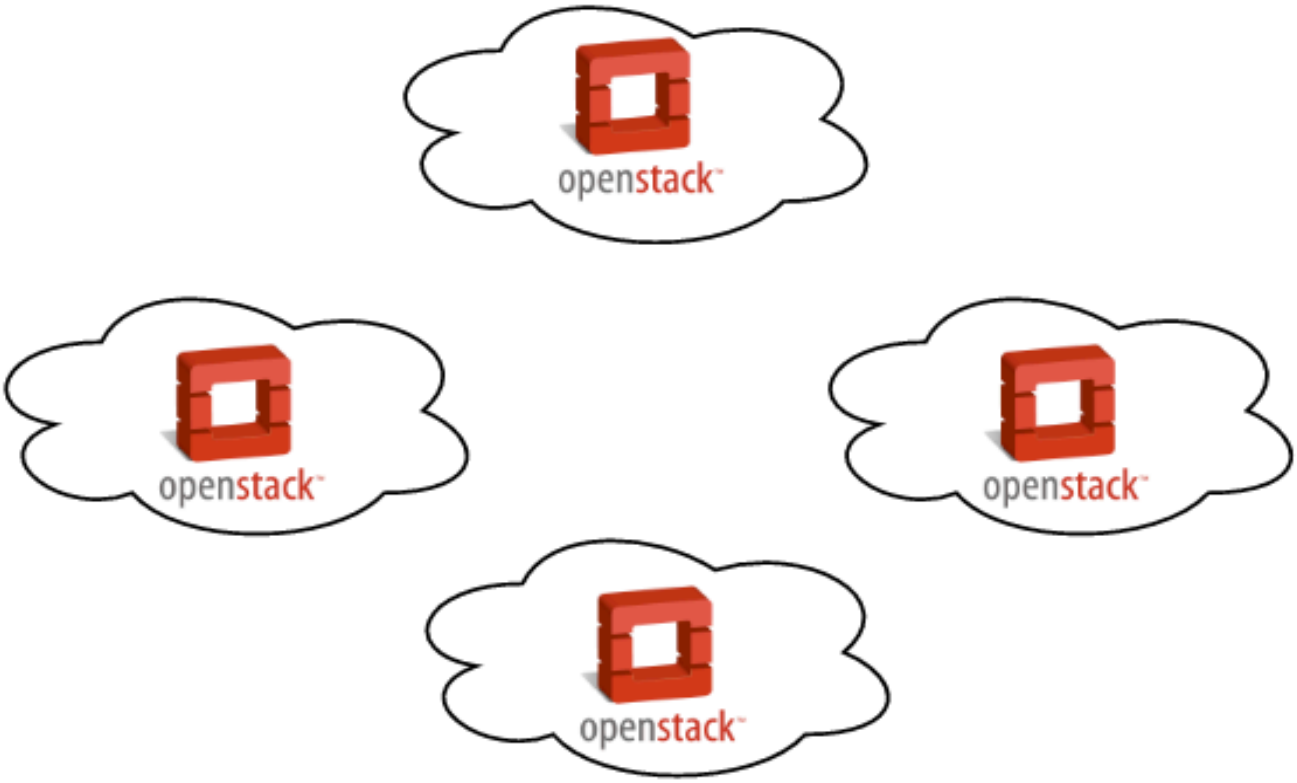
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Scaling OpenStack Clouds Using Peer-to-peer Technologies

Master's thesis in Computer Systems and Networks

XIN HAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Master's thesis 2016:NN

Scaling OpenStack CloudsUsing Peer-to-peer Technologies

XIN HAN

Department of Computer Science and Engineering
Chalmers University of Technology And University of Gothenburg
Gothenburg, Sweden 2016

Scaling OpenStack CloudsUsing Peer-to-peer Technologies
XIN HAN

Supervisors:
Vincenzo Gulisano, Department of Computer Science and Engineering
Joao Monteiro Soares, Ericsson
Fetahi Wuhib, Ericsson
Vinay Yadhav, Ericsson

Examiner:
Magnus Almgren, Department of Computer Science and Engineering

Master's Thesis 2016:NN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of P2P OpenStack cloud system. Each cloud is a standalone OpenStack cloud instance. Cloud instances are federated as one using P2P technologies.

