node via messaging layer if necessary. Then, the API manager at the compute node will receive the message and handle it by sending commands to the hypervisor via a driver. Finally, the driver class will return the result and pass to the user following the reverse call sequence.

The cloudlet extensions follow the same call hierarchy. Once a user sends a request via a RESTful interface, the message will be propagated to the matching compute node. Then the hypervisor driver performs the given task. Here's an example command flow for creating a VM overlay. The command is applied to a running virtual machine and generates a VM overlay which extracts the difference between the running VM and the base VM. To define a new action to a virtual machine, a cloudlet extension class is declared following the OpenStack extension rule. The user-issued API first arrives at the extension class, and then is passed to a corresponding physical machine via API and message layer. The message is handled by a cloudlet hypervisor driver, which interacts with a target virtual machine. Finally, the cloudlet hypervisor driver will create a VM overlay using the VM snapshot.

To support a specific API, the API manager and hypervisor driver should be able to handle the message in addition to the cloudlet extension. This requires modifications to the original files/classes of OpenStack. Modifying original OpenStack code, however, will cause significant maintenance overhead, especially because OpenStack is frequently updated. Instead, I create a new class for both the API manager and hypervisor driver inheriting a matching class in OpenStack and save each of them at a new file. Fortunately, OpenStack provides a way to use a custom class for the API manager and hypervisor driver via a configuration file.

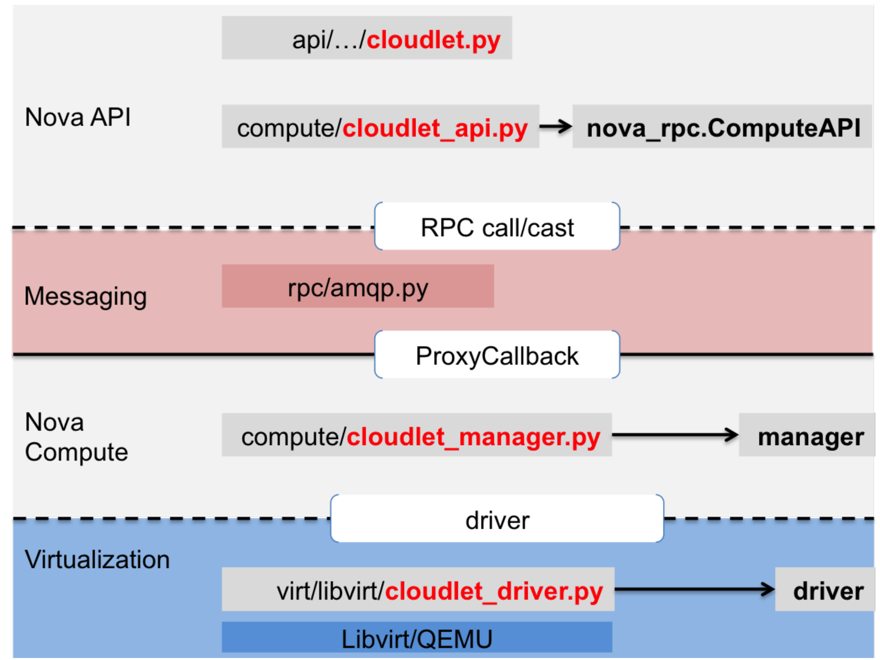


Figure 7: Classes/Files for Cloudlet API call hierarchy

As shown in Figure 7, cloudlet specific code is at separate files such as cloudlet_api.py, cloudlet_manager.py, and cloudlet_driver.py. This makes management overhead much lower because the cloudlet feature can be added by simply placing those files at OpenStack directory and changing a configuration file.

3.2 Support for both OpenStack and a Standalone Executable

It is important to support a standalone version of cloudlet execution in addition to the OpenStack extended version. There are two rationales for maintaining a standalone executable along with OpenStack++. First, OpenStack is designed for providing end-to-end cloud computing services, so it is not trivial to install and maintain OpenStack itself. For those users who want to use only cloudlet features in a simple way, a standalone executable will be a neat solution. Second, a standalone executable is much easier to debug and simple to assess the performance. Since OpenStack is a complex system, the standalone version provides a straightforward way to debug the system. To support both approaches effectively, I created a cloudlet library that both OpenStack and standalone code can use. This library is packaged as a python library because OpenStack uses python. Figure 8 shows a high-level diagram the cloudlet library is used.

4 Implementation

OpenStack++ implements the following features.

