GENERALIZED RESOURCE MANAGEMENT FOR HETEROGENEOUS CLOUD DATA CENTERS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AHMET EROL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2019

Approval of the thesis:

GENERALIZED RESOURCE MANAGEMENT FOR HETEROGENEOUS CLOUD DATA CENTERS

submitted by **AHMET EROL** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _______________

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Engineering** _______________

Prof. Dr. Şenan Ece Güran Schmidt
Supervisor, **Electrical and Electronics Engineering** _______________

Examining Committee Members:

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU _______________

Prof. Dr. Şenan Ece Güran Schmidt
Electrical and Electronics Engineering Dept., METU _______________

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering Dept., METU _______________

Prof. Dr. Özcan Öztürk
Computer Engineering Dept., Bilkent University _______________

Assist. Prof. Dr. Pelin Angın
Computer Engineering Dept., METU _______________

Date:

Name, Surname:	Ahmet Erol
Signature :	

ABSTRACT

GENERALIZED RESOURCE MANAGEMENT FOR HETEROGENEOUS CLOUD DATA CENTERS

Erol, Ahmet
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Şenan Ece Güran Schmidt

September 2019, 66 pages

OpenStack is a widely used management tool for cloud computing which is designed to work on servers and allocate standard computing resources such as CPU, memory or disk. The current trend for integrating different hardware accelerators such as FPGAs and GPUs in the cloud requires managing these heterogeneous resources. In this thesis, we propose a generalization for OpenStack Nova project which extends the relevant data structures to include these new resources. More importantly, we present a new lightweight Nova Compute module that we call Nova-G Compute. Nova-G Compute is suitable to work with different hardware platforms and can communicate with the rest of the OpenStack Projects. We implement a hypervisor-like software to enable Nova-G Compute accessing the FPGA resources. We perform experimental evaluation of Nova-G Compute using the known and used OpenStack benchmarking tool Rally. Our results show that Nova-G Compute works as desired without any reduced performance compared to standard Nova.

Keywords: Cloud computing, virtualization, OpenStack, FPGA, Nova, Rally

ÖZ

HETEROJEN BULUT VERİ MERKEZLERİ İÇİN GENELLEŞTİRİLMİŞ KAYNAK YÖNETİMİ

Erol, Ahmet
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Şenan Ece Güran Schmidt

Eylül 2019 , 66 sayfa

OpenStack bulut hesaplama için yaygınca kullanılan bir yönetim aracıdır. CPU, bellek ve disk gibi standart hesaplama kaynaklarının yönetimi için kullanılmaktadır. Bulut bilişimde FPGA ve GPU benzeri donanım hızlandırıcıların kullanılması trend olmuştur. Bu şekildeki heterojen kaynakların yönetilmesine ihtiyaç duyulmaktadır. Bu çalışmada, OpenStack Nova projesinin bu yeni kaynak tiplerini de kapsayacak şekilde genelleştirilmesini sunmaktayız. Daha da önemlisi, yeni az kaynak tüketen bir Nova Compute modülü geliştirdik. Nova-G Compute ismini verdiğimiz modül çeşitli donanımlar üzerinde çalışabilmekte ve diğer OpenStack projeleri ile sorunsuz şekilde haberleşebilmektedir. Hypervisor benzeri bir yazılım ile Nova-G Compute modülünün FPGA kaynaklarına erişimine olanak sağladık. Openstack test aracı olan Rally'yi kullanarak deneysel testler gerçekleştirdik. Yapılan testlerin sonucu Nova-G Compute beklendiği gibi çalıştığını ve orijinal Nova'ya göre herhangi bir performans düşümü olmadığını göstermiştir.

Anahtar Kelimeler: Bulut Hesaplama, Sanallaştırma, OpenStack, FPGA, Nova, Rally

To My Family

ACKNOWLEDGMENTS

I would like to express my great appreciations to my supervisor Prof. Dr. Şenan Ece Güran Schmidt for her support and guidance throughout this thesis.

The author would like to thank ASELSAN Inc. for their support. This thesis was supported by the Scientific and Research Council of Turkey (TUBITAK) [Project Code 117E667-117E668].

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	
VM	Virtual Machine
API	Application Programming Interface
NIC	Network Interface Card
RPC	Remote Procedure Call
NAT	Network Address Translation

CHAPTER 1

INTRODUCTION

Cloud computing has become a popular computing model in the recent years as it exploits the economies of scale for efficient use of computing resources. The data centers of today are mostly cloud based with virtualized servers to provide on-demand scalability and flexibility of the available resources such as CPU, memory, data storage and network bandwidth. A cloud data center provider may offer Infrastructure as a Service (IaaS), where the user gets a virtual machine (VM) with processing, memory, storage and networking resources, which can be installed with any desired operating system and software. Differently, Platform as a Service (PaaS) commonly provides a ready environment with operating system, programming language execution environment, database and web server for developers to test and deploy their programs and applications. Finally in Software as a Service (SaaS), the user only accesses the provided application for example via a web browser without any control of the underlying infrastructure.

OpenStack is an open source software that is preferred by many large cloud providers [3] to assign physical resources to users in the form of Virtual Machines (VM) in cloud computing systems. OpenStack is composed of a number of projects with different functionalities such as authentication, network management and image services. The actual managing of resources by means of virtualization is carried out by the OpenStack Nova project. Nova has two components called Nova Conductor and Nova Compute. A controller node in the managed cloud runs Nova Conductor and each server that is provisioned for VMs runs Nova Compute.

The slowdown of Moore's law and the increased data and problem sizes together with the development of high performance programmable hardware platforms such as

FPGAs increase the popularity of hardware accelerators. Hardware accelerators can provide better performance and less energy consumption depending on the problem properties and size [4]. On one hand, these different hardware platforms may not be compatible with the operating system and hypervisor software used on standard cloud servers. Furthermore, their processing capabilities may be more limited and they might not be able to run OpenStack in a scalable and high performance manner. On the other hand, integrating hardware resources in the cloud based data center should be seamless, together with virtualization and dynamic resource allocation capabilities.

OpenStack is designed to work in cloud data centers with conventional servers. To this end, the current Nova implementation is limited to traditional computing resources such as CPU, memory and disk. Furthermore, Nova is only compatible with certain operating systems and hypervisor software.

This focus of this thesis is extension of OpenStack Nova project to support cloud data centers with heterogeneous resources.

The first contribution of this thesis a new lightweight project that we call Nova-G Compute which is designed to replace the standard Nova Compute for such heterogeneous hardware platforms. Nova-G Compute can work with the standard OpenStack projects by sending and receiving messages in the correct format. The implementation of Nova-G is in Python language and is independent of the operating system. In this way, different hardware platforms can be supported. The second contribution is extending the data structures of Nova Controller with the generalized resource types to work with Nova-G Compute. Different than the previous work, new resource types are defined at the same level with conventional server resources which enables using the standard Nova Schedulers to allocate the available resources to the VM requests. Nova-G Compute includes a generalized hypervisor driver to access the available resources on the respective hardware similar to standard Nova Compute. To this end, we implement an emulation for the FPGAvisor software that functions similar to a hypervisor for FPGA as an example for allocation of non-standard hardware resources.

We demonstrate the functionality and performance of Nova-G Compute with a number of experiments including tests with Rally Tool which is the OpenStack framework for performance analysis and benchmarking. Our results show that Nova-G Compute can work seamlessly with OpenStack and can boot VMs as desired without any performance decrease compared to Nova Compute.

The remainder of this thesis is organized as follows. In Chapter 2 we introduce OpenStack components and the relevant issues in heterogeneous computing. Chapter 3 presents our workflow in installing and modifying OpenStack and the Rally test environment. Chapter 4 presents the main contribution of the thesis which is the generalized OpenStack Nova Project that we call Nova-G. Chapter 5 presents the evaluation methodology and the results for Nova-G. Chapter 6 states our conclusions and future work.

CHAPTER 2

OPENSTACK AND HETEROGENEOUS CLOUD COMPUTING

OpenStack is an open source cloud platform that controls large groups of resources that can be categorized as compute, network and storage. It gives administrative control over cloud hardware by provisioning virtual machines (VM) [5]. OpenStack software has flexible architecture that enables customization of platform according to business needs. OpenStack is a very popular cloud resource management tool with yearly increasing revenue in the market [6].

OpenStack is a set of open source software projects for managing and orchestrating various cloud resources. Each OpenStack project has specific duty for achieving completely managed cloud structure. In this chapter, general structure and operation of OpenStack software is discussed by giving details of each OpenStack project.

2.1 OpenStack Components

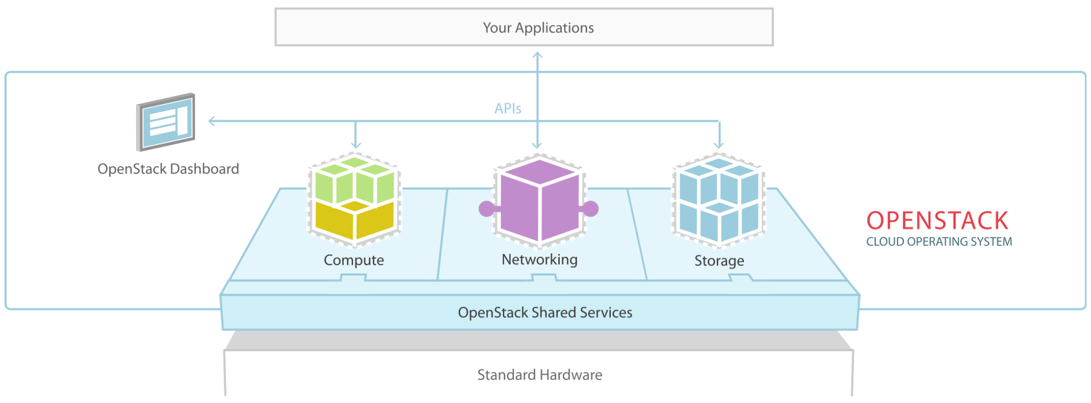


Figure 2.1: OpenStack Components Groups [1]

OpenStack components can be mainly grouped into four different categories namely compute, networking, storage and shared services as shown in Figure 2.1. All these services run on a *controller node*. The VMs created and managed by OpenStack run on the *compute nodes*.

Nova is OpenStack compute service that provides access to compute resources by means of virtual machines. To this end, each compute node runs a Nova component that enables OpenStack to control the node. **Neutron** is focused on networking-as-services (NaaS) in cloud environment. To provide storage services, **Swift** (Object Storage) and **Cinder** (Block Storage) are deployed in OpenStack clouds. Some OpenStack services are shared with different OpenStack projects. The shared services enable other services communicate with each other. **Keystone** is shared service that provides necessary API client for authentication of users and applications. VM images are managed and stored by OpenStack service named as **Glance**. Each OpenStack project has its own database which consists of many tables with the information for each project.

There are more OpenStack projects available but only required ones to have fully functional OpenStack cloud are discussed here.

2.1.1 Keystone

Keystone project consists of multiple services that are used together for proper Keystone operation. Keystone runs on the controller node of the data center. Identity service provides user/project credential validation with user name and passwords. When valid user name and password are supplied with authentication request, Token service replies this request with newly created valid *token*. Tokens are used by other users or applications to use APIs of OpenStack projects [7].

Just like all other OpenStack projects Keystone provides very well documented and rich API. Some provided API operations can be summarized as follows [8].

- Authentication and token management
- Creation and management of domains

- Group management
- Project and Projects Tags
- Service and Endpoint Managements

To be able to use an OpenStack cloud, users or applications should authenticate themselves through Keystone API. User should provide a valid user name and password with scope of authentication. Users can request access to project, domain or a system. Identity service reply user request with a token if user name and password is valid for requested scope. An HTTP request with POST method should be made to controller node API address. Keystone services use port number 5000 to serve its rest API functions in default settings. Example URL for making post request are shown in Listing 2.1. *controller* is host name of controller node. It can be replaced with IP addresses of controller node. */v3/auth/tokens* is the required URL for authentication process.

Code Listing 2.1: Example API request URL of authentication process

```
http://controller:5000/v3/auth/tokens
```

HTTP POST request should be made for URL given in Listing 2.1 with request body containing information in Listing 2.2. Message body is formatted in JSON language for all OpenStack API requests. After validation of user name and password Keystone will reply the request with valid token which is in header of reply message. Reply message body also has lots of information like issue time and expiration time about token. In the reply header *x-subject-token* variable holds the created token ID. This token ID can be used for future API requests of all OpenStack projects as longs as it is valid.

Code Listing 2.2: POST request body for project scoped authentication

```
{
    "auth": {
        "identity": {
            "methods": [
                "password"
            ],
            "password": {
                "user": {
                    "name": "admin",
                    "domain": {
                        "name": "Default"
                    },
                    "password": "ADMIN_PASS"
                }
            }
        },
        "scope": {
            "project": {
                "id": "561d55f847114a05b7fb74a933ba26e9"
            }
        }
    }
}
```

Code Listing 2.3: Reply Message Header containing Token ID

```
date: Sat, 06 Jul 2019 10:43:32 GMT
server: Apache/2.4.18 (Ubuntu)
x-subject-token: gAAAAABdIHtZ_FJECqOD_ORdxzDdm1eag48eW8NytUNiZ3YG4nlh02vB-
    avlSIQeNslQKq_wMw9Mz56l0t5MpxpELUFD0liSPMuzEUyWbzGX_oTW-6
    jZ9BgXrZcU8Wq9J9IZRCqt48Uy4AklV2AJ0nzyKpkxd1JMjPxj5JLGwCg9zKG7t9VxVgo
vary: X-Auth-Token
x-distribution: Ubuntu
x-openstack-request-id: req-06ea9ad7-ae6b-48c8-b729-e5fb06289212
content-length: 3290
content-type: application/json
```

2.1.2 Nova

Nova is the OpenStack project that enables provisioning of compute nodes by providing virtual compute instances on compute nodes. It uses various virtualization techniques to create a pool of compute resources to be used in compute instances. Nova supports different types of hypervisors for virtualization [9].

Nova project consists of different components some of which run on the controller

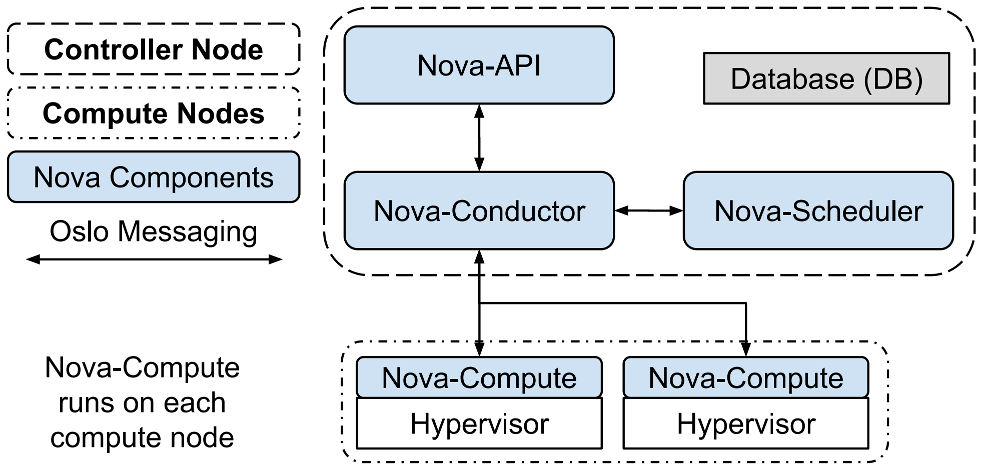


Figure 2.2: OpenStack Nova Block Diagram

node while others run on the compute nodes. Nova components and their communications with each other are shown in Figure 2.2.