Gothenburg, Sweden 2016

Scaling OpenStack Clouds Using Peer-to-peerTechnologies
XIN HAN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

OpenStack is an open-source software platform for cloud computing, mostly deployed as an infrastructure-as-a-service (IaaS) and has a user base in industry and academia to date. Despite its popularity, OpenStack still has drawbacks in terms of scalability of number of compute nodes (metal machines) in a single cloud instance. More precisely, a single standard OpenStack cloud instance does not scale well and fails to handle user request once its number of compute nodes reaches a particular amount. The particular amount depends on how the cloud instance is deployed and how many computing resources are provisioned to the cloud instance. This thesis proposes a solution that allows to scale up OpenStack cloud instances by using peer-to-peer (P2P) technologies. The solution abstracts multiple OpenStack cloud instances as one, providing the same user experience as using a single and standard OpenStack cloud instance. This thesis was done at Ericsson Research Department in Stockholm, Sweden. In the thesis, we design and develop a proof-of-concept of the solution by implementing a software agent which runs on an OpenStack cloud instance, working as a message broker and providing OpenStack services to users. Association of agents is achieved by an inexpensive group membership protocol – Cyclon. We evaluate our P2P-based solution by comparing its system performance with a standard OpenStack deployment in terms of response time, failure resistance and CPU utilization. Results show that it is feasible to integrate virtual resources across multiple OpenStack cloud instances while abstracting them as a single cloud instance. Moreover, it is also shown that the proposed approach has higher failure resistance to certain operations (e.g. upload image and boot virtual machine). In addition, the solution has no limitation on a number of cloud instances and its performance, such as response time, failure resistance and CPU utilization, improves with the increasing number of cloud instances.

Keywords: Cloud Computing, OpenStack, Peer-to-peer, Distributed Systems, Scalability.

Acknowledgements

I would like to show my sincere gratitude to my supervisor Vincenzo Gulisano and examiner Magnus Almgren and Joao Monteiro Soares, Fetahi Wuhib, Vinay Yadhav at Ericsson for sharing their pearls of wisdom and assisting with my thesis work during the whole process from proposing, planning, implementing to writing and David Bennehag and Yanuar T. Aditya Nugraha for their comments on an earlier version of the manuscript.

Xin Han, Stockholm, December 2016

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Cloud computing is an Internet-based computing which provides shared and on-demand computer processing resources in terms of networks, servers, storage and services. It enables enterprises and users to store and process their data with elastic capabilities. A cloud is a large-scale distributed system that dynamically provides computing resources as a service. The services provided by cloud are generally classified into different models such as Infrastructure-as-a-Service(IaaS), Platform-as-a-Service(PaaS) and Software-as-a-Service(SaaS) [1].

1.1.1 Peer-to-peer Overlay

Due to the growth of Internet in terms of size and speed in the past few years, deployment of networks and services indicate a shift from traditional server-client models to fully distributed Peer-to-peer(P2P) models. In P2P system, peers collaborate with each other, sharing duties and benefits. Peers are capable of operating large-scale tasks in a simple and scalable way by distributing responsibilities among participating peers, instead of depending on dedicated, expensive, and hard to manage centralized servers.

Presently, there are three main categories of P2P systems in terms of overlay management. One is structured P2P system which imposes a linkage structure among peers such as Distributed Hash Table [2]. Notable distributed overlays which use DHTs include the Kad network, the Coral Content Distribution Network, YaCy, the Storm botnet, and BitTorrent's distributed tracker [3]. Another one is unstructured P2P system, peers are created either based on proximity metric probabilistically or just randomly, instead of being imposed to be a particular structure on the overlay network by design. Gnutella, Gossip, and Kazaa are examples of unstructured P2P protocols [4]. The third category is the hybrid model which is a combination of server-client and P2P models to have a central server that helps peers find each other. Currently, the hybrid model is possession of better performance than either pure structured overlays or pure unstructured overlays as certain functions, such as searching, requires a centralized functionality but benefit from the decentralized aggregation of nodes provided by unstructured networks [5].