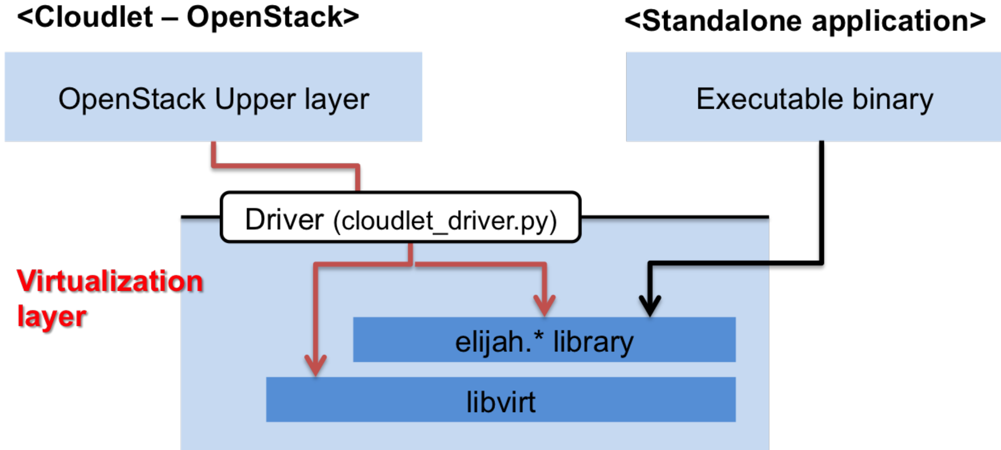


Figure 8: Supporting both OpenStack and Standalone

1. **Import Base VM**: Import a Base VM from a file. I assumed that each Cloudlet has a set of prepopulated Base VMs as explained in Section2.1, and this is a function for importing a Base VM.
2. **Resume Base VM**: Resume one of the base VMs to make a customized VM and to create a new VM overlay for the customized VM.
3. **Create VM overlay**: Create a VM overlay from a running VM instance.
4. **VM synthesis**: Provisioning a VM instance at a OpenStack++ cluster using a VM overlay.
5. **VM handoff**: Migrating a VM instance to a different OpenStack++ cluster.

Among those features, *Resuming Base VM* and *Creating VM overlay* are off-line operations that the developers use to create a VM overlay of the back-end server. That means those two operations will not be used by mobile users, but used by the application developers. *Importing Base VM* is for pre-provisioning the base VM and it is also an off-line operation where the administrator of the OpenStack++ cluster uses after the installation of a new OpenStack. The remaining two features, *VM synthesis* and *VM handoff*, are the run-time operations that are executed during the mobile application's offloading. Figure 9 shows how these operations are interpreted from the perspective of OpenStack. For example, resuming a base VM and performing VM synthesis can be considered a process of instantiating a new virtual machine on OpenStack.

4.1 Import Base VM

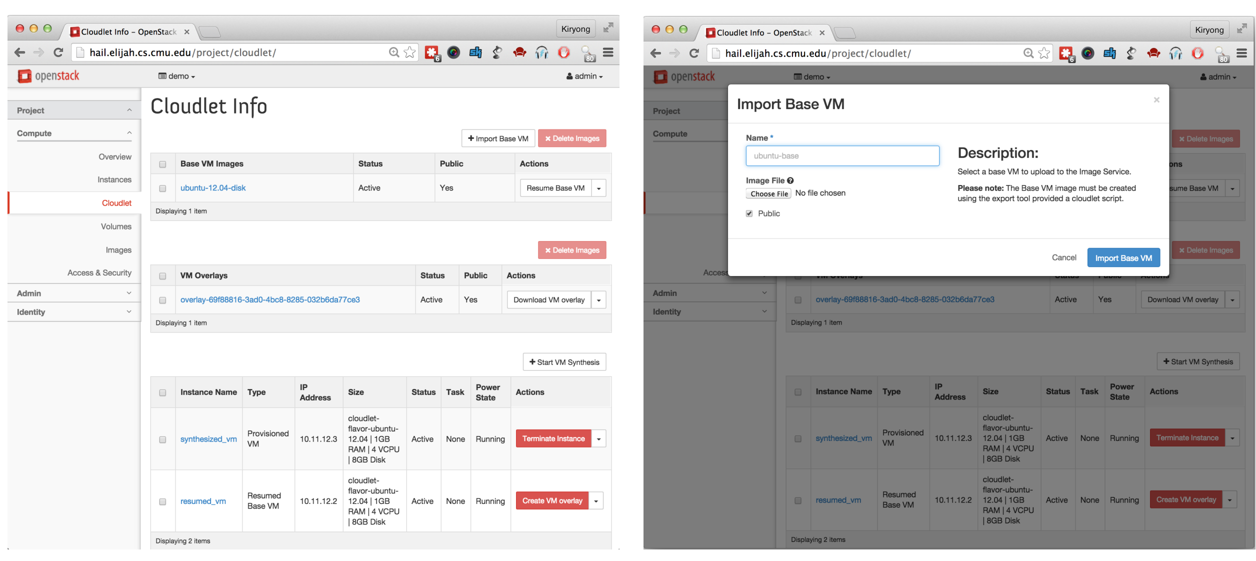
We presume that every cloudlet caches a set of base VMs. The OpenStack++ administrator imports base VMs after installing OpenStack++ by using the *import Base VM* operation. Figure 10-(a) shows a screenshot of the Cloudlet panel in the OpenStack dashboard. The first table shows a list of base VMs on this OpenStack cluster. The second table displays a list of VM overlays saved in the Glance storage, where OpenStack saves virtual machine images. The last table presents running VMs, each of which is either a resumed base VM or

Cloudlet features	OpenStack interpretation	Output in OpenStack
Import Base VM	Save VM snapshot to the Glance image storage	New VM image
Resume Base VM	Resume a VM from memory and disk snapshot	New VM instance
Create a VM overlay	Incremental snapshot of the VM	1) Disk incremental snapshot 2) Memory incremental snapshot
VM synthesis	Recover the VM from the incremental snapshot (a.k.a VM overlay)	New VM instance
VM handoff	Migrate a running VM instance from one OpenStack cluster to another	1) Terminate VM at source OpenStack 2) create new VM instance at destination OpenStack

Figure 9: Cloudlet features as an analogy for OpenStack

a synthesized VM. The *Import Base VM* button is at the right corner of the first table, which an administrator can use to import a new base VM. Figure 10-(b) shows the UI for import base VM. Input file path for a base VM and base VM's name are required.