Nova-API is the Nova component which enables the users or applications to access and control Nova services. Using Nova API, one can control the whole OpenStack cloud by creating or deleting compute instances, managing hypervisors and controlling the compute instance images. Nova-API runs on controller node. Users or third party applications do not have direct access to compute nodes. All the information about compute nodes should be accessed from controller node through Nova-API.

Database (DB) stores all the information about resources and their allocations which are defined by API models. Nova project stores the information about compute nodes and their resources in the database table called `compute_nodes`. The cloud services user requests a VM that is configured according to one of the pre-defined *flavors*. The available flavors are stored in the Nova-API database in `flavors` table.

Nova-API is responsible for delivering third party applications' requests to the **Nova-Conductor** which is the core component of the Nova project. Nova Conductor runs on the controller node and has full control over other Nova components and database. Other Nova components do not have direct connection to the database. They use Nova-Conductor services to access database [10]. Moreover, Nova-Conductor will use Oslo Messaging services whose details will be given in OpenStack operation section to communicate with other modules.

Nova-Scheduler is used to determine how to dispatch compute request. When a compute instance is requested, one compute node should be chosen to run requested compute instance. Nova-Conductor will request Nova-Scheduler to choose best compute host according to compute instance specification by using compute filters. Nova-Scheduler collects all information about available compute hosts in cloud. It will filter available hosts by request specifications and cloud owner defined rules. Compute hosts that are able to pass filtering step are evaluated by weighting them with predefined rules [11]. Compute node with the highest score will be chosen and result will be returned to Nova-Conductor.

All compute nodes are managed by **Nova-Compute** module. Nova-Compute at chosen node will be informed by Nova-Conductor. Nova-Compute is responsible for building disk image, resource allocation, instantiating compute instance and terminating it via underlying hypervisor driver. It also needs to update compute node and compute instance status by periodically reporting information. Nova-Compute will instantiate requested compute instance and returned its information to Nova-Conductor.

The focus of this thesis is the Nova project. Hence, we provide further details about Nova in Chapter 4.

2.1.3 Glance

Glance project provides image service where users can upload and manage compute instance images that are meant to be used by Nova when creating compute instances. Glance also provides API that enables creation, deleting and management of images. Moreover, using API third party applications can download full image data when they are authorized by Keystone [12]. Nova requires glance to operate because it uses glance to store compute node instance images.

2.1.4 Neutron

Network connectivity of compute instances is provided by Neutron project. Neutron maintains network connectivity service between virtual NIC of compute instances and it managed by Nova project. Neutron provides API to control all network connectivity. Using provided Neutron API, third party applications are able to create and manage virtual networks, create/delete ports and manage L3 services [13]. Nova uses also Neutron API to manage network.

2.1.5 Horizon

Horizon is OpenStack project that provides web based user interface for OpenStack services. OpenStack projects including Nova, Glance, Neutron etc. can be managed via Horizon with a web browser. It can be used to monitor OpenStack cloud and create new compute instances.

2.2 OpenStack Operation

In this section, OpenStack operation is discussed in detail. Complete flow of creation of a compute instance from beginning to end is explained. To be understood by one who is new to OpenStack, some OpenStack terms are explained before OpenStack operation.

Compute Instance: Virtual machine created by Nova-Compute via hypervisor on compute node.

Service: For each OpenStack component a service is created to manage software. Services of OpenStack can be stopped or restarted by command line interface. There is a table in the database that holds information about service. Each compute node has different service that manages the Nova-Compute.

Flavor: In OpenStack, a flavor defines compute, memory, and storage properties of VM instances. Users can create many flavors with different properties. Each compute instance is instantiated by predefined flavors. Compute instance properties defined in flavors effect compute nodes selection process.

Oslo Messaging: It is a python library that supports RPC and notifications over a different messaging protocols. OpenStack uses Oslo messaging library for all communication between projects and modules. RabbitMQ driver used in default OpenStack deployment and verification.

Rabbit Message Queue: OpenStack uses RabbitMQ message broker system at default settings. It provides messaging system between applications. Simply, it consists of producer which means applications that try to send data and consumer that is application waiting for new messages. RabbitMQ provides *queue* between producer and consumer. Producers put messages into queue while consumers reads messages from queue. If sender application waits for response after sending messages, that is technically RPC, consumer responds the producer message by sending another message for reply queue. Producer takes responses from reply queue. RabbitMQ has two different modules. One of them is the server, where all messages are processed and stored. Producers and consumers are the clients of the server.

OpenStack is a cloud operating system that manages the resources of the cloud by fulfilling user requests in the best way. In this section, OpenStack operation flow will be discussed by giving an example of requesting a compute instance and fulfilling this request. Complete operation is summarized in Figure 2.3 and the details of each step will be discussed.

(1) User or 3rd party applications request a compute instance with predefined properties in flavor by using API or Horizon web interface. This application or user should get authentication token from Keystone beforehand to be used in its API request. If request is made by using Horizon tool, then Horizon makes the necessary API Request.

(2) After receiving API request, Nova-API should authenticate the user with provided token. Not only the user but also the scope of authentication is checked by Keystone. User may be valid but user may not have access to create compute instance in given project. If the request is valid, Keystone authenticates the request and gives response to Nova-API.

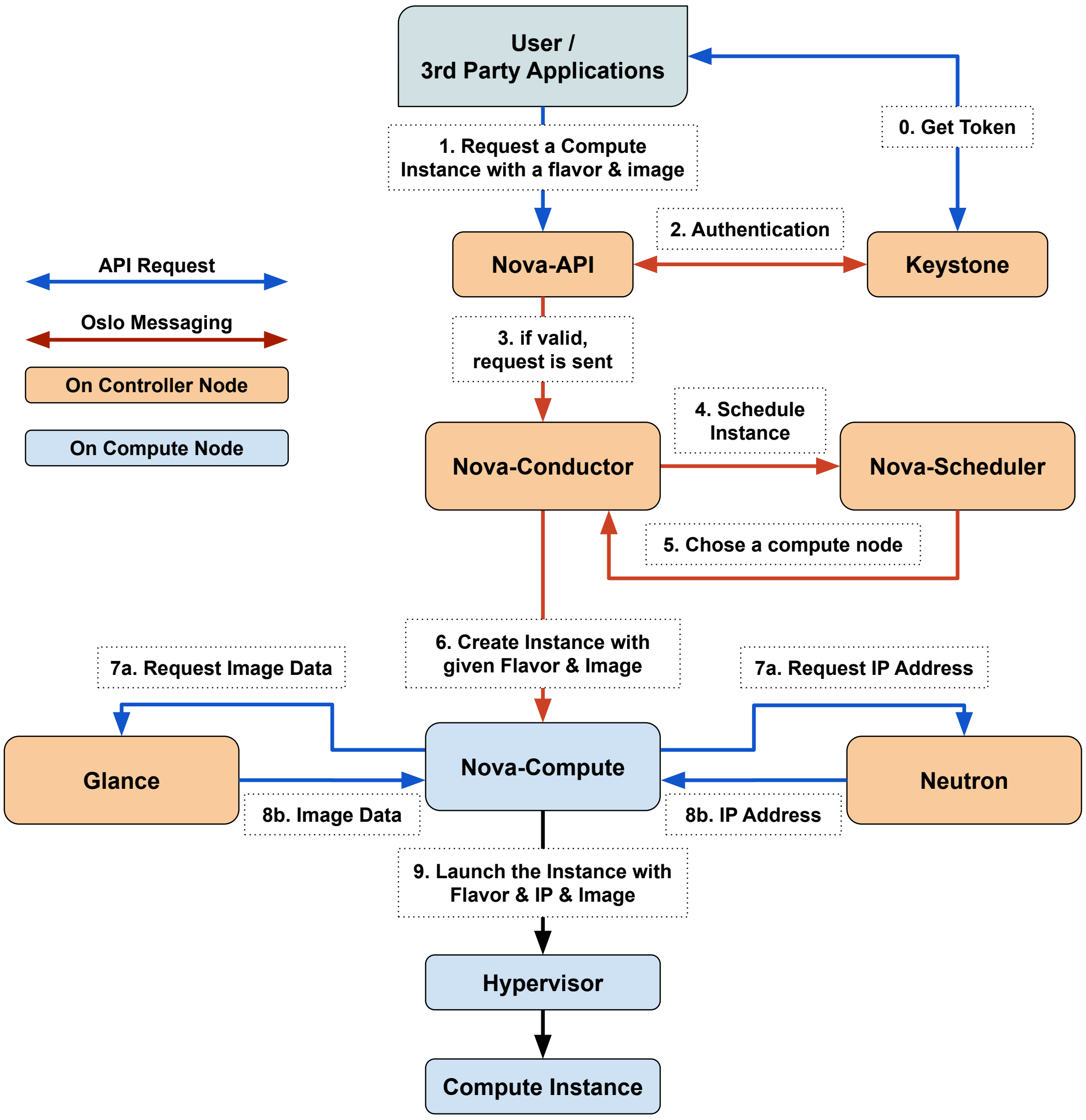


Figure 2.3: OpenStack Operation Flow Chart

(3) Nova-API makes sure that given request is valid. Afterwards, it sends the request with Oslo Messaging using RabbitMQ. Each OpenStack component has a queue named with component name in RabbitMQ. The message is sent to message queue of Nova-Conductor.

(4) Nova-Conductor receives the message from its message queue and decodes the message. After successfully decoding message, a decision should be made for finding which compute node should host new compute instance. Nova-Conductor makes an RPC request to Nova-Scheduler with its message queue for decision to choose a compute node.

(5) Nova-Scheduler collects all information like CPU, RAM and Disc usage about compute nodes. It has a job to select the best candidate of compute nodes depending on the flavor of compute instance. It filters compute nodes that are capable of hosting the compute instance with given flavor. Selected compute nodes are evaluated based on predefined weighting rules and the best candidate is selected. Selected compute host name is returned to Nova-Conductor with related RPC request.

(6) After receiving response to RPC call that is asking for the best compute node to host new compute instance, Nova-Conductor is responsible for delivering compute instance request to related compute node. At this point, it is guaranteed that selected compute node is capable of hosting compute instance with given flavor and image. A message that contains all required information about the new compute instance is sent to selected compute node message queue for processing.

(7) Selected compute node's Nova-Compute software receives the message of creation of new compute instance. Nova-Compute needs to collect all the data that is required to instantiate compute instance. Nova-Compute asks Neutron using its API to make the necessary arrangement in the network and respond with new compute instance IP address. At the same time, it requests the compute instance image data from Glance. New compute instance will be created using this image data.

(8) Nova-Compute waits for the image data and IP address from the related OpenStack projects. It allocates hardware resources required to host given compute instance. Nova-Compute uses Neutron and Glance API to communicate with them.

Once all the required information is collected, Nova-Compute is ready to boot compute instance.

(9) Hypervisors at compute nodes are used to create compute instances. Nova-Compute uses various hypervisors through its hypervisor drivers. It passes all the information to the hypervisor so that it can create a new compute instance. Hypervisor creates a virtual machine with the given flavor. It boots the instance with the given image and initializes all its network interface with the given IP address. When a compute instance is created, hypervisor returns status information to Nova-Compute. When compute instance successfully created, user or 3rd party application is informed, and they can start using the compute instance.

2.3 Heterogeneous Cloud Computing and OpenStack Support

Employing different hardware platforms in the cloud is a popular topic both academically and commercially. To this end, cloud computing systems are integrated with hardware accelerators realized on FPGA platforms [14], GPU (Graphics Processing Unit)[15] , TPU (Tensor Processing Unit)[16] and IoT Hardware[17]. [18] is a pioneering work that proposes a cloud-based data center architecture accelerated with reconfigurable FPGA for use in Microsoft data centers. In this study, an Altera Stratix V-based hardware accelerator card was added to each compute node which can accelerate an application running on the machine on which it is connected or can be used as a network appliance without putting a load on the CPU. Regarding the management of the FPGA resources [19] proposes assigning FPGA reconfigurable regions in the scope of IaaS/HaaS (Hardware as a Service) by using OpenStack. However, it does not describe how FPGA resources are defined and assigned using Nova. In the follow-up work [20], virtualization of hardware accelerators is discussed without giving details on what should be changed in OpenStack and how it should be implemented. Here it is important to note that FPGA SoC platforms come with a processor [21] which enables the cloud service provider to employ stand alone FPGA cards without a server for implementing energy efficient accelerators.

The sub-field `extra_spec` of Nova data structure is used to integrate GPUs into

OpenStack platform in the paper [22]. As this field is a sub-field of existing standard resource types, it cannot be used by the standard Nova Schedulers. In [23], network is accelerated by transferring OpenStack network services to switch hardware and allowing Nova to access them. In [24], a new component named IoTronic is added to OpenStack as Sensing as a Service for IoT systems. OpenStack has a recent project called Cyborg [25] which is proposed as a service for managing accelerators such as FPGAs, GPUs etc. Cyborg is designed to work with Nova Compute as an agent and cannot work stand alone. Hence, it does not support heterogeneous cloud hardware that is not connected to a CPU via PCI. Moreover, effective scheduling of accelerators are not possible because it uses `extra_spec` field to define the different types of resources similar to [22].

CHAPTER 3

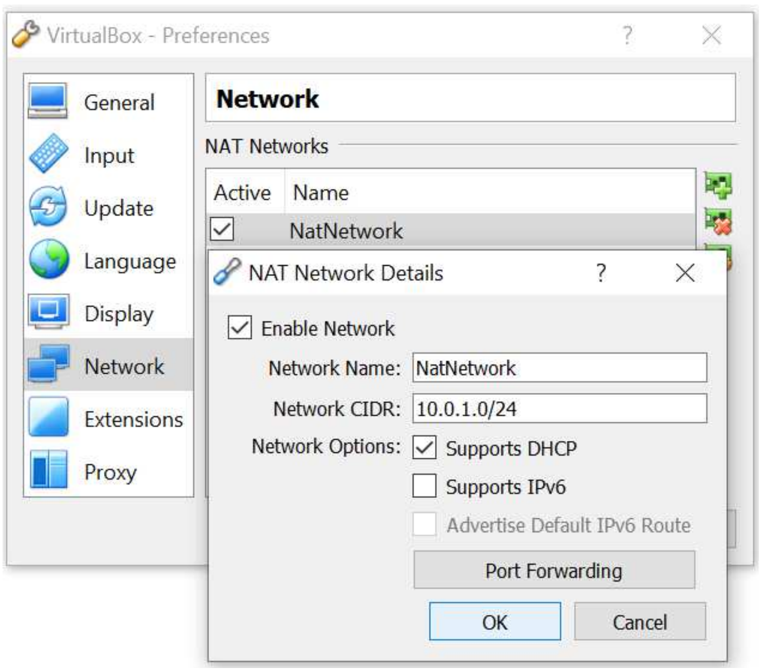
OPENSTACK INSTALLATION, MODIFICATION AND TEST ENVIRONMENT

This chapter presents the detailed workflow for installing and modification of OpenStack and the relevant software components. Furthermore, we present the installation of the Rally test tool and the procedures that we follow to run the tests.

3.1 Installing OpenStack on Single Physical Machine

In this section, OpenStack installation on single physical machine is explained. OpenStack components are installed on virtual machines. Three different virtual machines are created to simulate a controller node and two compute nodes. OpenStack can work without any problem with one controller node and one compute node. On controller node, Keystone, Nova, Glance, Neutron and Horizon OpenStack projects are installed. Only Nova-Compute is installed for compute nodes. The physical machine has Intel Core i7-7500U CPU at 2.7 GHz and 16 GB of RAM. It has Windows 10 operating system. For virtualization product, Oracle VM VirtualBox version 5.2.8 is used.