1.1.2 CYCLON Protocol

CYCLON is a gossip-based, highly scalable, robust, completely decentralized and inexpensive group management protocol for unstructured overlays. CYCLON protocol is shown to construct graphs that have low diameter, low clustering, highly symmetric node degrees, and that are highly resilient to massive node failures. Moreover, it is highly reactive to restoring randomness when a large number of nodes fail. CYCLON protocol does not maintain any global information or require any sort of administration, instead, each peer knows a small, continuously changing set of other peers, named its neighbors and periodically contacts a neighbor with the highest age from to exchange part of their neighbors.

1.1.3 RESTful API

Representational state transfer (RESTful) web services provide interoperability between computing systems on the Internet. It is a software architectural design pattern and a practical approach for web application development where a system needs to scale out or needs a simple way to interact with independent components [6]. An Application Programming Interface (API) that follows the RESTful fashion is called a RESTful API. RESTful API uses Uniform Resource Identifier (URI) to represent resources, it is scalable and stateless. The original HTTP verbs map to operations on resources. GET method is used for getting the resources; POST method is used for creating a new resource; PUT method is used for updating a resource by resource's id and DELETE method is used for deleting a resource or a collection of resources.

1.1.4 OpenStack

OpenStack is an open source IaaS cloud computing platform widely used in industry [7]. It consists of interrelated components that control hardware pools of computing, storage and networking resources throughout a data center and empowers users to provision resources. Notable users of OpenStack are NASA, PayPal and eBay. OpenStack embraces a modular architecture and comprises a set of interrelated services. Below is a quick breakdown of what they are called in OpenStack, and what they do.

- Compute service(Nova): Nova is a component which allows users to create and manage virtual machines (VMs) using machine images. It is the brain of a cloud. Nova facilitates management of VMs through an abstraction layer that interfaces with supported hypervisors.
- Identity service (Keystone): Keystone provides a central directory of users mapped to the OpenStack services. It is used to provide an authentication and authorization service for other OpenStack services throughout the entire cloud infrastructure.
- Image service (Glance): Glance provides image discovery, registration and delivery services to Nova, as needed.
- Networking (Neutron): A component for managing virtual networks. Neutron provides various networking services to cloud users (tenants) such as IP ad-

dress management, DNS, DHCP, load balancing, and security groups (network access rules, like firewall policies).

- Dashboard (Horizon): Horizon provides a web-based user interface to OpenStack services including Nova, Swift, Keystone, etc.

Figure 1.1 provides an overview of main services of OpenStack.

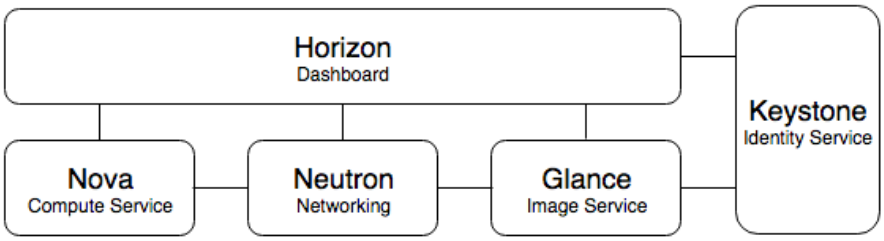


Figure 1.1: OpenStack Services Overview

An endpoint is no more than a URL that can be used to access a service of OpenStack. A service is any OpenStack service such as Compute service, Image service and Networking service. Through services, users are able to access and operate computing resources on demand. OpenStack exposes endpoints of services of internal components, such as Nova, Glance, Neutron, Keystone, to external users by OpenStack APIs. The APIs are RESTful interfaces in which requests are built as URL paths, users can directly send URL requests to OpenStack services to issue commands of managing resources in OpenStack cloud through cURL, OpenStack command-line client and REST clients[12]. Figure 1.2 illustrates an example of interaction between OpenStack and client via OpenStack API.