From OpenStack's viewpoint, importing a base VM is equivalent to saving new blobs at a Glance storage. Hence, instead of creating a new API, I reuse existing Glance APIs to accomplish this operation. The only difference is that multiple files should be saved to import a base VM because a single base VM is a zipped file composed of four files internally; a base disk image, a base memory snapshot, a hash value list for the disk image, a hash value list for the memory snapshot. A received base VM file is first decompressed into four files, and then saved to the Glance storage one by one. To indicate that these are all related to the



(a) Dashboard

(b) Import Base VM

Figure 10: Screenshot of Cloudlet Dashboard and Importing Base VM

```
krha@hail:~$ nova image-show ubuntu-12.04-diskhash
+---------------------------+------------------------------------------------------------------+
| Property                  | Value                                                            |
+---------------------------+------------------------------------------------------------------+
| OS-EXT-IMG-SIZE:size      | 118018164                                                        |
| created                   | 2015-06-30T21:24:10Z                                             |
| id                        | 666f1d7e-d546-4bf7-a22a-cfa9ee253e75                             |
| metadata base_sha256_uuid | abda52a61692094b3b7d45c9647d022f5e297d1b788679eb93735374007576b8 |
| metadata cloudlet_type    | cloudlet_base_disk_hash                                          |
| metadata image_location   | snapshot                                                         |
| metadata image_type       | snapshot                                                         |
| metadata is_cloudlet      | True                                                             |
| minDisk                   | 8                                                                |
| minRam                    | 1024                                                             |
| name                      | ubuntu-12.04-diskhash                                            |
| progress                  | 100                                                              |
| status                    | ACTIVE                                                           |
| updated                   | 2015-06-30T21:24:26Z                                             |
+---------------------------+------------------------------------------------------------------+
krha@hail:~$
```

(a) Metadata for a Base VM's memory snapshot file

Custom Properties

Image Type	snapshot
base_resource_xml_str	<domain type='kvm'> <name>cloudlet-d71b7be54b104dee82c9038fddbbf39c</name> <uuid>d71b7be5-4b10-4dee-82c9-038fddbbf39c</uuid> <memory unit='KiB'>1048576</memory> <currentMemory unit='KiB'>1048576</currentMemory> <vcpu placement='static'>4</vcpu> <os> <type arch='x86_64' machine='pc-1.0'>hvm</type> <boot dev='hd'/> </os> <features> <acpi/> </features> <cpu mode='custom' match='exact'> <model fallback='forbid'>core2duo</model> <topology sockets='1' cores='4' threads='1'/> </cpu> <clock offset='utc'/> <on_poweroff>destroy</on_poweroff> <on_reboot>restart</on_reboot> <on_crash>destroy</on_crash> <devices> <emulator>/usr/local/bin/cloudlet_qemu-system-x86_64</emulator> <disk type='file' device='disk'> <driver name='qemu' type='raw'/> <source file='/home/cloudlet/cloudlet/image/pci_hotplug/precise.raw'/> <target dev='hda' bus='ide'/> <address type='drive' controller='0' bus='0' target='0' unit='0'/> </disk> <disk type='file' device='cdrom'> <driver name='qemu' type='raw'/> <source file='/var/lib/cloudlet/conf/ovftransport.iso'/> <target dev='hdc' bus='ide'/> <readonly/> <address type='drive' controller='0' bus='1' target='0' unit='0'/> </disk> <controller type='ide' index='0'> <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/> </controller> <interface type='user'> <mac address='52:54:00:9f:a8:da'/> <model type='virtio'/> <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/> </interface> <input type='mouse' bus='ps2'/> <graphics type='vnc' port='-1' autoport='yes' listen='127.0.0.1' keymap='en-us'> <listen type='address' address='127.0.0.1'/> </graphics> <video> <model type='cirrus' vram='9216' heads='1'/> <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/> </video> <memballoon model='virtio'> <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/> </memballoon> </devices> <seclabel type='none'/></domain>
base_sha256_uuid	abda52a61692094b3b7d45c9647d022f5e297d1b788679eb93735374007576b8
cloudlet_base_disk_hash	666f1d7e-d546-4bf7-a22a-cfa9ee253e75
cloudlet_base_memory	8f7bcdd1-72b8-448e-b704-0054ee2d133c
cloudlet_base_memory...	8c49ceca-5e02-4415-8e3a-1a7f919892ad
cloudlet_type	cloudlet_base_disk
image_location	snapshot
is_cloudlet	True