Firstly, VirtualBox software is downloaded and installed on operating system [26]. For communication between controller node and compute nodes virtual NAT network is created as shown in the Figure 3.1. The virtual machines that are simulating controller and compute nodes are added to the created virtual NAT network. Reader of this document should have information about general overview of OpenStack installation process. OpenStack installation overview and components are explained in [27].


VirtualBox - Preferences

General, Input, Update, Language, Display, Network, Extensions, Proxy

Network

NAT Networks

Active	Name
✓	NatNetwork

NAT Network Details

✓ Enable Network

Network Name: NatNetwork

Network CIDR: 10.0.1.0/24

Network Options: ✓ Supports DHCP

☐ Supports IPv6

☐ Advertise Default IPv6 Route

Port Forwarding

OK Cancel

Figure 3.1: VirtualBox NAT Network Settings

In test setup, OpenStack host networking layout is used. In this layout, controller node and compute nodes have two different network interfaces. For management network, created virtual NAT network is used. VirtualBox bridged network is used to simulate “Provider Network” in Figure 3.2. Extra network interface is added for controller node for debugging purposes.

Using VirtualBox tool, new VM is created to host Ubuntu 16.04 operating system. For controller node, VM should have at least 4 GB RAM. The network adapters of controller node are configured as shown in the Figure 3.3.

Two VMs are created for compute nodes. Compute nodes should have at least 2 GB RAM. Network configuration of compute nodes are given in Figure 3.4.

Ubuntu 16.04 operating system should be installed on all of created VMs. After Ubuntu installation is finished, network interface of each node is configured. For controller node, management interface IP address and name resolution file (/etc/hosts) is configured as described in [28]. For debugging purposes, physical windows machine hosts file ("c:/Windows/System32/Drivers/etc/hosts") is modified. IP address of second bridged network is added to the hosts files with name "controller". Similar

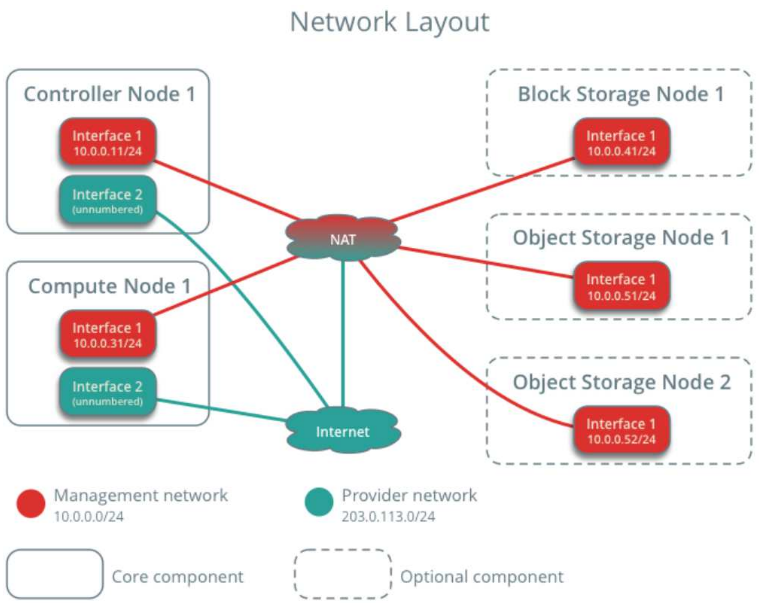


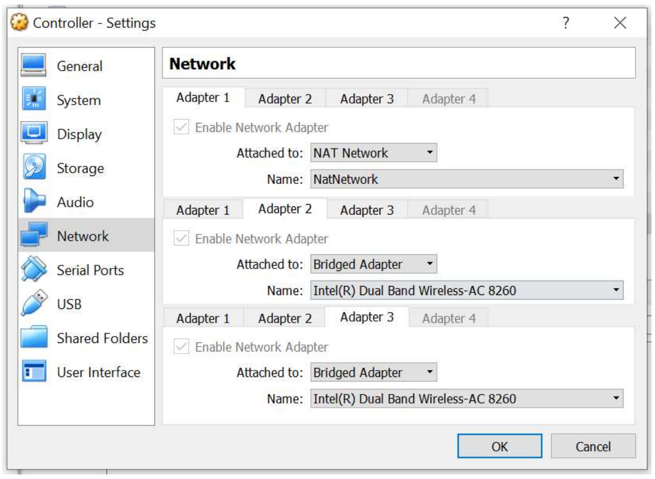
Figure 3.2: OpenStack Network Layout [2]

configuration is done for compute nodes as well. Compute nodes modifications are given in [29].

Connectivity of controller and compute nodes is checked before continuing the installation of OpenStack. Verification of connectivity is described in [30]. At this point, test setup is verified and OpenStack components can be installed on VMs.

OpenStack version Pike is used in this setup. OpenStack Pike for Ubuntu 16.04 LTS is installed for all nodes. OpenStack package installation steps are explained in [31]. Using these steps, OpenStack package is installed on both controller node and compute nodes. Following applications are installed on Controller Node. Configuration and installation steps of each software are given in OpenStack documents. Following applications are installed in given order.

- SQL Database : [32]
- RabbitMQ : [33]
- Memcached : [34]
- ETCD : [35]


Controller - Settings

General
System
Display
Storage
Audio
Network
Serial Ports
USB
Shared Folders
User Interface

Network

Adapter 1 | Adapter 2 | Adapter 3 | Adapter 4

✓ Enable Network Adapter
Attached to: NAT Network
Name: NatNetwork

Adapter 1 | Adapter 2 | Adapter 3 | Adapter 4

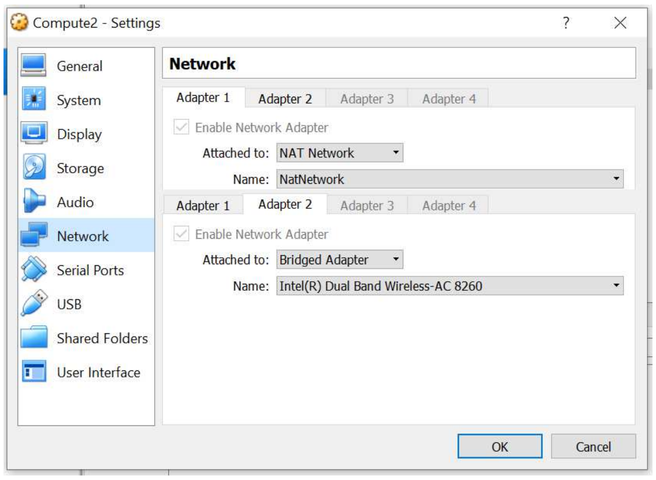
✓ Enable Network Adapter
Attached to: Bridged Adapter
Name: Intel(R) Dual Band Wireless-AC 8260

Adapter 1 | Adapter 2 | Adapter 3 | Adapter 4

✓ Enable Network Adapter
Attached to: Bridged Adapter
Name: Intel(R) Dual Band Wireless-AC 8260

OK | Cancel

Figure 3.3: Controller Node Network Configuration


Compute2 - Settings

General
System
Display
Storage
Audio
Network
Serial Ports
USB
Shared Folders
User Interface

Network

Adapter 1 | Adapter 2 | Adapter 3 | Adapter 4

✓ Enable Network Adapter
Attached to: NAT Network
Name: NatNetwork

Adapter 1 | Adapter 2 | Adapter 3 | Adapter 4

✓ Enable Network Adapter
Attached to: Bridged Adapter
Name: Intel(R) Dual Band Wireless-AC 8260

OK | Cancel

Figure 3.4: Compute Node Network Configuration

When all of these programs are installed, actual OpenStack projects should be installed. OpenStack projects given below are installed in given order using OpenStack documents web pages. They are installed on controller node.

- Keystone is installed using [36]. Given steps are followed and keystone installation is verified like in the "Verify operation" section of OpenStack Docs.
- Glance installation steps are given in [37]. Glance operation is verified using steps in "Verify operation" section of OpenStack Docs.
- Nova installation and configuration are done as discussed in [38]. Nova service

operation is verified after nova is installed on compute nodes.

- Horizon dashboard is installed using [39]. It should be configured to accept all hosts for easy debugging. Horizon operation can be verified by visiting web page `http://controller/horizon`.

  When accessing the Horizon web page, it may give error message of "Gateway Timeout error". To overcome this problem, "etc/apache2/conf-available/openstack-dashboard.conf" file is opened and following line is added.

  ```
  WSGIApplicationGroup %{GLOBAL}
  ```

  Horizon service is restarted using following command. Then, it should start working properly.

  ```
  service apache2 reload
  ```

- Neutron installation steps are given in [40]. Neutron installation provides options to choose networking architecture. Our setup is configured to use "Networking Option 1: Provider networks". Therefore, installation steps of option 1 given in [41] should be followed.

All required OpenStack projects are installed on Controller node. Some projects should also be installed on compute nodes. Nova installation on compute node is given in [42]. After installing, complete Nova operation can be verified with the steps given in "Verify operation" section of OpenStack Docs.

When all installations are finished, one can create a compute instance using Horizon dashboard and can test operation of OpenStack on test setup.

3.2 Making OpenStack Modification

Original OpenStack software is modified to support generalized resources. Moreover, OpenStack database tables are also modified. In this section, how these changes are made is discussed. For database modifications, phpMyAdmin software is used. phpMyAdmin is included in Ubuntu repository. It can be installed on controller node using command below.

```
sudo apt-get update
sudo apt-get install phpmyadmin php-mbstring php-gettext
```

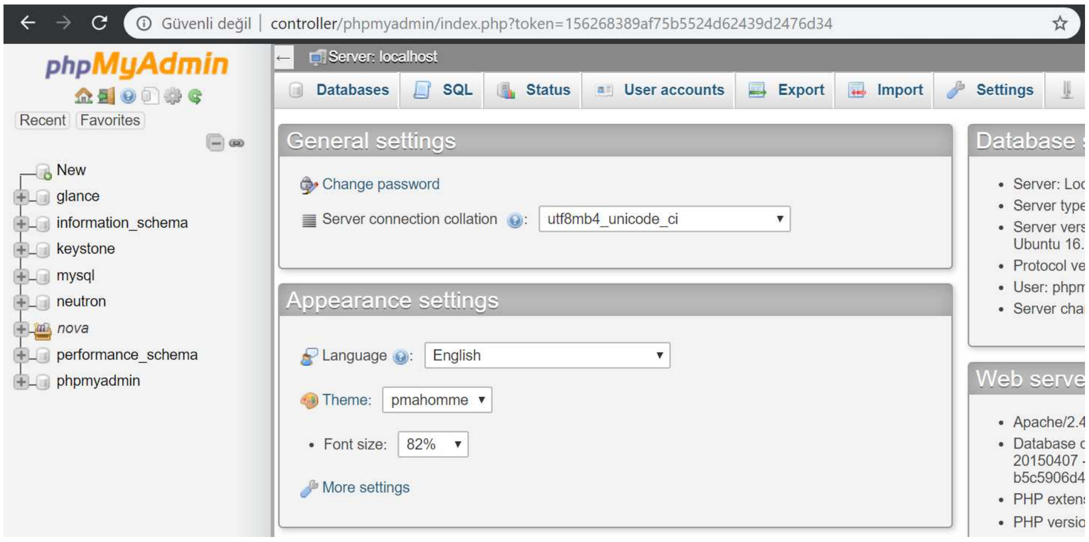
Installer asks few questions to complete installation. Choose apache by pressing space button and continue the installations. When it prompts to whether to use dbconfig-common, yes is selected. After executing following commands, phpMyAdmin applications is ready to use.

```
sudo phpenmod mcrypt
sudo phpenmod mbstring
sudo systemctl restart apache2
```

phpMyAdmin application can be reached using URL below.

`http://controller/phpmyadmin/`

After logging with admin user name and password, the screen shown in Figure 3.5. At the left bar, all created databases for OpenStack are listed. Each database contains multiple tables for storing data. The changes discussed in Section 4.1, are made using phpMyAdmin tool web interface.



New
glance
information_schema
keystone
mysql
neutron
nova
performance_schema
phpmyadmin

Server connection collation: utf8mb4_unicode_ci

Language: English

Theme: pmahomme

Font size: 82%

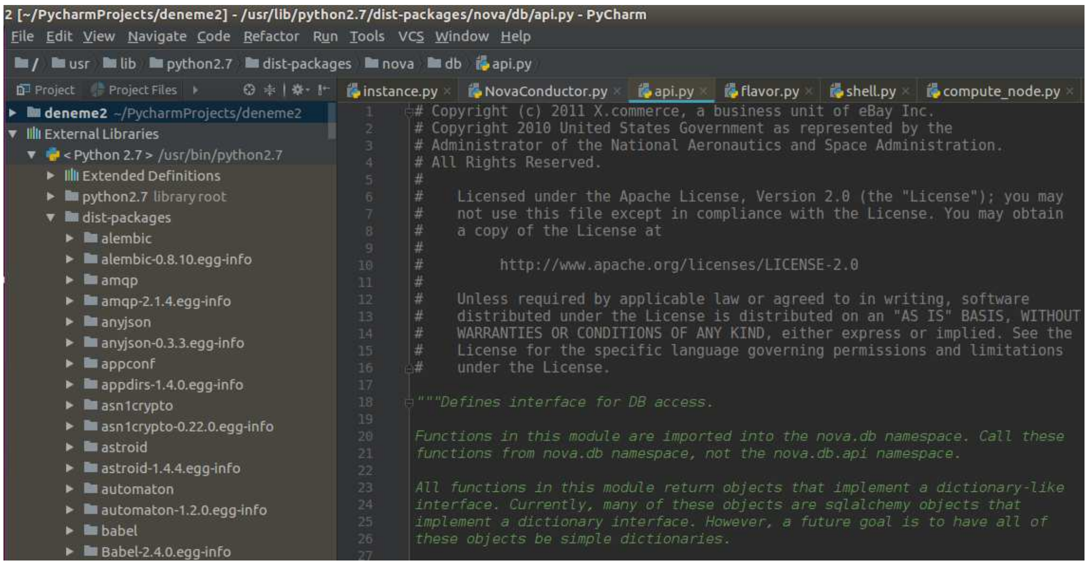
- Server: Loc
- Server type
- Server vers Ubuntu 16.
- Protocol ve
- User: phpm
- Server cha

- Apache/2.4
- Database c 20150407 - b5c5906d4
- PHP exten
- PHP versio

Figure 3.5: phpMyAdmin Default Screen

To modify OpenStack source code, Python IDE PyCharm is used on controller node. PyCharm can be downloaded from [43]. PyCharm should be started with sudo privileges to have access to modify all files. It can be started using command below on controller node.

```
sudo ./pycharm-community-2017.3.3/bin/pycharm.sh
```


2 [~/PycharmProjects/deneme2] - /usr/lib/python2.7/dist-packages/nova/db/api.py - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
/ usr lib python2.7 dist-packages nova db api.py
Project Project Files
instance.py NovaConductor.py api.py flavor.py shell.py compute_node.py

```
deneme2 ~/PycharmProjects/deneme2
External Libraries
  < Python 2.7 > /usr/bin/python2.7
    Extended Definitions
    python2.7 library root
    dist-packages
      alembic
      alembic-0.8.10.egg-info
      amqp
      amqp-2.1.4.egg-info
      anyjson
      anyjson-0.3.3.egg-info
      appconf
      appdirs-1.4.0.egg-info
      asn1crypto
      asn1crypto-0.22.0.egg-info
      astroid
      astroid-1.4.4.egg-info
      automaton
      automaton-1.2.0.egg-info
      babel
      Babel-2.4.0.egg-info
```

```
# Copyright (c) 2011 X.commerce, a business unit of eBay Inc.
# Copyright 2010 United States Government as represented by the
# Administrator of the National Aeronautics and Space Administration.
# All Rights Reserved.
#
#    Licensed under the Apache License, Version 2.0 (the "License"); you may
#    not use this file except in compliance with the License. You may obtain
#    a copy of the License at
#
#         http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing, software
#    distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
#    WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
#    License for the specific language governing permissions and limitations
#    under the License.

"""Defines interface for DB access.

Functions in this module are imported into the nova.db namespace. Call these
functions from nova.db namespace, not the nova.db.api namespace.

All functions in this module return objects that implement a dictionary-like
interface. Currently, many of these objects are sqlalchemy objects that
implement a dictionary interface. However, a future goal is to have all of
these objects be simple dictionaries.
```

Figure 3.6: PyCharm IDE Default Screen

PyCharm lists all installed python libraries including OpenStack projects under "dist-packages" folder shown in Figure 3.6.