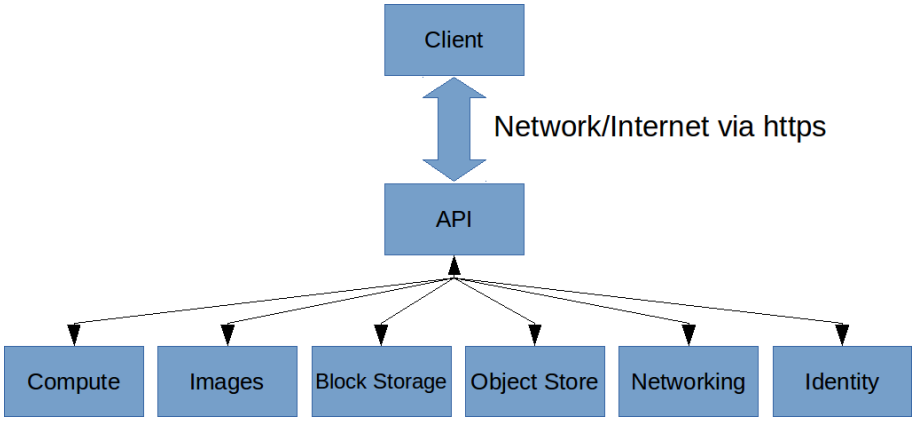


Figure 1.2: Example of Interaction Between OpenStack and Client via OpenStack API

1.2 Problem Description

In OpenStack's Nova component, a database is used to store current state of all objects among compute nodes and a message queue is used for collaboration be-

tween services in Nova such as nova-api, nova-conductor, nova-scheduler and nova-compute. Figure 1.3 shows the architecture of a standard Nova deployment. Nova services can be deployed on different physical machines. The message queue can be either RabbitMQ or Qpid and it is a central messaging component which enables communication and interoperability between Nova components in a loosely couple fashion.

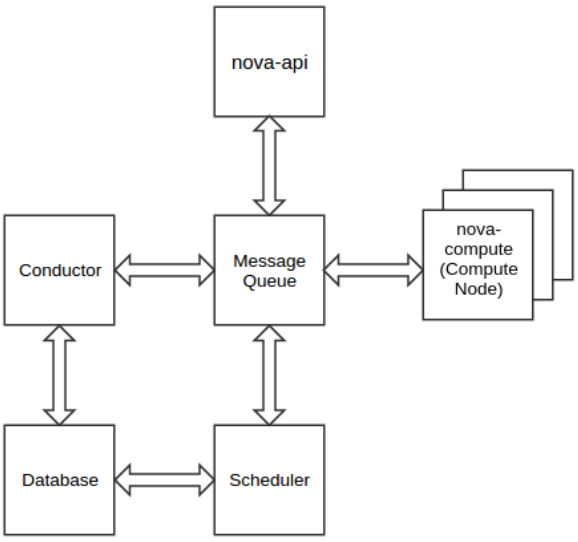


Figure 1.3: Architecture of Standard Nova deployment

Despite OpenStack's popularity, Nova still has limited scalability in terms of scaling up amount of compute nodes. More precisely, the database and the message queue of OpenStack Nova are identified as bottlenecks [8, 9]. In a test case of Cisco's testing report [9], the test was monitored using RabbitMQ management plugin and was done by adding compute nodes to OpenStack cloud simultaneously. The finding of this test case was that number of RabbitMQ Socket Descriptors was the limiting factor at 147 compute nodes per cloud controller. From Cisco's testing result, we know that OpenStack does not scale well when a quantity of compute nodes reaches a specific number.

As an open source project, OpenStack has been significantly developed for years, however, the scalability drawback has not been perfectly solved yet. For the sake of solving the scalability issue, one can look at this problem from different angles. One way to look at this problem is component wise. A solution could be fixing the problem of each individual component, for instance, message queue of Nova component. However, due to OpenStack's complexity, solving scalability problem in this way would be extremely complicated and would take a risk of affecting integrity with other components. Besides, in order to scale an OpenStack cloud instance (cloudlet) in terms of compute nodes, even if the message queue issue of Nova is solved, all other components, such as Neutron and Glance, still need to be scalable to satisfy service demand of increased number of compute nodes as well. From another angle to look at this problem, instead of aiming to solve scalability issue of an individual component, we can solve the scalability issue on a higher

and abstracted level by associating multiple clouds instances as one cloud cluster, regardless of the scalability properties of each cloud instance or individual OpenStack component. A possible solution could be scaling OpenStack cloud instances using P2P technologies and providing an abstraction of these collaborated cloud instances as one in order to enable the abstracted system to allocate as many compute nodes as all associated cloud instances could do so.