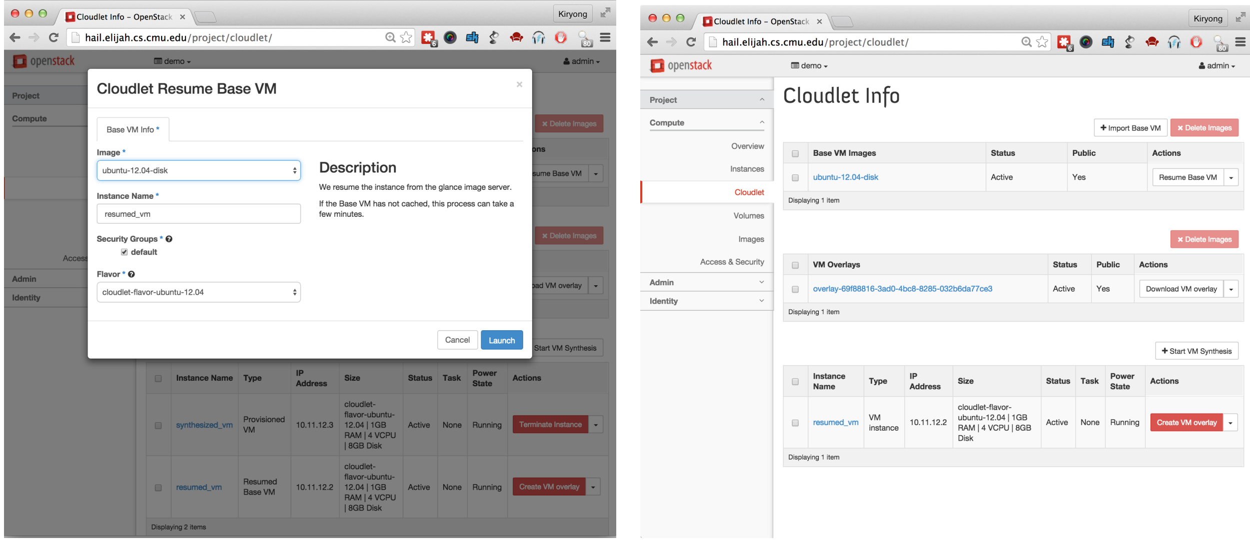
(b) Metadata for a Base VM's disk image file

Figure 11: Glance metadata of Base VM

Cloudlet's base VM, I marked each file with a metadata using file type keyword as shown in Figure 11-(a). Further, since OpenStack treats each glance image as an independent entity, the disk image saves the UUIDs of all other associated files to connect all the participating files. In addition, I save VM's libvirt configuration at the metadata of the Base VM's disk image for the later use. Figure 11-(b) shows an example of metadata of a disk image.

4.2 Resume Base VM

Before creating a VM overlay, a developer installs a back-end server of a front-end application. This installation process typically includes preparing dependent libraries, downloading/setting executable binaries, and changing OS/system configurations. A developer can perform these operations using a resumed base VM. Figure 12 shows a screenshot for resuming a base VM. At the first table of base VM list, *Resume Base*



(a) UI for resuming a Base VM

(b) Finishing Base VM Resume

Figure 12: Screenshot of 'Resume Base VM'

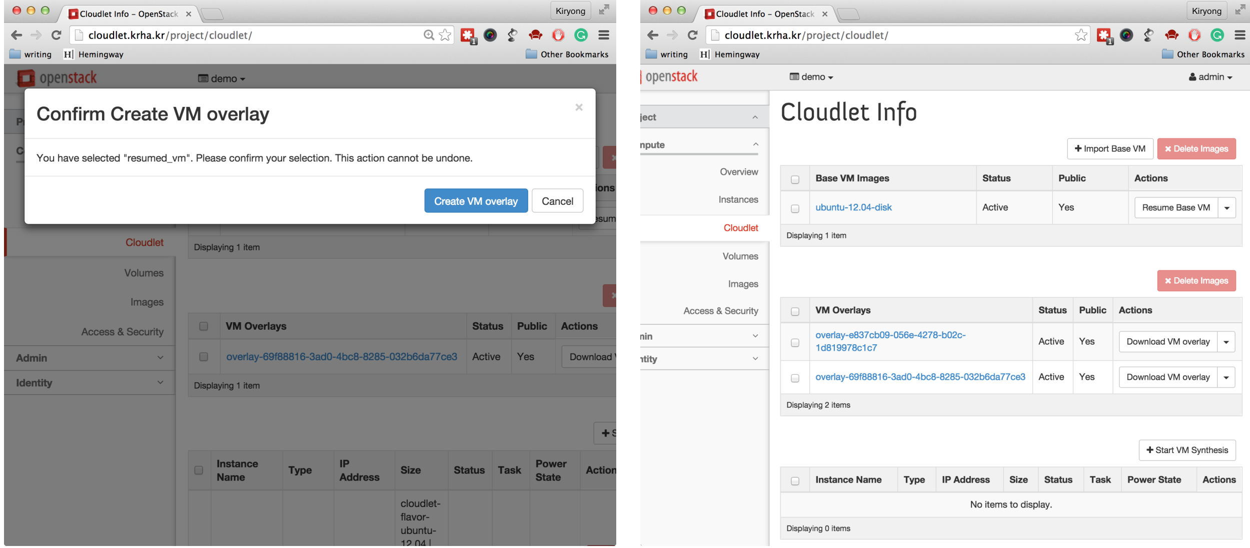
VM button will resume the selected base VM. The resumed instance will be displayed at the third table.

For OpenStack, resuming a base VM is analogous to instantiating a new VM instance using a VM snapshot. Therefore, instead of devising a new API, I modified the original API for launching a new VM instance. The original API will handle all error checking conditions such as permission, quota, and resource availability. After passing all condition checking, the message will finally arrive at the compute node to launch a new VM. At the code level, this message will arrive at hypervisor driver class and passed to the underlying virtualization. To handle resuming a Base VM task at the hypervisor driver, I build a cloudlet hypervisor driver class, *CloudletDriver*, that inherited the original *LibvirtDriver*. Upon a new message, CloudletDriver examines the metadata of the associated virtual disk image. If the virtual disk image has cloudlet flag, then CloudletDriver resumes the selected base VM instead of starting a new VM instance from booting.