Any file from OpenStack projects can be accessed from navigation bar. In this section, we listed modified files by giving file location address and explained why these changes are made. Flavor class is defined in the file flavor.py. Modified file is shown in Listing 3.1.

`/usr/lib/python2.7/dist-packages/nova/objects/flavor.py`

Code Listing 3.1: Flavor class after modifications

```
class Flavor(base.NovaPersistentObject, base.NovaObject,
    VERSION = '1.1'

    fields = {
        'id': fields.IntegerField(),
        'name': fields.StringField(nullable=True),
        'memory_mb': fields.IntegerField(),
        'vcpus': fields.IntegerField(),
        'resource_g1': fields.IntegerField(),
        'resource_g2': fields.IntegerField(),
        'root_gb': fields.IntegerField(),
        'ephemeral_gb': fields.IntegerField(),
        'flavorid': fields.StringField(),
        'swap': fields.IntegerField(),
        'rxtx_factor': fields.FloatField(nullable=True, default=1.0),
        'vcpu_weight': fields.IntegerField(nullable=True),
        'disabled': fields.BooleanField(),
        'is_public': fields.BooleanField(),
        'extra_specs': fields.DictOfStringsField(),
        'projects': fields.ListOfStringsField(),
        }
```

Flavor sqlalchemy api model is also modified and details are given in Listing 4.2. File location is given below.

`/usr/lib/python2.7/dist-packages/nova/db/sqlalchemy/api_models.py`

Flavor related functions are also modified so that creating new flavors is possible. Some part of modified files are shown in Listing 3.2. Moreover, the files where are this function is called are also modified accordingly.

`/usr/lib/python2.7/dist-packages/nova/compute/flavors.py`

Code Listing 3.2: Flavor create function after modifications

```
def create(name, memory, vcpus, resource_g1, resource_g2, root_gb,
    ephemeral_gb=0, flavorid=None,
           swap=0, rxtx_factor=1.0, is_public=True):
    kwargs = {
        'memory_mb': memory,
        'vcpus': vcpus,
        'resource_g1': resource_g1,
        'resource_g2': resource_g2,
        'root_gb': root_gb,
        'ephemeral_gb': ephemeral_gb,
        'swap': swap,
        'rxtx_factor': rxtx_factor,
    }

    flavor_attributes = {
        'memory_mb': ('ram', 1),
        'vcpus': ('vcpus', 1),
        'resource_g1': ('resource_g1', 1),
        'resource_g2': ('resource_g2', 1),
        'root_gb': ('disk', 0),
        'ephemeral_gb': ('ephemeral', 0),
        'swap': ('swap', 0)
    }
```

/usr/lib/python2.7/dist-packages/nova/api/openstack/compute/flavor_manage.py

Code Listing 3.3: Flavor create function after modifications

```
def _create(self, req, body):
    vals = body['flavor']

    name = vals['name']
    flavorid = vals.get('id')
    memory = vals['ram']
    vcpus = vals['vcpus']
    resource_g1 = vals['resource_g1']
    resource_g2 = vals['resource_g2']
    root_gb = vals['disk']
    ephemeral_gb = vals.get('OS-FLV-EXT-DATA:ephemeral', 0)
    swap = vals.get('swap', 0)
    rxtx_factor = vals.get('rxtx_factor', 1.0)
    is_public = vals.get('os-flavor-access:is_public', True)

    try:
        flavor = flavors.create(name, memory, vcpus, resource_g1,
            resource_g2, root_gb,
            ephemeral_gb=ephemeral_gb,
            flavorid=flavorid, swap=swap,
            rxtx_factor=rxtx_factor,
            is_public=is_public)
```

/usr/lib/python2.7/dist-packages/nova/api/openstack/

compute/schemas/flavor_manage.py

Code Listing 3.4: Flavor create function after modifications

```
create = {
    'type': 'object',
    'properties': {
        'flavor': {
            'type': 'object',
            'properties': {
                'name': parameter_types.name,
                'id': {
                    'type': ['string', 'number', 'null'],
                    'minLength': 1, 'maxLength': 255,
                    'pattern': '^(?! )[a-zA-Z0-9. _-]+(?<! )$'
                },
                'ram': parameter_types.flavor_param_positive,
                'resource_g1': parameter_types.flavor_param_positive,
                'resource_g2': parameter_types.flavor_param_positive,
                'fpga': parameter_types.flavor_param_positive,
                'disk': parameter_types.flavor_param_non_negative,
                'OS-FLV-EXT-DATA:ephemeral':
                    parameter_types.flavor_param_non_negative,
                'swap': parameter_types.flavor_param_non_negative,
                # positive ( > 0) float
                'rxtx_factor': {
                    'type': ['number', 'string'],
                    'pattern': '^[0-9]+(\.[0-9]+)?$',
                    'minimum': 0, 'exclusiveMinimum': True,
                    'maximum': db.SQL_SP_FLOAT_MAX
                },
                'os-flavor-access:is_public': parameter_types.boolean,
            },
            'required': ['name', 'ram', 'vcpus', 'disk'],
            'additionalProperties': False,
        },
    },
    'required': ['flavor'],
    'additionalProperties': False,
}
```

/usr/lib/python2.7/dist-packages/nova/api/openstack/compute/views/flavors.py

Code Listing 3.5: Flavor show function after modifications

```
def show(self, request, flavor):
    flavor_dict = {
        "flavor": {
            "id": flavor["flavorid"],
            "name": flavor["name"],
            "ram": flavor["memory_mb"],
            "disk": flavor["root_gb"],
            "swap": flavor["swap"] or "",
            "OS-FLV-EXT-DATA:ephemeral": flavor["ephemeral_gb"],
            "OS-FLV-DISABLED:disabled": flavor["disabled"],
            "vcpus": flavor["vcpus"],
            "resource_g1": flavor["resource_g1"],
            "resource_g2": flavor["resource_g2"],
            "links": self._get_links(request,
                                     flavor["flavorid"],
                                     self._collection_name),
        },
    }
```

3.3 Installing Rally and Running Tests

Rally is an OpenStack bench-marking tool. In this section, Rally installation on Ubuntu is explained. Firstly, new virtual machine is created using VirtualBox with Ubuntu operating system. Rally software is downloaded and installed with below commands.

```
wget -q -O- https://raw.githubusercontent.com/openstack/rally/master/
    install_rally.sh
sudo ./install_rally.sh
pip install rally-openstack
```

Rally software uses API to test OpenStack. Therefore, we need to provide Rally with existing OpenStack deployment information. For easy debug, hosts file should be modified with controller node IP address. Rally configuration is provided with JSON file given below. JSON file is named as existing.json

```
{
    "openstack": {
        "auth_url": "http://controller:5000/v3/",
        "region_name": "RegionOne",
        "endpoint_type": "public",
        "admin": {
            "username": "admin",
            "password": "ADMIN_PASS",
            "tenant_name": "admin"
        },
        "https_insecure": false,
        "https_cacert": ""
    }
}
```

Rally deployment is created command below with given settings.

```
rally deployment create --file=existing.json --name=existing
```

After Rally is successfully configured, it can be started with existing sample tests with command below.

```
rally task start samples/tasks/scenarios/nova/boot-and-delete.json
```

Rally is able to create HTML report of executed tasks. HTML report is created using command below when Rally finishes its operation.

```
rally task report --out=report1.html --open
```

Example test scenario and content of boot-and-delete.json file is given below.

```
{
    "NovaServers.boot_and_delete_server": [
        {
            "args": {
                "flavor": {
                    "name": "mini"
                },
                "image": {
                    "name": "cirros"
                },
                "force_delete": false
            },
            "runner": {
                "type": "constant",
                "times": 100,
                "concurrency": 9
            },
            "context": {
                "users": {
                    "tenants": 3,
                    "users_per_tenant": 2
                }
            }
        }
    ]
}
```

CHAPTER 4

GENERALIZATION OF OPENSTACK NOVA

OpenStack is developed to manage a large group of resources in the cloud. However, current OpenStack implementation includes management of conventional compute resources such as CPU, RAM and disk. Moreover, compute nodes are restricted to standard server hardware and few operating systems. We extend OpenStack resource data structures to support generalized resource types [44]. Furthermore, we develop a new lightweight Nova-Compute that we name Nova-G(eneralized) Compute to enable the management of other resource types. Nova-G Compute does not have any hardware restrictions therefore it can run on any OS and hardware platform. Nova-G compute provides compatible messaging interfaces and seamlessly integrates with the rest of the OpenStack components.

4.1 Nova Data Structure Extensions and Modifications to Support Generalized Resources

We introduce Nova Database in Chapter 2. Most of the OpenStack services use SQL database and different types of SQL databases are supported such as MariaDB, MySQL and PortgeSQL. In this thesis work, MySQL is used as OpenStack database application. All the modification details are given in Section .

4.1.1 Nova `compute_nodes` Database Table Extensions

Nova project stores the information about compute nodes in the database table called `compute_nodes`. Compute nodes' id, available resources, status information about compute nodes are stored in this table.The content of original database table shown in Table B.1 in Appendix B. In this database table, the main resources of compute nodes which are virtual cpu, memory and disk are explicitly stored. Nova stores the total capacity of each resource. Moreover, it stores the amount of used resources in a different field of the table. The main resource types are very limited because of the data storage structure of Nova. The resource types of compute nodes that do not exist in the table data structure by default must be given in `extra_resources` field. This field is a sub-field of existing standard resource types and it cannot be used by the standard Nova Schedulers. That restricts the independent usage of other resources from main resources.

Table 4.1: Extended Compute Node Resource Database Structure

Field	**Type**	**Field**	**Type**
`id`	`Integer`	`service_id`	`Integer`
`vcpus`	`Integer`	`vcpus_used`	`Integer`
`memory_mb`	`Integer`	`memory_mb_used`	`Integer`
`local_gb`	`Integer`	`local_gb_used`	`Integer`
`hypervisor_type`	`Text`	`hypervisor_version`	`Text`
`cpu_info`	`Text`		
`resource_g1`	`Integer`	`resource_g1_used`	`Integer`
`resource_g2`	`Integer`	`resource_g2_used`	`Integer`

To this end, we extend the database table by adding new resource types and their usage values maintaining the original data structure. Table 4.1 shows a partial example with the current resource types and two new added fields. Here, `resource_g1` can represent `fpga` resources whereas `resource_g2` can represent `gpus`. The usage of these resources in the related compute node is represented as an Integer type. These database modifications are made using phpmyadmin tool. New fields are added to database without interrupting operation of OpenStack. More resource types can be added to the database in the same manner. Since original data structure is preserved, each new resource type can be chosen independently in a flavor to create new compute instance configurations. Generalized resources do not have to be on the same compute node with vcpus or memory. With the help of this flexibility, one can create a compute instance without any cpu or memory that is only including generalized resources. Nova-G Compute module will instantiate `resource_g` type resources defined in flavor. For example if the compute node is a standalone FPGA accelerator card, a VM without vCPU and with a `fpga` can be instantiated.

4.1.2 Nova-API `flavors` Database Table Extensions

The original structure of the `flavors` table of Nova-API database is shown in Table B.2. To this end, we update `flavors` table of Nova-API database to include the new resource types in the offered flavors. Some part of the modified flavor table structure is shown in Table 4.2 with the new fields of `resource_g1` and `resource_g2` fields. Each field in the flavor table can be specified independently by user.

Table 4.2: Extended Nova_API flavors Database Structure

Field	Type	Field	Type
id	Integer	name	text
vcpus	Integer	memory_mb	Integer
swap	Integer	root_gb	Integer
resource_g1	Integer	resource_g2	Integer

4.1.3 Nova SQLAlchemy Database Model Modifications

In OpenStack, software access to database tables are defined by SQLAlchemy model files. For each table, a model is defined so that the software trying to access database is able to use the database structure. Since we modify the table structure in the database, we also modify SQLAlchemy model classes of each table and add the new generalized resource types. Modified `ComputeNode` classes are shown in Listing 4.1.

Code Listing 4.1: Part of extended Compute Nodes sqlalchemy model after modifications

```
class ComputeNode(BASE, NovaBase, models.SoftDeleteMixin):
    __tablename__ = 'compute_nodes'
    id = Column(Integer, primary_key=True)
    service_id = Column(Integer, nullable=True)
    host = Column(String(255), nullable=True)
    uuid = Column(String(36), nullable=True)
    vcpus = Column(Integer, nullable=False)
    resource_g1 = Column(Integer, nullable=False)
    resource_g2 = Column(Integer, nullable=False)
    memory_mb = Column(Integer, nullable=False)
    local_gb = Column(Integer, nullable=False)
    vcpus_used = Column(Integer, nullable=False)
    resource_g1_used = Column(Integer, nullable=False)
    resource_g2_used = Column(Integer, nullable=False)
    memory_mb_used = Column(Integer, nullable=False)
    local_gb_used = Column(Integer, nullable=False)
    hypervisor_type = Column(MediumText(), nullable=False)
    hypervisor_version = Column(Integer, nullable=False)
    hypervisor_hostname = Column(String(255))
```

Similar modifications are done for `Flavors` classes model shown in the Listing 4.2.

Code Listing 4.2: Part of extended Flavors sqlalchemy model after modifications

```
class Flavors(API_BASE):
    __tablename__ = 'flavors'
    id = Column(Integer, primary_key=True)
    name = Column(String(255), nullable=False)
    memory_mb = Column(Integer, nullable=False)
    vcpus = Column(Integer, nullable=False)
    resource_g1 = Column(Integer, nullable=False)
    resource_g2 = Column(Integer, nullable=False)
    root_gb = Column(Integer)
    ephemeral_gb = Column(Integer)
    flavorid = Column(String(255), nullable=False)
    swap = Column(Integer, nullable=False, default=0)
    rxtx_factor = Column(Float, default=1)
    vcpu_weight = Column(Integer)
    disabled = Column(Boolean, default=False)
    is_public = Column(Boolean, default=True)
```

4.1.4 Nova Scheduler Operation with Resource Extensions

Nova scheduler uses these extended flavors to choose suitable compute nodes for the new compute instances that are requested. Therefore, adding `resource_g` fields at the same level as the conventional resources such as `vcpus` makes these resources usable by the scheduling algorithms. All of existing filtering and weighting algorithms of Nova Scheduler can be extended to include `resource_g` or new filtering algorithms can be added to leverage efficiency of scheduling algorithms.