4.3 Create VM overlay

After resuming the selected base VM, a developer can install a desired back-end server on it. A developer is supposed to start creating a VM overlay after finishing all the installation and after launching the back-end server process. Overlay creation will start by simply clicking a *Create VM overlay* button next to the VM instance row as shown in Figure 13-(a). This operation will apply optimizations to generate a minimal VM overlay and finally save the VM overlay at Glance storage as listed at the second table in Figure 13-(b). One can download the VM overlay using *Download* button.

For *creating VM overlay* operation, I introduce a new API. Creating VM overlay can be classified as the



(a) Start VM overlay creation (b) Finish VM overlay creation

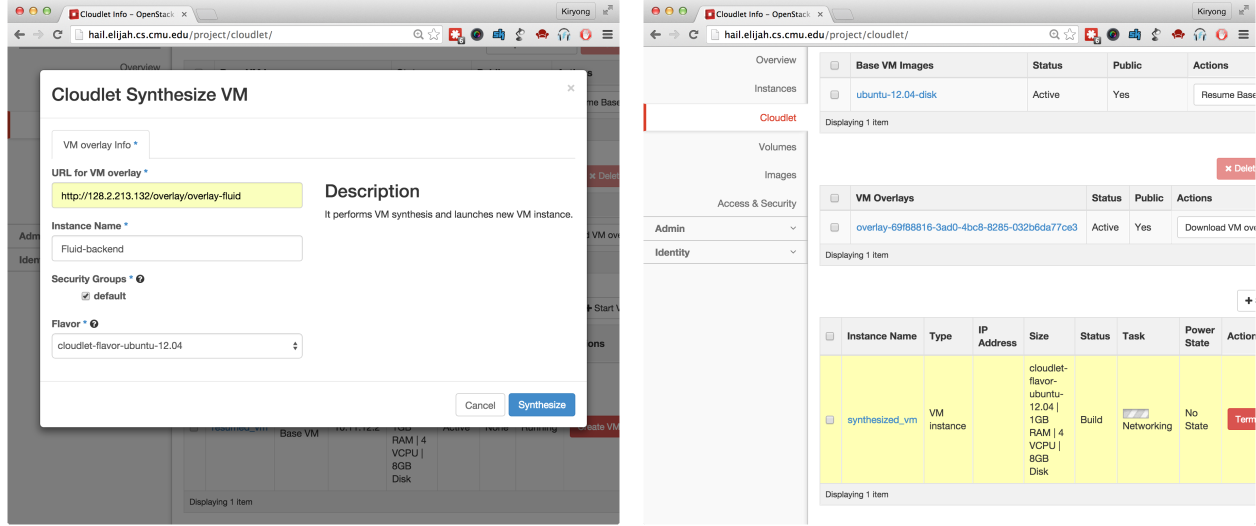
Figure 13: Screenshots of 'Creating VM overlay'

same category of API call as reboot VM and resize VM, which applies a specific action to a running VM. Therefore, I used the existing Action URL but declared a new action type using the OpenStack Extensions mechanism. The extension defines a new Action for creating a VM overlay and passes this command to the virtualization driver (a.k.a. CloudletDriver) via the internal API class. For the internal API, a cloudlet API class, *CloudletAPI*, that inherits *nova_rpc.ComputeAPI* was added. I avoid modifying the original OpenStack code or files, by class inheritance and by using a new file.

4.4 VM Provisioning

VM provisioning is a run-time operation for a rapid provisioning of an application's back-end server to a nearby Cloudlet. Since VM provisioning uses the VM synthesis approach, a VM overlay is required. Figure 14-(a) shows the UI for the VM provisioning. A user is asked to input a URL for the VM overlay. Instance flavor will be automatically selected by reading the metadata of the associated base VM. Once VM provisioning is finished, a new VM will launch and the relevant information is displayed in the third table of Figure 14-(b).

Similar to the resuming base VM in Section 4.2, VM provisioning launches a new VM instance at the OpenStack cluster. Therefore, I leverage OpenStack's original VM creation API. In OpenStack, to create a new VM instance, a user sends a HTTP POST message to a specific URL, *https://openstack-addr/v2.1/servers*. The detailed configurations for the new VM such as name, disk image, and flavor are described in JSON payload of a HTTP message. To differentiate a VM synthesis request from a regular VM



(a) UI for VM provisioning (b) VM provisioning

Figure 14: Screenshot of 'VM provisioning'

Base disk's UUID

URL to VM overlay

```
{
    "server": {
        "name": "synthesized-vm",
        "imageRef": "2d588a21-aebc-45d0-8243-33dc48c7ed2e",
        "key_name": "test",
        "flavorRef": "1",
        "metadata": {
           "overlay_url": http://storageserver.com/overlay-AR,
        }
    }
}
```

Figure 15: Example of VM provisioning request message

creation request, I add a special keyword, *'overlay_url'*, to the metadata of the message. Figure 15 shows an example of a VM provisioning message. To specify a location of VM overlay, an *'overlay_url'* is appended at the metadata.