4.2 Nova-G Compute

OpenStack provides virtualized compute resources by Nova-Compute which works on compute nodes with the help of hypervisors. Original Nova-Compute software is developed to work on cloud servers and has many complex sub-systems. Therefore, it cannot be scaled down to run physical machine that has limited processing power. Nova-G Compute can be used to manage FPGA with processors like ZYNQ. The CPU on the FPGA has very limited processing power. Thus, we develop Nova-G Compute so that it can run on such a machine. Moreover, Nova-Compute has lots of software dependencies which restrict operating system. Nova-G Compute is designed to replace Nova-Compute and work on standalone FPGA SoCs with CPU or any other customized hardware accelerators in the cloud such as the architecture proposed in [45]. To this end, we develop a lightweight Nova-G Compute that seamlessly works with other OpenStack projects by correctly generating messages as well as correctly parsing the received messages and taking the necessary actions. Nova-G Compute does not have any external software dependencies therefore it is meant to work on any operating system. Nova-G Compute is developed from scratch in Python 2.7 language in a similar structure to Nova-Compute as seen in the block diagram that is shown in the Figure 4.1. To this end, all functions that we describe in this Chapter are developed in the scope of this thesis work.

OpenStack operation requires two main communication channels that are Oslo Messaging and HTTP API requests. We develop Nova-G such that it is capable of using these two different channels by creating Rabbit-MQ Controller and API Controller. We also develop a general hypervisor driver to support different types of generalized resources.

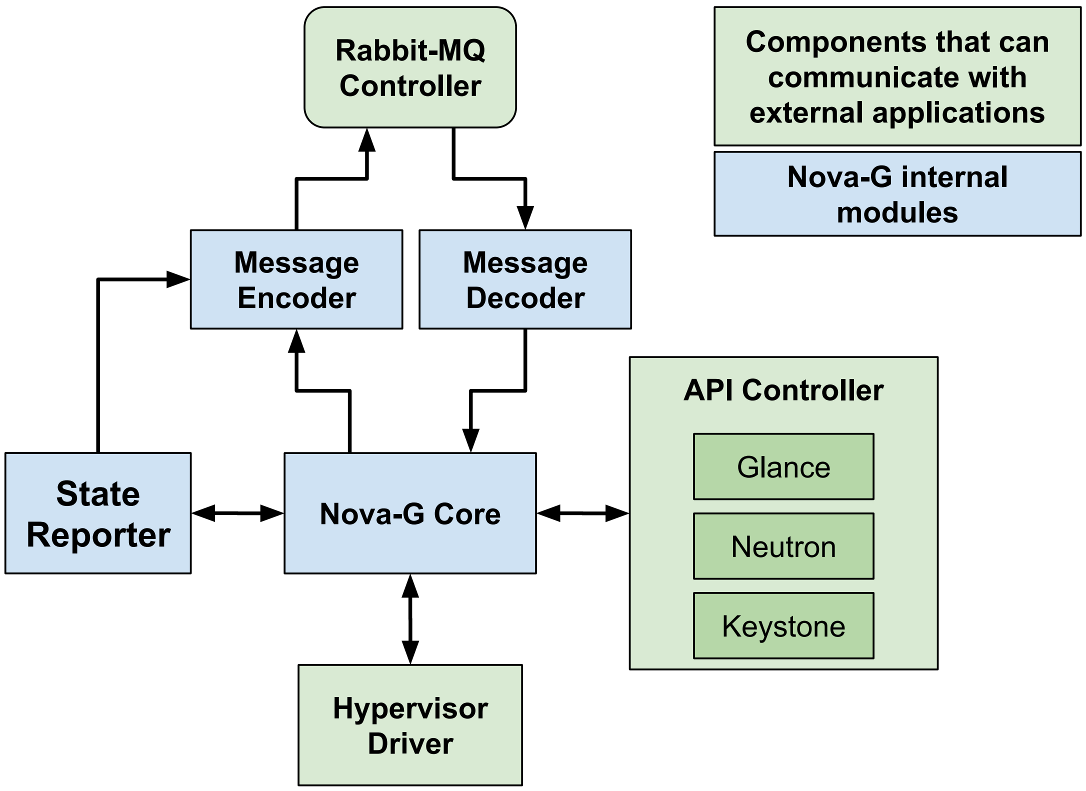


Figure 4.1: Nova-G Compute Block Diagram

4.2.1 OpenStack API Controller

OpenStack projects provide APIs for other applications to use it. We develop an API Controller to communicate with Keystone, Neutron and Glance. Other Nova-G components use API controller to perform their tasks. OpenStack API controller has three main functions namely `get_token`, `get_image_file` and `create_-port`.

`get_token` function uses Keystone API to authenticate Nova-G Compute module. Prior to starting any operation Nova-G Compute requests a token using this function. The acquired token is used in all future API requests in Nova-G Compute.

`get_image_file` retrieves compute instance image that is chosen by the user using Glance API. Retrieved compute instance image MD5 checksum is calculated to ensure that correct data is received. It stores image data to be used by hypervisor driver when it boots compute instance. `create_port` method is used to adjust network connectivity of compute instance with help of Neutron API. It returns an IP and

MAC address to be used in the compute instance. The hypervisor boots the compute instance with these parameters.

4.2.2 RabbitMQ Controller

Nova components communicate with each other using Oslo Messaging provided by RabbitMQ. Compute nodes directly communicate with Nova-Conductor in the controller node as shown in Figure 2.2. Similar to the standard Nova-Compute, Nova-G Compute should have also direct communication with Nova-Conductor to continue its operation. Nova-G Compute listens to its message queue continuously to serve Nova-Conductor requests. Moreover, it is capable of sending messages and RPC requests to Nova-Conductor and receiving the results of RPC requests. All of these functions are provided by the four basic methods of developed RabbitMQ Controller. These methods are explained below.

`start_consume_queue` method is used to start listening the message queue of the related compute node. It continuously monitors the message queue. When a message is received, it calls the `consume_callback` method to respond to this message. `consume_callback` function checks the integrity of the message. `consume_callback` uses Message Decoder block to understand the received message. Depending on message contents, it starts required actions by using Nova-G Core.

Nova-G Compute sends message and makes RPC requests to Nova-Conductor by using `send_message_nova_conductor` method. It sends encoded messages in string format to the Nova-Conductor message queue. Messages can include RPC requests. Nova-Conductor sends the responses of RPC requests to the reply queue. When the reply message is received, `result_callback` method is called by passing the reply message content. Depending on the message content required actions are taken.

4.2.3 Message Decoder

Message Decoder module decodes messages received by RabbitMQ Controller module. Message Decoder's functions are called when a message received. Messages from RabbitMQ are in JSON string format. Messages are converted into Python dictionaries by using `convert_str_to_dict` method. After converting into dictionary, message type is determined by `determine_message_type`. Depending on message type different decoder functions are called to get the required information from message. Some messages can be discarded because of unknown message types. Two main message decoder functions are explained below.

`decode_build_and_run_instance` method is called when Nova-Conductor sends a message to Nova-G Compute for creating a new compute instance. This function returns `instance_uuid` of the new compute instance, `network_id` of Neutron network which new instance will be connected to and `image_id` of the compute instance image.

`decode_terminate_instance` is used when terminating compute instances. Nova-Conductor sends a message to Nova-G Compute for terminating specific compute instance. This function decodes the messages and returns `instance_uuid` and `instance_id` of the compute instance that is to be terminated.

4.2.4 Message Encoder

Message Encoder module is used when sending message to Nova-Conductor. These messages are compatible with the messages that are sent by the standard Nova Compute. This module creates string messages that are ready to be sent by RabbitMQ. The messages are created in the Oslo Messaging format. Depending on the message type to be sent different encoder methods are used. After the message is encoded, it is converted to a string by method `convert_dict_to_str`. The string messages are encapsulated in Oslo messaging format using `serialize_msg` function. This function's output is directly sent to RabbitMQ. We develop different encoding functions for different operations. Each function is explained in below.

`encode_destroy_instance` is used to create a message when terminating compute instance. When Nova-G Compute is freshly started, it receives its service information by sending message. The message is created by `encode_get_service_-by_uuid`. Nova-G Compute update status of compute instance continuously by help of `encode_update_instance_status`. State Reporter updates status of compute node using `encode_update_report_count`. Network details of instance is updated using the message created by `encode_update_instance_info_-cache`.

4.2.5 Nova-G Core

All actions of Nova-G Compute are managed by Core module. This module uses other modules methods to perform its tasks. Nova-G Core has three main functions that are; creating compute instances, terminating compute instances and reporting the current state of the compute node. Each Nova-Compute software on compute nodes are named a *service* in OpenStack. Information about the compute nodes is stored in the database by these defined services. Nova-G Core creates a service entity for Nova-G compute module by retrieving data from the database. Then created service entity is given to State Reporter Module for updating status. `get_service_by_uuid` method is used to get information from database and create a service entity.

When Nova-G Compute module is requested to create compute instance, `build_-and_run_instance` method of Nova-G Core is called. Nova-G Core decodes the message by using Message Decoder related function and gets `instance_uuid`, `network_id` and `image_id` of requested compute instance. Furthermore, it gets IP and MAC addresses for the compute instance with given `network_id` by using API Controller. It gets the boot image data of the compute instance with the given `image_id`. Once all required information is collected, Core module calls hypervisor driver's `build_and_run_instance` method to create the instance. After the hypervisor completes its operation, it returns its response about compute instance creation. If compute instance is successfully created, Nova-G Core sends a message to Nova-Conductor for updating status of compute instance.

Nova-Conductor may also request to terminate the created compute instances. Nova-

G Core `terminate_instance` method is responsible for terminating instances. Nova-Conductor sends a message to terminate an instance. Firstly, the message decoded using Message Decoder related function. Then, the hypervisor is requested to terminate the instance. When hypervisor successfully terminates the instance, Nova-G Core releases all the resources of the terminated instance and updates its status by sending a different message that is created by Message Encoder to Nova-Conductor.

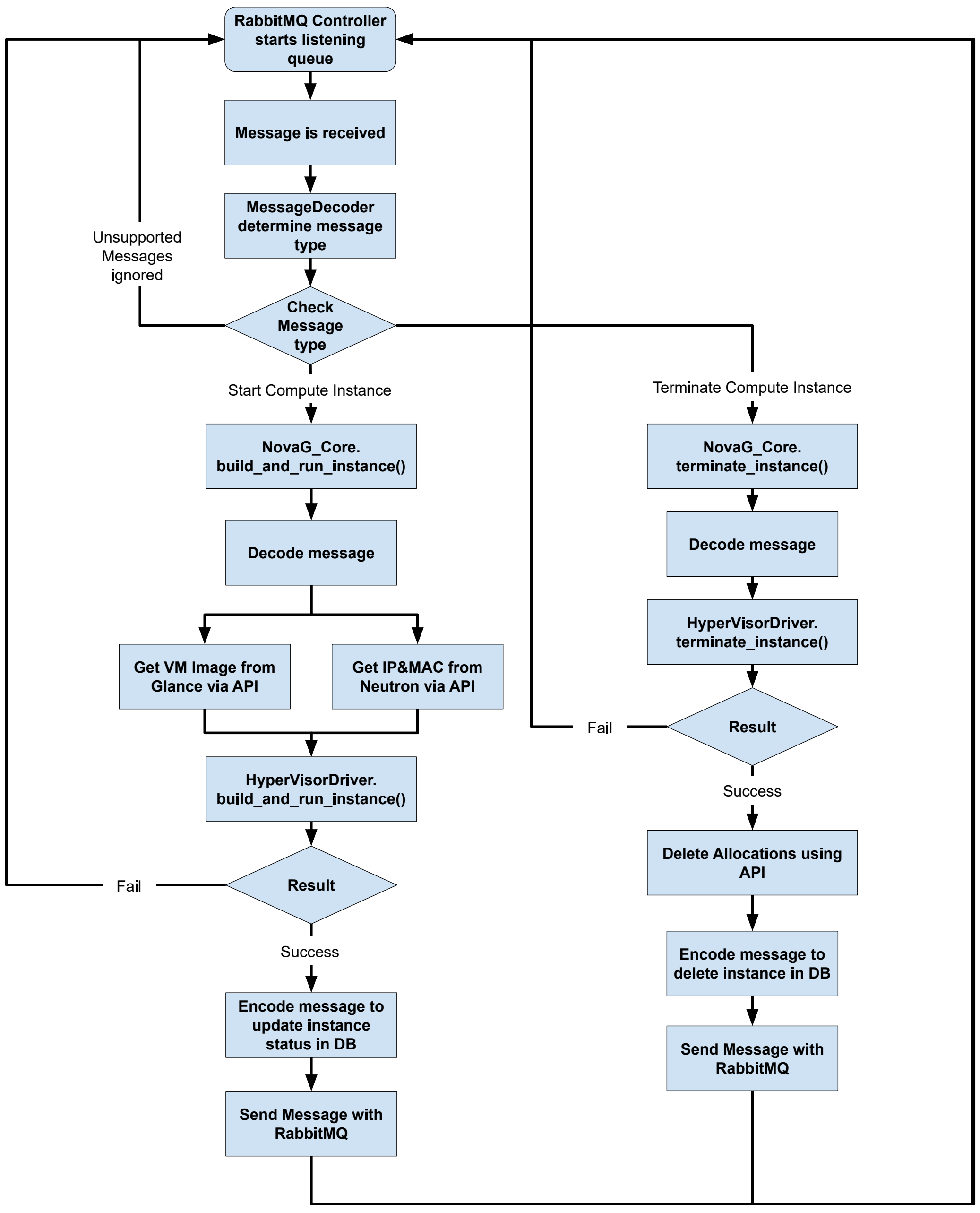
4.2.6 Hypervisor Driver

Compute Instances are created by hypervisors using virtualized resources of compute nodes in OpenStack. We design Nova-G so that it can keep same architecture with the original Nova. Nova-G is designed to support different resources than the conventional computing resources where each new resource is expected to have its own hypervisor. To this end, Hypervisor Driver is developed by defining general methods of driver to support such different hypervisors. Two main functions are defined for the hypervisor driver. `build_and_run_instance` is used to create compute instances while `terminate_instance` method is used for terminating instances. Nova-G core uses these functions regardless of resource type. The hypervisor driver takes the necessary actions depending on the resource type. It can use hypervisor utilities to create compute instances.

4.2.7 State Reporter

Each compute node has its own service entity in the database. OpenStack creates services for each Nova-Compute in the cloud system. Similarly, Nova-G Compute has service entities managed by Nova-G Core. State Reporter module uses service the entity to update the status of service in database. Other OpenStack components use this status information to know that compute nodes are alive and working properly. For example, Nova-Scheduler chooses the compute nodes depending on their status information. If the service of given node is not alive, that compute node is filtered out and not selected for hosting a compute instance. State Reporter functions are periodically called to update status information using RabbitMQ Controller.

4.3 Nova-G Compute Operation



RabbitMQ Controller starts listening queue
Message is received
MessageDecoder determine message type
Unsupported Messages ignored
Check Message type
Start Compute Instance
Terminate Compute Instance
NovaG_Core. build_and_run_instance()
Decode message
Get VM Image from Glance via API
Get IP&MAC from Neutron via API
HyperVisorDriver. build_and_run_instance()
Fail
Result
Success
Encode message to update instance status in DB
Send Message with RabbitMQ
NovaG_Core. terminate_instance()
Decode message
HyperVisorDriver. terminate_instance()
Fail
Result
Success
Delete Allocations using API
Encode message to delete instance in DB
Send Message with RabbitMQ

Figure 4.2: Nova-G Compute Flow Chart

In this section, complete operation flow of Nova-G Compute is discussed. Firstly, Nova-G Compute software starts its all modules. If all modules are successfully initialized, it gets a token to use in future API Calls. Then RabbitMQ Controller starts listening message queue. Operation flow of Nova-G Compute from this point onward is shown in the Figure 4.2.

Nova-Conductor sends a message to Nova-G Compute module. After message reception, Message Decoder determines message type. If message type cannot be determined, messages are discarded. Determined valid message types can be starting compute instance and terminating existing compute instance. If Nova-Conductor requests to start a new compute instance, message is decoded with respective method. Network id and boot image id of compute instance are extracted from message by using Message Decoder. API Controller asks Neutron to give IP and MAC addresses from given network. At the same time, boot image of compute instance is requested from Neutron. Once IP and MAC addresses are given and image data is received, Nova-G Core passes all collected information to Hypervisor Driver. Hypervisor Driver instantiates compute instance by using Hypervisor utilities. Result of creating compute instance returned to Nova-G Core. If it is success, Nova-G Core updates the instance status in database by sending a message to Nova-Conductor.

Nova-G Compute is requested to terminate instance by sending terminate instance message. Message is decoded by related Message Decoder function to get instance info. Nova-G Core calls terminate instance method of Hypervisor Driver. Hypervisor Driver terminates the running instance by help of Hypervisor. After successful termination of the compute instance, Nova-G Core deletes all allocated resources for the instance. The compute instance is removed from database by sending a message to Nova-Conductor.

4.4 Capabilities of Nova-G Compute

Nova-G Compute takes advantage of the current OpenStack structure and supports heterogeneous hardware resources. Nova-G Compute provides most of the capabilities of Nova Compute without needs for changing general OpenStack implementation. It works flawlessly with other OpenStack projects. Capabilities of the Nova-G Compute are summarized below.

- Compute node status in database is updated periodically by State Reporter module.
- Different types of resources require different Hypervisors. Nova-G Compute

provides Hypervisor Driver to support different resource types.

- Nova-G is capable of using Oslo Messaging. It can make and respond to RPC requests in the same way OpenStack does.
- Compute instances are successfully created with given boot images.
- Neutron provides full control over network of compute instances. Nova-G Compute takes advantages of Neutron to have full control and management of IP and MAC addresses.
- Status of compute instances are reported continuously even in booting phase.
- Nova-G Compute software does not have any software dependencies rather than RabbitMQ.
- Nova-Compute is not required by Nova-G Compute when using non-standard resources.
- Nova-G has lightweight implementation that results in low RAM usage.

CHAPTER 5

EVALUATION AND RESULTS

The developed Nova-G Compute together with the extensions to support different resource types is tested on a test setup. The test setup includes OpenStack projects such as Horizon, Nova, Neutron and Glance. All these installed projects are fully functional. After verification of OpenStack projects functionality, Nova-G Compute compatibility is tested with OpenStack projects. We test performance of Nova-G Compute and compare scalability of Nova-G Compute with existing Nova-Compute.

5.1 Test Setup

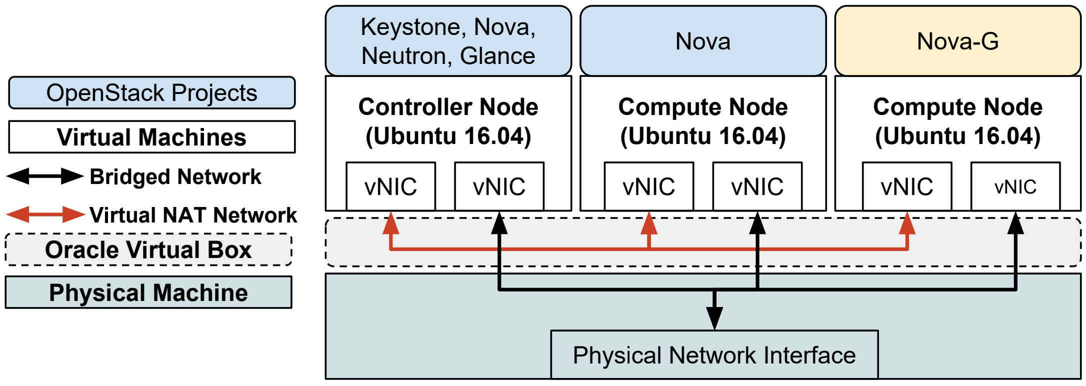

OpenStack Projects
Virtual Machines
Bridged Network
Virtual NAT Network
Oracle Virtual Box
Physical Machine
Keystone, Nova, Neutron, Glance
Nova
Nova-G
Controller Node (Ubuntu 16.04)
Compute Node (Ubuntu 16.04)
Compute Node (Ubuntu 16.04)
vNIC
vNIC
vNIC
vNIC
vNIC
vNIC
Physical Network Interface

Figure 5.1: Test Setup Block Diagram

Nova-G Compute is tested on a single physical computer. The physical computer has Intel Core i7-7500U CPU, 16 GB of RAM and Windows 10 64 bit operating system. We create three virtual machines to simulate the controller node and two compute nodes. One compute node hosts original Nova-Compute while other node hosts Nova-G Compute. The complete block diagram of test setup is show in the

Figure 5.1. The virtualization software to create the nodes is selected as Oracle VM VirtualBox. VirtualBox provides flexibility with VM management, has user-friendly interface and more importantly supports creating virtual NAT network for VMs. All of the steps of creating test setup and installing OpenStack components are explained in Section 3.1.

We install Ubuntu 16.04 LTS operating system to all nodes because OpenStack requires Ubuntu 16.04 or later versions. Each virtual machine has two virtual network interfaces. The first interface is used as management network for OpenStack with predefined virtual NAT network. This network uses custom IP domain `10.0.1.1`. The other interfaces are used to connect to The Internet with a network bridge interface. Each node can communicate with the other nodes over the virtual NAT network with a minimum delay.

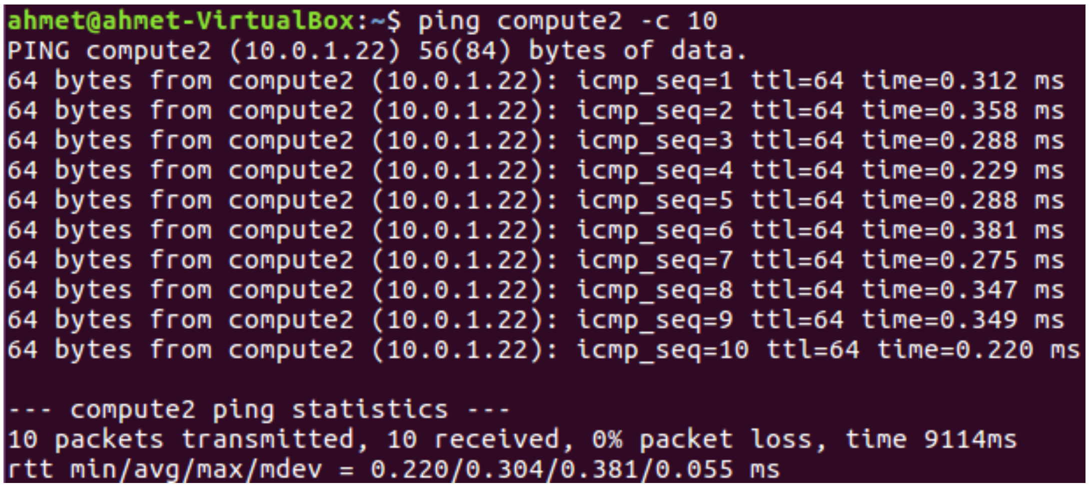
On the controller node, Keystone, Nova, Neutron and Glance OpenStack projects are installed and configured properly to work on given setup. Moreover, modifications of Nova data structures given in Section 4.1 are also applied to show that it does not affect operation of original OpenStack software. Standard Nova Compute is installed on the first node where Nova-G Compute is installed on the second node. Nova-G only depends on the RabbitMQ therefore RabbitMQ is also installed separately on this node. This set-up does not include any real heterogeneous hardware and is dedicated only for the performance evaluation of the extensions for the generalizations and Nova-G Compute. We added simple dummy hypervisor to test the functionality of Hypervisor Driver. The simple hypervisor gives response to create and terminate instance request with success. With such a basic hypervisor, Nova-G can continue its operation without any problem.

5.2 Basic Functional Tests

Before making any complex functional tests, basic tests are performed on the test setup. With help of these basic tests, we are able to detect problems beforehand. Moreover, these results create the base criteria to compare results within complex system.

5.2.1 Network Interface Verification

In test setup, there are two different networks. Virtual NAT network is used to communicate between controller node and compute nodes. OpenStack projects use NAT network for Oslo Messaging and API requests. Network interfaces are tested by basic `ping` command shown in the Figure 5.2. Command is issued at controller node to ping compute node. Mean RTT is measured as 0.304 ms. This RTT value is used as reference for other time measurements. With this test, we verify that created virtual NAT network working properly and OpenStack projects can communicate with each other.



```
ahmet@ahmet-VirtualBox:~$ ping compute2 -c 10
PING compute2 (10.0.1.22) 56(84) bytes of data.
64 bytes from compute2 (10.0.1.22): icmp_seq=1 ttl=64 time=0.312 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=2 ttl=64 time=0.358 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=3 ttl=64 time=0.288 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=4 ttl=64 time=0.229 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=5 ttl=64 time=0.288 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=6 ttl=64 time=0.381 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=7 ttl=64 time=0.275 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=8 ttl=64 time=0.347 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=9 ttl=64 time=0.349 ms
64 bytes from compute2 (10.0.1.22): icmp_seq=10 ttl=64 time=0.220 ms

--- compute2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9114ms
rtt min/avg/max/mdev = 0.220/0.304/0.381/0.055 ms
```

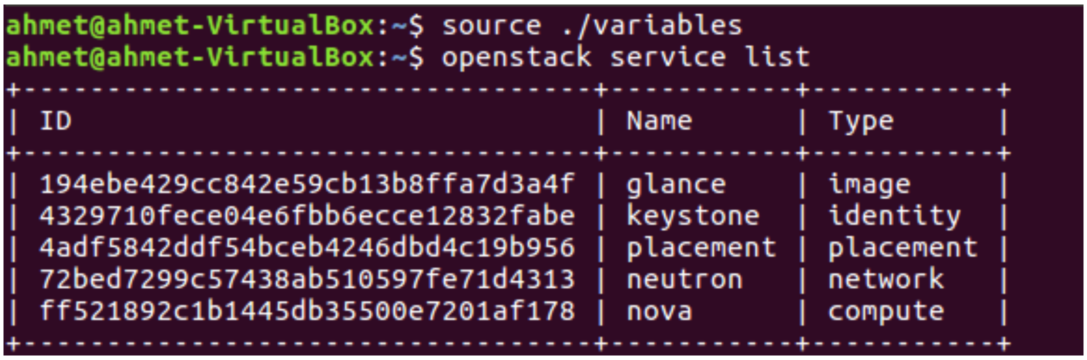
Figure 5.2: RTT measurements in virtual NAT network

5.2.2 OpenStack Operation Tests

After test setup connectivity is verified, OpenStack projects are installed to controller and compute nodes. OpenStack installation progress is explained in Section 3.1. After successful installation, basic OpenStack operation is tested. We use OpenStack command line interface (cli) to run commands. Before using cli, some global variables should be defined for easy authentication. Required global variables are defined in file named as `variables`. We include these variables into Ubuntu command line with `source ./variables` command. We listed all installed OpenStack services in controller node as shown in Figure 5.3.

Code Listing 5.1: Defined variables to use OpenStack command line interface

```
export OS_USERNAME=admin
export OS_PASSWORD=ADMIN_PASS
export OS_PROJECT_NAME=admin
export OS_USER_DOMAIN_NAME=Default
export OS_PROJECT_DOMAIN_NAME=Default
export OS_AUTH_URL=http://controller:35357/v3
export OS_IDENTITY_API_VERSION=3
```



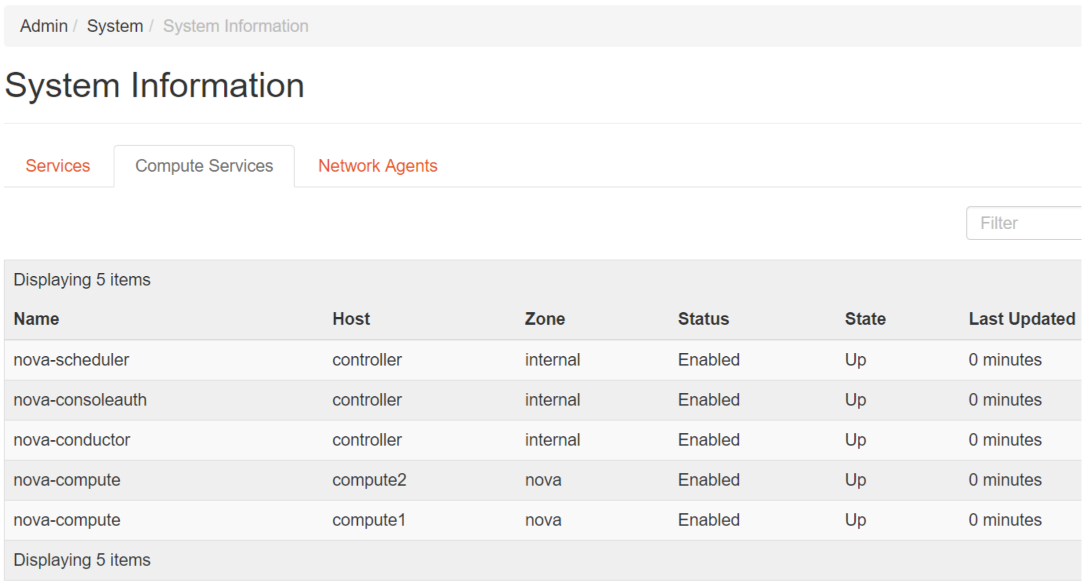
```
ahmet@ahmet-VirtualBox:~$ source ./variables
ahmet@ahmet-VirtualBox:~$ openstack service list
+----------------------------------+-----------+-----------+
| ID                               | Name      | Type      |
+----------------------------------+-----------+-----------+
| 194ebe429cc842e59cb13b8ffa7d3a4f | glance    | image     |
| 4329710fece04e6fbb6ecce12832fabe | keystone  | identity  |
| 4adf5842ddf54bceb4246dbd4c19b956 | placement | placement |
| 72bed7299c57438ab510597fe71d4313 | neutron   | network   |
| ff521892c1b1445db35500e7201af178 | nova      | compute   |
+----------------------------------+-----------+-----------+
```

Figure 5.3: OpenStack Service List

5.2.3 Verification of Nova-G Compute Compatibility

The connectivity between OpenStack services and their operation are tested. The OpenStack project, Horizon is used in this test. With the help of the Horizon Module, the state of services in OpenStack is viewed and tracked as shown in the Figure 5.4. Nova-G Compute is working on the compute node named as `compute1`. The original Nova Compute is active on the compute node called `compute2`. Both Nova Compute versions are identified by OpenStack system in the same manner named `nova-compute`.

Horizon shows both `compute1` and `compute2` nodes in `Up` state. Furthermore, these states are continuously updated as `Up`. This test verifies that Nova-G Compute properly operates and it is compatible with the standard OpenStack implementation.