This message is finally handled at cloudlet hypervisor driver, *CloudletDriver*, like the resuming base VM operation. At the code level, CloudletDriver inherits the Libvirt hypervisor driver, *LibvirtDriver*, overriding the VM spawning method. At the VM spawning method, it checks the metadata to find a keyword *'overlay_url'*. If the request has the overlay_url metadata, it will perform VM synthesis using the given URL of VM overlay.

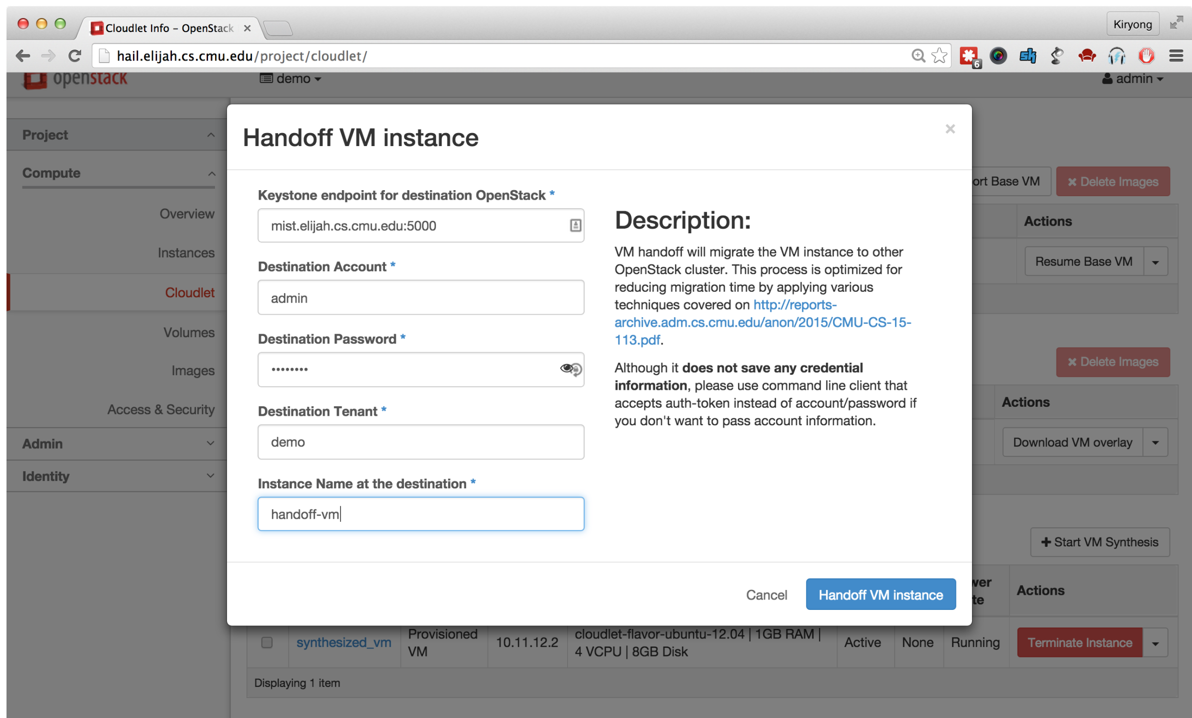


Figure 16: Screenshots of 'VM Handoff' configuration

```
{
    "cloudlet-handoff": {
        "handoff_url": "network://dest-openstack.com/"
        "dest_token": "akwqub1As8a61jsakaAj1Saa11"
    }
}
```

Authentication-token for the destination

Figure 17: Example of VM handoff request message

4.5 VM Handoff

VM handoff will migrate a running VM instance from one OpenStack cluster to another. Since it involves two independent OpenStack clusters, the operation starts from the assumption that a user has a permission to access both clusters. In other words, the source OpenStack cluster needs permission to call the API of the destination OpenStack cluster. To get the permission of the destination OpenStack cluster, the handoff UI at the source OpenStack cluster asks a credential information of the destination as shown in Figure 16. Although the Web UI does not save any credential information, one can use a command line tool that accepts auth-token instead of account/password.

Similar to creating the VM overlay in Section 4.3, VM handoff applies an action on a running VM instance. Accordingly, I create a new API extending the Action URL using OpenStack Extension.

the JSON payload of the HTTP POST message, *https://openstack_addr/v2.1/servers/server_id/action*, the handoff command and detail descriptions are added as shown in Figure 17.

5 Challenges

While porting the cloudlet open source code to OpenStack, there have been many design and implementation challenges. Some are related to complying with OpenStack practices and some are about implementation issues such as library compatibility. Though not every challenge is tightly coupled to the research, it is worth to report some of those challenges, because they are relevant to cloudlet deployment.

5.1 Portability of the VM

The first challenge is about resuming a suspended virtual machine. Technically, VM provisioning resumes a VM instance at the target OpenStack++ cluster, which is suspended at a different site. This causes several portability issues in the VM. Since a VM has a relatively narrow interface to run on a hypervisor compared to the high-level approach such as a process, it is known to be relatively easy to migrate from one place to another. That is why VM migration is used and is stable in today's data center. However, in our case, different from the data center VM migration, host machines can be highly heterogeneous and the source and destination machine can have a different networking environment. Those changes will introduce subtle issues in the VM portability. There are two major portability problems in the OpenStack++ porting; 1) CPU compatibility and 2) Stale networking state.