Admin / System / System Information

System Information

Services | Compute Services | Network Agents

Filter

Displaying 5 items

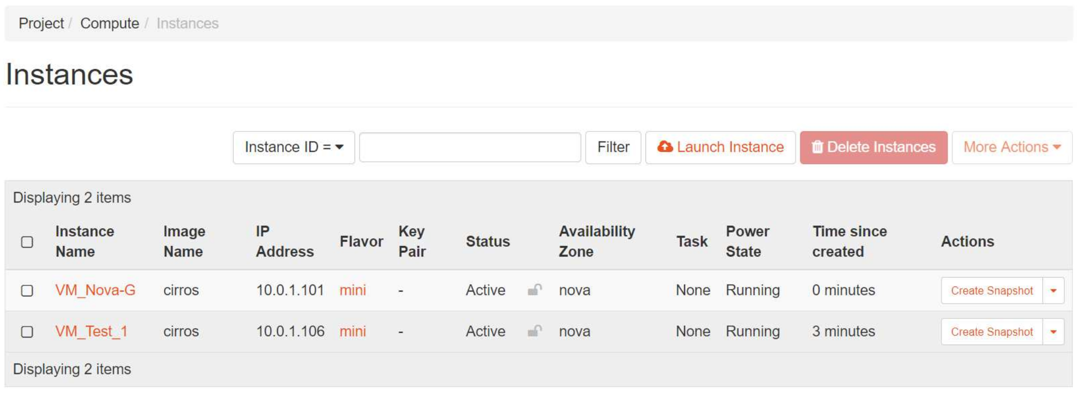
Name	Host	Zone	Status	State	Last Updated
nova-scheduler	controller	internal	Enabled	Up	0 minutes
nova-consoleauth	controller	internal	Enabled	Up	0 minutes
nova-conductor	controller	internal	Enabled	Up	0 minutes
nova-compute	compute2	nova	Enabled	Up	0 minutes
nova-compute	compute1	nova	Enabled	Up	0 minutes

Displaying 5 items

Figure 5.4: Testing of OpenStack services with Horizon.

5.2.4 Verification of Nova-G Compute VM Instantiation

In this test, Nova-G Compute instantiates a VM while standard Nova Compute instantiates another VM. To this end, the FPGAvisor communicates with the hypervisor driver in Nova-G . Consequently, as seen in Figure 5.5, OpenStack Horizon shows that `VM_Nova-G` VM is working on node `compute1` which runs Nova-G Compute. In addition, `VM_Test_1` is working on node `compute2` that runs the standard Nova Compute. Hence, it is verified that Nova-G module is capable of correctly getting VM requests of the users and instantiating the VMs.


Project / Compute / Instances

Instances

Displaying 2 items

	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
☐	VM_Nova-G	cirros	10.0.1.101	mini	-	Active	nova	None	Running	0 minutes	Create Snapshot
☐	VM_Test_1	cirros	10.0.1.106	mini	-	Active	nova	None	Running	3 minutes	Create Snapshot

Displaying 2 items

Figure 5.5: VM instantiation by Nova-G.

5.2.5 Nova-G Compute communication latency

Every 100 ms, compute nodes update their status by sending a message to the controller node. We measure an average latency of 0.5 ms over 100 measurements between `compute1` and the controller node when Nova-G Compute updates the status. Since the mean RTT between nodes is 0.355 ms as measured in Test 1, we conclude that Nova-G Compute works with a minumum overhead.

Nova-G Compute module uses `get_by_uuid` method of `Service` objects on Nova Conductor to make RPC requests. We measure the mean RPC request response time of Nova-G Compute over a total of 100 different RPC requests made in a time interval of 100ms. The results of measurements are summarized in Table 5.1.

Table 5.1: Nova-G Compute communication latency.

Status Updates	**Time (ms)**
`Minimum Latency`	`0.4`
`Maximum Latency`	`1.2`
`Average Latency`	`0.5`
RPC Requests	**Time (ms)**
`Minimum Delay`	`9`
`Maximum Delay`	`30`
`Mean Delay`	`14`

During these tests, the memory usage of Nova-G module is measured during normal operation by Python `psutil` library. Module has an average memory usage around 33.4 MB.

5.3 Functional Test Using Rally

After verifying the basic functionality of Nova-G Compute, we conduct performance tests using the benchmarking tool **Rally** [46] to test the overall operation of Nova-G Compute and compare its scalability with the standard Nova Compute [47]. Rally is an external tool that uses OpenStack services to test cloud infrastructure with a standard testing environment. Rally is used by many companies including Intel, IBM, Huawei and Cisco to test the performance and scalability of their clouds [48]. Here we note that, our capability of using Rally for Nova-G Compute tests is an important indicator to show that Nova-G Compute is well integrated to OpenStack.

Our test set-up is on a single physical machine, hence, we test Nova Compute and Nova-G independently so that they do not affect the performance of each other. Accordingly, in each test one compute node is used either with Nova Compute or Nova-G Compute. The corresponding test case block diagrams are shown in Figure 5.6.

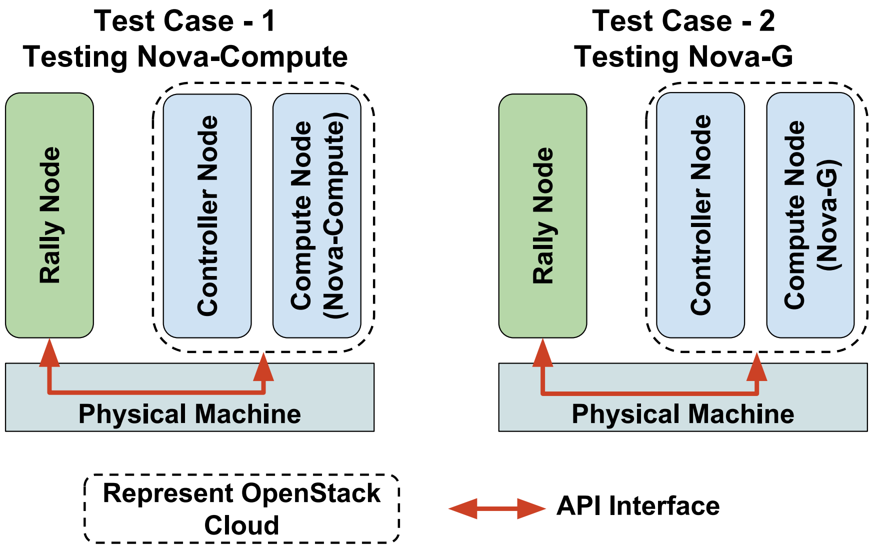


Figure 5.6: Test setup used by Rally.

5.3.1 Nova-G VM instantiate latency for single requests

We test the performance of Nova-G Compute in the complete operation cycle of standard VM creation, booting and deletions. To this end, Rally sends a VM creation request with a determined flavor and a VM image to the OpenStack cloud which is represented by our test set-up. Afterwards, Rally observes the state of the VM using OpenStack API. When the compute node finishes instantiating of a VM, Rally sends a delete request to the controller node to terminate the VM. Then it starts making a new VM request to the cloud. This process continues over a defined number of iterations. Figure 5.7 shows the booting and deletion of server times for Nova-G Compute for 50 iterations.

Total durations

Action	Min (sec)	Median (sec)	90% (sec)	95% (sec)	Max (sec)	Avg (sec)	Success	Count
nova.boot_server	3.046	3.144	3.339	3.47	8.994	3.294	100.0%	50
nova.delete_server	2.655	2.739	2.911	2.93	3.112	2.769	100.0%	50
total	**5.734**	**5.914**	**6.154**	**6.256**	**11.914**	**6.062**	**100.0%**	**50**
-> duration	*4.734*	*4.914*	*5.154*	*5.256*	*10.914*	*5.062*	*100.0%*	*50*
-> idle_duration	*1*	*1*	*1*	*1*	*1*	*1*	*100.0%*	*50*

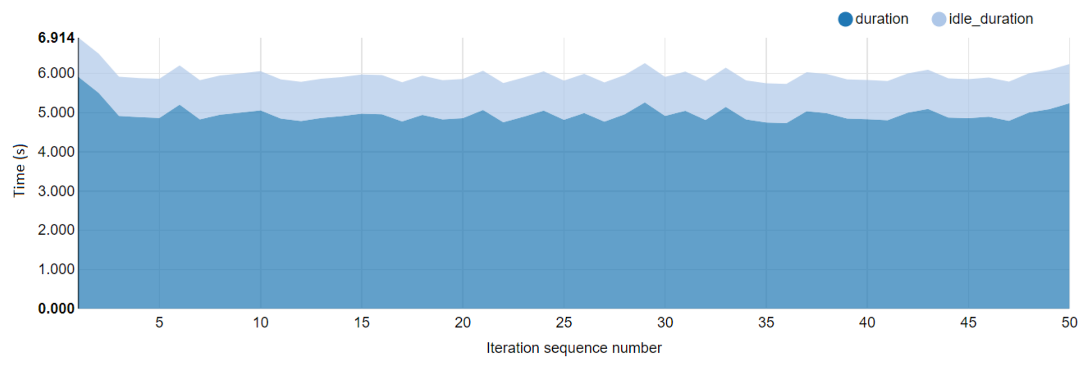


Figure 5.7: Nova-G Compute performance over 50 VM boot and delete iterations with Rally.

5.3.2 Scalability of VM instantiate latency comparison

Rally is configured to make VM instantiation request to OpenStack system in these tests. Rally has a configuration parameter for stating the number of concurrent VM requests. For example, if the concurrent VM request parameter is set to 2, Rally keeps track of VMs on compute nodes and maintains 2 VMs on the compute node for testing period. For this experiment, we run 5 tests where we change the concurrent VM request number from 1 to 5 in each test. A total of 50 VM requests are handled in each test. All tests are conducted both for original Nova Compute and Nova-G. Comparison between Nova Compute and Nova-G average VM instantiation time is shown in Figure 5.8. Here we note that FPGAvisor of Nova-G Compute does not execute the actual boot up actions. To this end, the hypervisor boot time of standard Nova Compute is subtracted from the total VM boot time for fair comparison. We observe that the boot time of Nova-G Compute is less than standard Nova Compute because of its lightweight implementation with essential components only.

We present the mean time figures normalized with the VM instantiation time of a single request for Nova-G Compute and standard Nova Compute respectively in Fig-

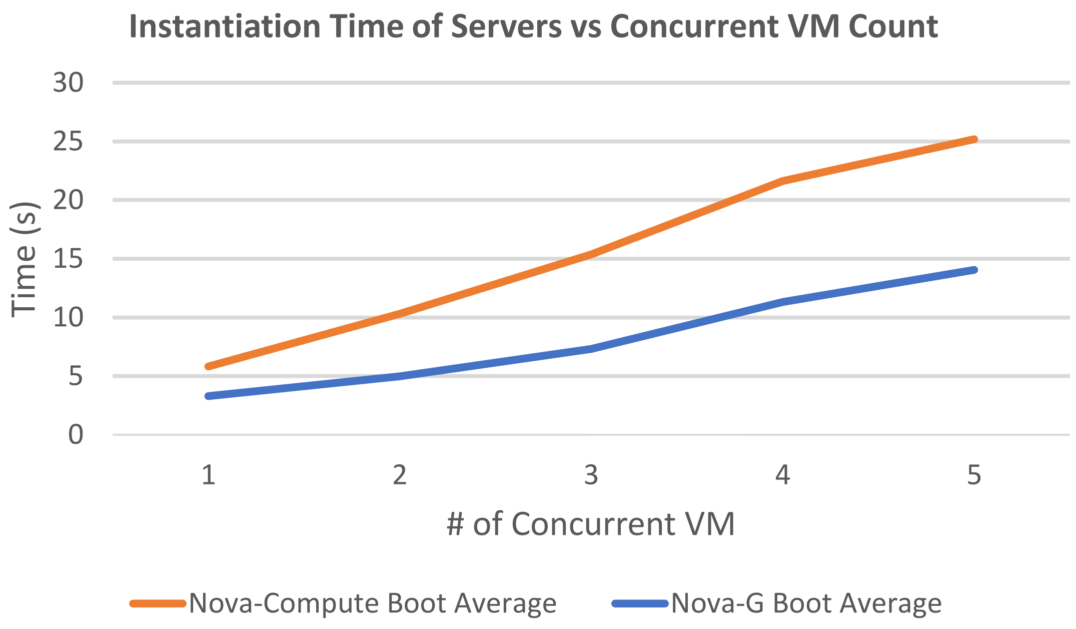


Figure 5.8: Nova Compute and Nova-G Compute VM boot times.

ure 5.9 for better demonstration of the Nova-G Compute scalability. Here we see that the VM booting time grows linearly with the number of concurrent VM requests. Hence, Nova-G Compute performs as goods as the original Nova Compute in terms of scalability.

Scalability of Nova-G is measured with respect to increasing concurrent VM count. Total execution time of 50 VM request is compared with original nova. The result is shown in Figure 5.10. Total execution time of 50 VM request slightly reduces when concurrent VM count increases in original Nova. However, it saturates after a certain point. When concurrent VM count is small, increasing VM count reduces total execution time. Because, software does not need to wait for new request. If concurrent VM count is more than one, Nova-Compute immediately starts processing other requests. Therefore, total execution time reduces. However, increasing concurrent VM count furthermore does not reduce total execution time. Because, software starts to run at full performance without waiting any new request. The important point is that our developed Nova-G shows similar behavior with original Nova.

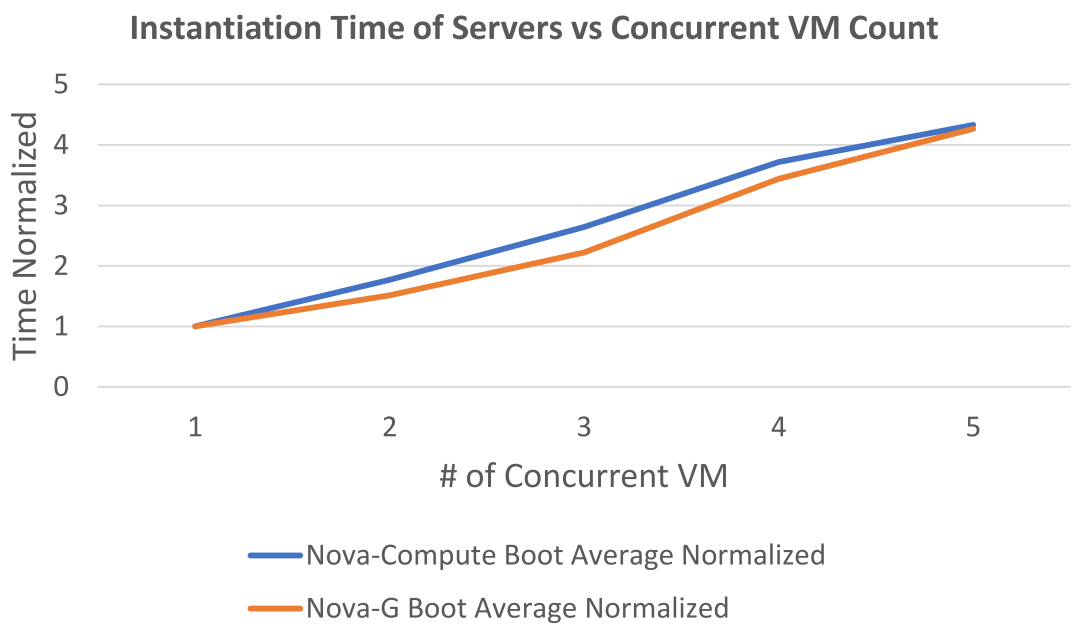


Figure 5.9: Nova Compute and Nova-G Compute scalability.