CPU compatibility: To get full performance from a host machine, a virtual machine usually inherits CPU flags from the host machine. That means if a source host machine, where VM is suspended, supports a wider range of CPU features than a destination host machine, where VM is resumed, then a resumed OS will crash when it tries to use a CPU feature that is not available at the destination. This happens because 1) suspend/resume is agnostic to the guest OS and 2) the guest OS checks CPU flags only once at the boot-time. Unfortunately, QEMU/KVM does not strictly check CPU flags when resuming a VM, either. Figure 18 shows an example of guest OS failure (kernel panic) when a VM is resumed at a host machine with insufficient CPU features.

To handle this CPU incompatibility issue, I use a pre-defined CPU model for the base VM. *Libvirt* library defines a set of CPU models for the virtual machine [2]. Among the wide range of CPU models including *pentiumpro, coreduo, n270, core2duo, qemu64, Conroe, Penryn, Nehalem, Westmere, SandyBridge*, and *Haswell*, I choose *Core2duo* because it covers a reasonable range of CPU flags (even more than *qemu64*), but is common enough to support old machines. If one has a more managed environment where minimal CPU features can be forced on all cloudlet machines, then a more decent CPU model like *SandyBridge* will

(a) Success to Resume

(b) Fail to Resume (Kernel Panic)

Figure 18: Challenges on CPU Flag Compatibility

be helpful to maximize performance. In my implementation, I enforce the *Core2duo* CPU model and VM provisioning and VM handoff will not start if the host machine does not support it early rather than fail in run-time.

Staleness in networking configuration: A virtualized network interface card (NIC) is attached to the virtual machine using emulated hardware interfaces like PCI. This virtual NIC has unique hardware configurations such as the MAC address, and a guest OS loads them at boot-time assuming that it won't be changed until the next booting. Then the guest OS will setup networking via this NIC. For example, the guest OS can have a private IP address with NAT or it can access the Internet directly using a public IP address. Various network configurations are possible in OpenStack using *Neutron* [3]. However, the NIC and networking information configured at a source OpenStack cluster are not valid at a destination OpenStack cluster where the VM is resumed. This is applied to 1) resuming a base VM, 2) VM provisioning, and 3) VM handoff, because they are all technically resuming a memory state of the VM. Figure 19 shows a diagram of broken networking when a VM is provisioned with the memory state. The resumed VM originally has a NIC with MAC address 11-22-33-44-55, but OpenStack networking assigned a new NIC card for this newly instantiated VM. This new NIC card has MAC address, aa-bb-cc-dd-ee, and the underlying OpenStack network module uses this MAC address to configure networking. For example, a router will forward a network packet designated to the VM using MAC aa-bb-cc-dd-ee, but it won't be delivered to the application because the guest OS thinks that its MAC is 11-22-33-44-55.

To overcome this inconsistency and to enable networking for the VM, we detach the old virtual NIC and attach a new virtual NIC given by the new OpenStack via Hot PCI Plugin [14]. Since it is an industry

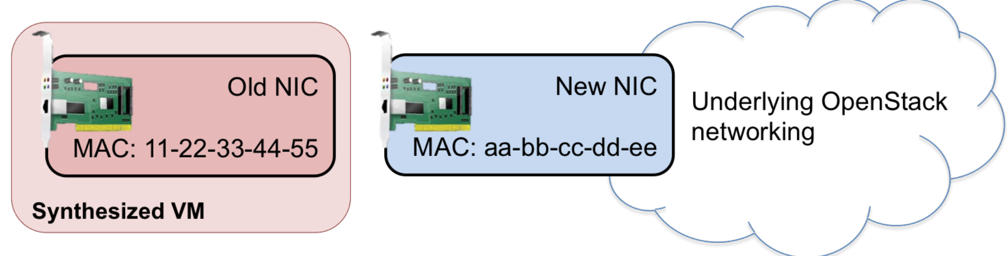


Figure 19: Example of networking staleness in synthesized VM

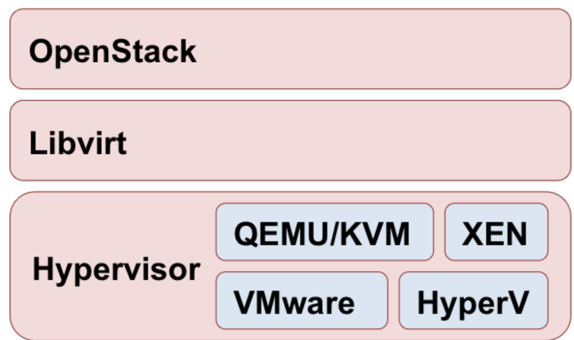


Figure 20: Layers in Cloud Computing software

standard, most modern OSes supports it. It is also supported by the KVM/QEMU hypervisor using VT-d techniques [1]. After resuming the VM successfully (either resumed from VM provisioning or VM handoff mechanism), I detach the old virtual NIC of the VM and attach a new NIC configured by the OpenStack networking. This way, the operating system understands the reattachment of the PCI device, and accordingly updates stale networking configurations. In theory, this won't affect the applications' networking because applications are running on the TCP/IP layer and are agnostic to the underlying network level.

5.2 Modification on hypervisor (QEMU/KVM)

Figure 20 shows the layers of cloud computing software. At the bottom, a hypervisor is responsible for creating and running virtual machines. Commercial products and open source projects including VMWare ESX, Microsoft Hyper-V, and QEMU/KVM are the examples at this layer. One level above, *Libvirt* is an open source management tool for managing virtual machines. It supports KVM, Xen, VMware ESX and other virtualization hypervisors. Libvirt provides a set of APIs for the orchestration of hypervisors. This is useful because each hypervisor has slightly different syntax for a similar function. Libvirt provides a high-level abstraction hiding complexity and diversity of various hypervisors. At the top layer, OpenStack tries to provide a complete end-to-end software system providing 1) resource management for computing (e.g. virtual machines), networking, storage, 2) authentication and permission, and 3) high-level API for easy use and so on.

In this hierarchy of the cloud computing software, OpenStack++ patches the original OpenStack to enable cloudlet features. However, to get the best performance for both provisioning and handoff, the cloudlet code modifies the QEMU/KVM hypervisor, which can cause a problem for merging OpenStack upstream. This is because OpenStack and QEMU/KVM are maintained by independent organizations. Therefore, the modifications to the QEMU/KVM hypervisor can be an obstacle for OpenStack upstream merging because this modification is not an official release from QEMU/KVM community. The right approach is 1) first merge cloudlet's modified QEMU/KVM to QEMU/KVM upstream, 2) wait for the new release of cloudlet-enabled QEMU/KVM, 3) merge OpenStack++ to OpenStack upstream with the requirement of cloudlet-enabled QEMU/KVM. This is not a research challenge or an implementation issue but matters in practice when pursuing OpenStack upstream merging.

In order to have a better understanding, I list the necessity of modified QEMU/KVM in the cloudlet implementation. For VM provisioning, the QEMU/KVM modification helps by improving the memory snapshot format. Memory snapshot in QEMU/KVM is an internal data structure and its format is poorly maintained in terms of compatibility. As a result, a QEMU memory snapshot saved at a certain version might not be resumable at a different version. In addition, the original QEMU/KVM compresses each memory page if possible, and that prevents the cloudlet code from performing deduplication and randomly accessing a specific memory page. Further, cloudlets modify QEMU/KVM to support *early start* optimization which allows a VM instance to start without a full memory snapshot. Early start optimization uses on-demand fetching of the memory snapshot to speed up VM provisioning.

For VM handoff, a behavior of the original VM live migration has been changed. While the VM handoff is proceeding, new dirty memory pages of the running VM are not sent to the network immediately but accumulated under the control of VM handoff code. This is different from the original behavior of live migration where dirty memory pages are immediately sent. This change is designed to save network bandwidth by avoiding transmission of frequently modified memory pages (hot region) repeatedly. Also, the VM handoff module will achieve adaptive VM handoff that balances computation speed and network transmission speed.

Modifications to QEMU/KVM are inevitable to speed up provisioning and handoff, but it can be an obstacle for OpenStack upstream merging from a practical standpoint. To minimize the influence of modified QEMU/KVM at OpenStack, I used it only for the cloudlet related tasks. That is, modified and unmodified QEMU/KVM coexists at OpenStack++ and OpenStack's original tasks use unmodified QEMU/KVM as before.

6 Conclusion

Cloudlets are becoming widely accepted from academia and industry. However, the development of cloudlets face a classic bootstrapping problem. It needs practical applications to incentivize cloudlet deployment while

developers cannot heavily rely on cloudlet infrastructure until it is widely deployed. To provide a systematic way to incentivise cloudlet deployment, I implemented OpenStack++ that extends OpenStack, an open source ecosystem for cloud computing. With OpenStack++, any individual or any vendor who uses OpenStack for their cloud computing can easily use the cloudlet structures. For this work, I designed and implemented OpenStack++ APIs and ported the cloudlet open source project to OpenStack. In addition, I provide a web interface and a client program for OpenStack users.

References

[1] How to assign devices with VT-d in KVM. http://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM.

[2] Libvirt: Domain XML format. https://libvirt.org/formatdomain.html.

[3] Neutron - OpenStack. https://wiki.openstack.org/wiki/Neutron.

[4] OpenStack Wikipedia. https://en.wikipedia.org/wiki/OpenStack.

[5] T. Agus, C. Suied, S. Thorpe, and D. Pressnitzer. Characteristics of human voice processing. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, Paris, France, June 2010.

[6] Apple. iPhone 4S - Ask Siri to help you get things done. http://www.apple.com/iphone/features/siri.html.

[7] S. R. Ellis, K. Mania, B. D. Adelstein, and M. I. Hill. Generalizeability of Latency Detection in a Variety of Virtual Environments. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 48, 2004.

[8] Google. Glass Glass. https://www.google.com/glass/start/, 2014.

[9] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards Wearable Cognitive Assistance. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys 2014)*, 2013.

[10] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 153–166. ACM, 2013.

[11] InfoWord. HP blade server targets cloud computing. http://www.infoworld.com/article/2651934/computer-hardware/hp-blade-server-targets-cloud-computing.html.

[12] OpenStack. OpenStack - Open Source Cloud Computing Software. http://www.openstack.org/.

[13] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), October-December 2009.

[14] P. SIG. PCI Hot-Plug Specification Revision 1.1. http://www.drydkim.com/MyDocuments/PCI%20Spec/specifications/pcihp1_1.pdf, 2001.