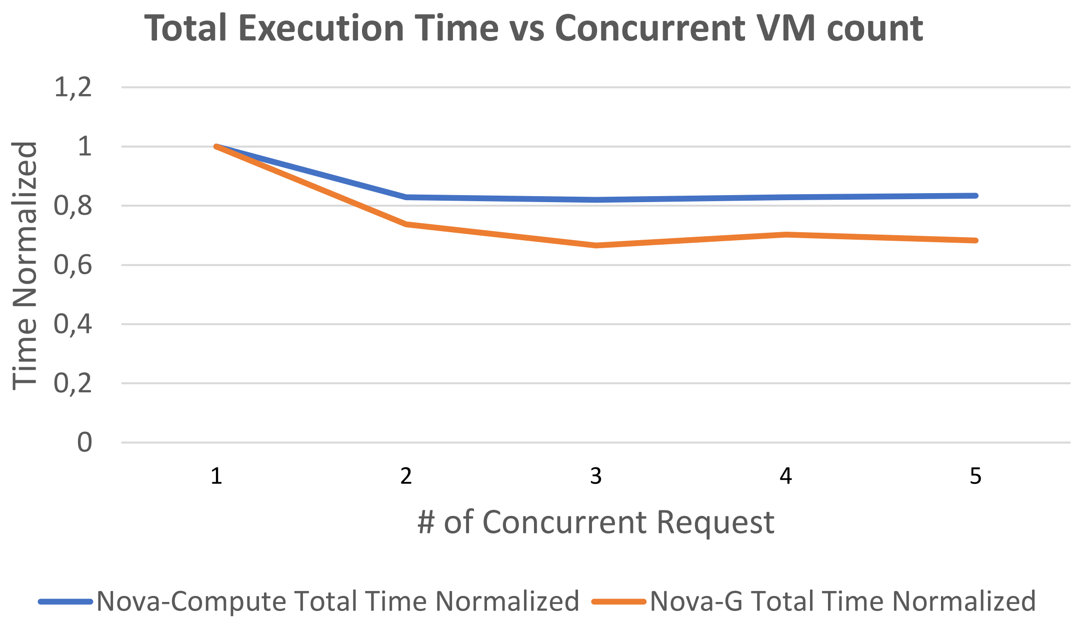


Figure 5.10: Nova Compute and Nova-G Compute Total Execution time

CHAPTER 6

CONCLUSION

In this thesis, we propose a generalization for OpenStack resource allocation project Nova to accommodate the new types of resources in hardware accelerated clouds. To this end, we extend the database structures of OpenStack by the new types of resources such as FPGA or GPU. This extension is at the same level with the standard resources of OpenStack database which enables standard Nova Scheduler algorithms to work with these new resource types. More importantly we develop a new lightweight Nova Compute module that we call Nova-G Compute that works seamlessly with the standard OpenStack services and can work on any hardware platform that supports Rabbit MQ thanks to its implementation that does not have OS dependencies. We further develop an FPGAvisor that gives access to FPGA resources to Nova-G Compute through its generalized hypervisor driver.

Our experimental evaluations with the standard OpenStack benchmarking tool Rally show that Nova-G correctly communicates with other OpenStack components and can boot VMs on generalized resources without any performance degradation with respect to standard Nova Compute. In this thesis, Nova-G Compute performance and functionality is evaluated on a test setup with virtual machines.

Our future work includes implementing Nova-G Compute for managing a real accelerated cloud system in a laboratory environment where the FPGA accelerators are implemented on Zynq SoC platform.

REFERENCES

[1] "OpenStack: An Overview." https://www.openstack.org/assets/marketing/presentations/OpenStack-General-Datasheet-for-US-A4-size.pdf. Accessed: 2019-03-17.

[2] "Openstack docs: Host networking." https://docs.openstack.org/install-guide/environment-networking.html. Accessed: 2019-07-24.

[3] "Openstack user stories." https://www.openstack.org/user-stories/. Accessed: 2019-02-02.

[4] P. Cooke, J. Fowers, G. Brown, and G. Stitt, "A tradeoff analysis of fpgas, gpus, and multicores for sliding-window applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 1, p. 2, 2015.

[5] "What is openstack?." https://www.openstack.org/software/. Accessed: 2019-02-02.

[6] "Openstack global market revenues from 2014 and 2021." https://www.statista.com/statistics/498552/openstack-market-size/. Accessed: 2019-03-21.

[7] "Openstack docs: Keystone, the openstack identity service." https://docs.openstack.org/keystone/latest/. Accessed: 2019-02-02.

[8] "OpenStack Docs: Identity API v3 (CURRENT)." https://developer.openstack.org/api-ref/identity/v3/. Accessed: 2019-07-05.

[9] "Openstack docs: Feature support matrix." https://docs.openstack.org/nova/latest/user/support-matrix.html. Accessed: 2019-07-06.

[10] "Openstack docs: Conductor." https://docs.openstack.org/mitaka/config-reference/compute/conductor.html. Accessed: 2019-07-06.

[11] "Openstack docs: Compute schedulers." https://docs.openstack.org/newton/config-reference/compute/schedulers.html. Accessed: 2019-07-06.

[12] "Openstack docs: Image service api v2 (current)." https://developer.openstack.org/api-ref/image/v2/index.html. Accessed: 2019-07-06.

[13] "Openstack docs: Networking api v2.0." https://developer.openstack.org/api-ref/network/v2/index.html. Accessed: 2019-02-02.

[14] "Amazon ec2 f1 instances." Accessed: 2019-03-22.

[15] "Graphics processing unit (gpu) | google cloud." https://cloud.google.com/gpu/. Accessed: 2019-03-22.

[16] "Cloud tpus - ml accelerators for tensorflow | cloud tpu | google cloud." https://cloud.google.com/tpu/. Accessed: 2019-03-22.

[17] "Cloud iot core | cloud iot core | google cloud." Accessed: 2019-03-22.

[18] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 7, IEEE Press, 2016.

[19] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 109–116, IEEE, 2014.

[20] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling flexible network fpga clusters in a heterogeneous cloud data center,"

in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), pp. 237–246, ACM, 2017.

[21] "Xilinx zynq-7000 soc." https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Accessed: 2019-03-22.

[22] S. Crago, K. Dunn, P. Eads, L. Hochstein, D. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. P. Walters, "Heterogeneous cloud computing," in *2011 IEEE International Conference on Cluster Computing*, pp. 378–385, Sep. 2011.

[23] L. Phan and K. Liu, "Openstack network acceleration scheme for datacenter intelligent applications," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 962–965, July 2018.

[24] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito, "Stack4things: a sensing-and-actuation-as-a-service framework for iot and cloud integration," *Annals of Telecommunications*, vol. 72, pp. 53–70, Feb 2017.

[25] "Cyborg-nova-glance interaction in compute node." https://docs.openstack.org/cyborg/latest/specs/rocky/approved/compute-node.html. Accessed: 2019-03-21.

[26] "Downloads – oracle vm virtualbox." https://www.virtualbox.org/wiki/Downloads. Accessed: 2019-07-24.

[27] "Openstack docs: Overview." https://docs.openstack.org/install-guide/overview.html. Accessed: 2019-07-24.

[28] "Openstack docs: Controller node." https://docs.openstack.org/install-guide/environment-networking-controller.html. Accessed: 2019-07-24.

[29] "Openstack docs: Compute node." https://docs.openstack.org/install-guide/environment-networking-compute.html. Accessed: 2019-07-24.

[30] "Openstack docs: Verify connectivity." https://docs.openstack.org/

install-guide/environment-networking-verify.html. Accessed: 2019-07-24.

[31] "Openstack docs: Openstack packages for ubuntu." https://docs.openstack.org/install-guide/environment-packages-ubuntu.html. Accessed: 2019-07-25.

[32] "Openstack docs: Sql database for ubuntu." https://docs.openstack.org/install-guide/environment-sql-database-ubuntu.html. Accessed: 2019-07-25.

[33] "Openstack docs: Message queue for ubuntu." https://docs.openstack.org/install-guide/environment-messaging-ubuntu.html. Accessed: 2019-07-25.

[34] "Openstack docs: Memcached for ubuntu." https://docs.openstack.org/install-guide/environment-memcached-ubuntu.html. Accessed: 2019-07-25.

[35] "Openstack docs: Etcd for ubuntu." https://docs.openstack.org/install-guide/environment-etcd-ubuntu.html. Accessed: 2019-07-25.

[36] "Openstack docs: Keystone install and configure." https://docs.openstack.org/keystone/pike/install/keystone-install-ubuntu.html. Accessed: 2019-07-25.

[37] "Openstack docs: Glance install and configure (ubuntu)." https://docs.openstack.org/glance/pike/install/install-ubuntu.html. Accessed: 2019-07-25.

[38] "Openstack docs: Nova install and configure controller node for ubuntu." https://docs.openstack.org/nova/pike/install/controller-install-ubuntu.html. Accessed: 2019-07-25.

[39] "Openstack docs: Horizon install and configure for ubuntu." https://docs.openstack.org/horizon/pike/install/install-ubuntu.html. Accessed: 2019-07-25.

[40] "Openstack docs: Neutron install and configure controller node." https://docs.openstack.org/neutron/pike/install/controller-install-ubuntu.html. Accessed: 2019-07-25.

[41] "Openstack docs: Networking option 1: Provider networks." https://docs.openstack.org/neutron/pike/install/controller-install-option1-ubuntu.html. Accessed: 2019-07-25.

[42] "Openstack docs: Nova install and configure a compute node for ubuntu." https://docs.openstack.org/nova/pike/install/compute-install-ubuntu.html. Accessed: 2019-07-25.

[43] "Download pycharm: Python ide for professional developers by jetbrains." https://www.jetbrains.com/pycharm/download/#section=linux. Accessed: 2019-07-26.

[44] A. Erol, A. Yazar, and E. G. Schmidt, "A generalization of openstack for managing heterogeneous cloud resources," in *2019 27th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, April 2019.

[45] A. Yazar, A. Erol, and E. G. Schmidt, "Accloud (accelerated cloud): A novel fpga-accelerated cloud archictecture," in *2018 26th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, IEEE, 2018.

[46] "what is rally?." https://docs.openstack.org/developer/rally/. Accessed: 2019-03-22.

[47] A. Erol, A. Yazar, and E. G. Schmidt, "Openstack generalization for hardware accelerated clouds," in *2019 28th International Conference on Computer Communications and Networks (ICCCN 2019)*, pp. 1–8, July 2019.

[48] "Overview rally." https://rally.readthedocs.io/en/latest/overview/overview.html. Accessed: 2019-03-17.

[49] "Firehose tracer — rabbitmq." https://www.rabbitmq.com/firehose.html. Accessed: 2019-07-27.

APPENDIX A

RABBITMQ DEBUGGING

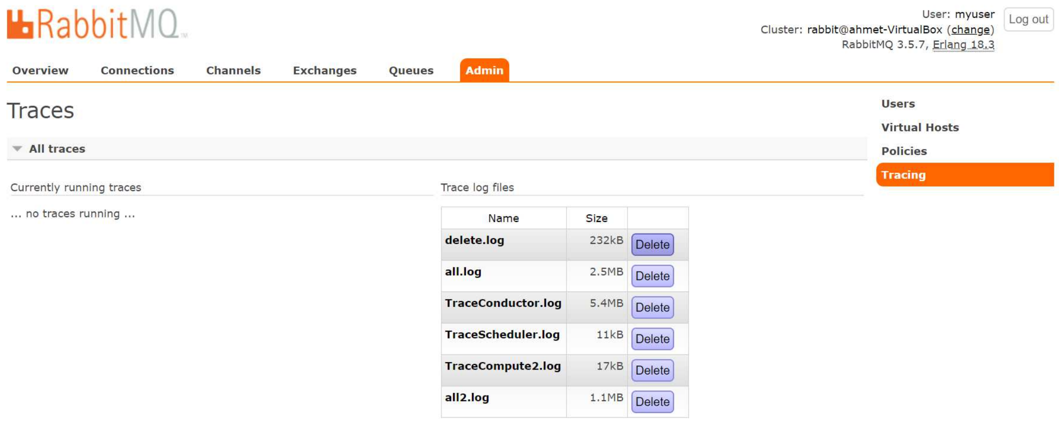
RabbitMQ server is installed on controller node. OpenStack projects use it for communication. In this section, RabbitMQ debugging tools are explained. It can be used to debug OpenStack messages. RabbitMQ provides a plugin to trace messages. It can be activated command below on controller node [49].

```
rabbitmq-plugins enable rabbitmq_tracing
```

RabbitMQ Management web interface can be reached with URL below.

`http://controller:15672`

RabbitMQ can be configured to all messages using tracing functionality of RabbitMQ. This configuration is shown in Figure A.1



User: myuser
Cluster: rabbit@ahmet-VirtualBox (change)
RabbitMQ 3.5.7, Erlang 18.3

Traces

Currently running traces

... no traces running ...

Trace log files

Name	Size	
delete.log	232kB	Delete
all.log	2.5MB	Delete
TraceConductor.log	5.4MB	Delete
TraceScheduler.log	11kB	Delete
TraceCompute2.log	17kB	Delete
all2.log	1.1MB	Delete

Figure A.1: RabbitMQ Tracing Configuration

APPENDIX B

OPENSTACK DATABASE TABLES

Table B.1: Nova `compute_nodes` Original Database Table

#	Name	Type	Collation	Attributes	Null	Default	Extra
1	**created_at**	datetime			Yes	*NULL*	
2	**updated_at**	datetime			Yes	*NULL*	
3	**deleted_at**	datetime			Yes	*NULL*	
4	**id**	int(11)			No	*None*	AUTO_INCREMENT
5	**service_id**	int(11)			Yes	*NULL*	
6	**vcpus**	int(11)			No	*None*	
7	**memory_mb**	int(11)			No	*None*	
8	**local_gb**	int(11)			No	*None*	
9	**vcpus_used**	int(11)			No	*None*	
10	**memory_mb_used**	int(11)			No	*None*	
11	**local_gb_used**	int(11)			No	*None*	
12	**hypervisor_type**	mediumtext	utf8_general_ci		No	*None*	
13	**hypervisor_version**	int(11)			No	*None*	
14	**cpu_info**	mediumtext	utf8_general_ci		No	*None*	
15	**disk_available_least**	int(11)			Yes	*NULL*	
16	**free_ram_mb**	int(11)			Yes	*NULL*	
17	**free_disk_gb**	int(11)			Yes	*NULL*	
18	**current_workload**	int(11)			Yes	*NULL*	
19	**running_vms**	int(11)			Yes	*NULL*	
20	**hypervisor_hostname**	varchar(255)	utf8_general_ci		Yes	*NULL*	
21	**deleted**	int(11)			Yes	*NULL*	
22	**host_ip**	varchar(39)	utf8_general_ci		Yes	*NULL*	
23	**supported_instances**	text	utf8_general_ci		Yes	*NULL*	
24	**pci_stats**	text	utf8_general_ci		Yes	*NULL*	
25	**metrics**	text	utf8_general_ci		Yes	*NULL*	
26	**extra_resources**	text	utf8_general_ci		Yes	*NULL*	
27	**stats**	text	utf8_general_ci		Yes	*NULL*	
28	**numa_topology**	text	utf8_general_ci		Yes	*NULL*	
29	**host**	varchar(255)	utf8_general_ci		Yes	*NULL*	
30	**ram_allocation_ratio**	float			Yes	*NULL*	
31	**cpu_allocation_ratio**	float			Yes	*NULL*	
32	**uuid**	varchar(36)	utf8_general_ci		Yes	*NULL*	
33	**disk_allocation_ratio**	float			Yes	*NULL*	
34	**mapped**	int(11)			Yes	*NULL*	

Table B.2: Nova-API `flavors` Original Database Table

#	Name	Type	Collation	Attributes	Null	Default	Extra
1	**created_at**	datetime			Yes	*NULL*	
2	**updated_at**	datetime			Yes	*NULL*	
3	**name**	varchar(255)	utf8_general_ci		No	*None*	
4	**id**	int(11)			No	*None*	AUTO_INCREMENT
5	**memory_mb**	int(11)			No	*None*	
6	**vcpus**	int(11)			No	*None*	
7	**swap**	int(11)			No	*None*	
8	**vcpu_weight**	int(11)			Yes	*NULL*	
9	**flavorid**	varchar(255)	utf8_general_ci		No	*None*	
10	**rxtx_factor**	float			Yes	*NULL*	
11	**root_gb**	int(11)			Yes	*NULL*	
12	**ephemeral_gb**	int(11)			Yes	*NULL*	
13	**disabled**	tinyint(1)			Yes	*NULL*	
14	**is_public**	tinyint(1)			Yes	*NULL*	