1.3 Motivation

This thesis aims to provide a solution for scaling OpenStack cloud instances on cloud cluster level. Scaling standalone OpenStack cloud instances using P2P technologies and abstracting these cloud instances as one without affecting end-user experience are what we aim to solve in this thesis. In order to associate multiple OpenStack cloud instances, an additional software agent is supposed to be implemented and deployed on each cloud instance, working as a message broker and providing services of its affiliated OpenStack cloud to end users. Agents on different cloud instances should be interconnected to each other to comprise a P2P network. By abstracting OpenStack cloud instances through agents, physical resources such as computing capacity, storage and network are capable of being shared across distinct cloud instances. By scaling cloud instances using P2P technologies, an OpenStack cloud instance is supposed to easily join or leave the P2P system without causing any improper functioning and the whole system should be scalable and flexible.

This thesis report is structured as follows. Chapter 2 provides an overview of agent design. Chapter 3 discusses the implementation of agent in details. Chapter 4 evaluates the proposed solution and shows evaluation results, whereas chapter 5 and chapter 6 provide a discussion of future work and related work, separately. Finally, chapter 7 concludes this thesis.

2
Agent Design

The chapter presents what the agent is for, the system architecture of our proposed P2P system with agents plugged on each OpenStack cloud instance, how does the pluggable agent integrate with OpenStack cloud instance and group membership among cloud instances.

2.1 Agent

An agent is a message broker which is supposed to be implemented and plugged on each OpenStack cloud instance. Agent acts as proxy between OpenStack cloud instances and end users which enables resource federation and collaboration among cloud instances. In standard OpenStack deployment, a single user request is not feasible to forwarded across multiple cloud instances and virtual resources that a user acquires on distinct cloud instances lacks relevant association. While in our proposed P2P system, each agent runs on its affiliated OpenStack cloud instance and provides its own dedicated RESTful service endpoints to serve end users. More precisely, agent maps received user request to a standard OpenStack user request and forwards the request to corresponding cloud instance's RESTful service endpoint. Besides, agent federates computing resources from cloud instances and is capable of provisioning or allocating resources on-demand and it maintains associated relation among resources that user acquires on distinct cloud instances. In addition, each agent keeps possession of a list of cloud instances based on the CYCLON group membership protocol that we will discuss in the following section. These cloud instances on the list are called neighbors and the list is called neighbor list. From an user's view, an agent is an entry point of the abstracted P2P system and cloud instances behind the agent are transparent. Instead of sending request to original OpenStack service endpoints, user sends requests to agent's service endpoints by the same manner of using a standard OpenStack cloud instance. With agents, multiple cloud instances act as one.

2.2 System Architecture

In OpenStack, a region is a logical fence used to group OpenStack services in close proximity to one another [10]. In other words, an OpenStack cloud instance is an independent region. With OpenStack multi-region deployment, the identity service Keystone, which is used by all other OpenStack components for authentication and authorization, is capable of serving multiple OpenStack cloud instances.

Our proposed solution constructs a hybrid P2P system that is based on the OpenStack multi-region deployment in which all cloud instances share identity service provided by one centralized Keystone instance. With agents deployed, our solution abstracts a flat structure which consists of one Keystone instance and multiple cloud instances without any hierarchical layers. Each individual cloud instance with its affiliated agent acts as one peer in the P2P system. Except requiring identity service from Keystone instance, every peer shares resources amongst each other. Due to advantages of flat P2P structure, load is able to be uniformly distributed to participating peers in a sense and each individual peer is easy to join or leave the P2P system without affecting operation of any other peer [11]. From a user's perspective of view, the whole abstracted P2P system serves as same as a single standard OpenStack cloud instance. Figure 2.1 illustrates the system architecture of our proposed P2P system.

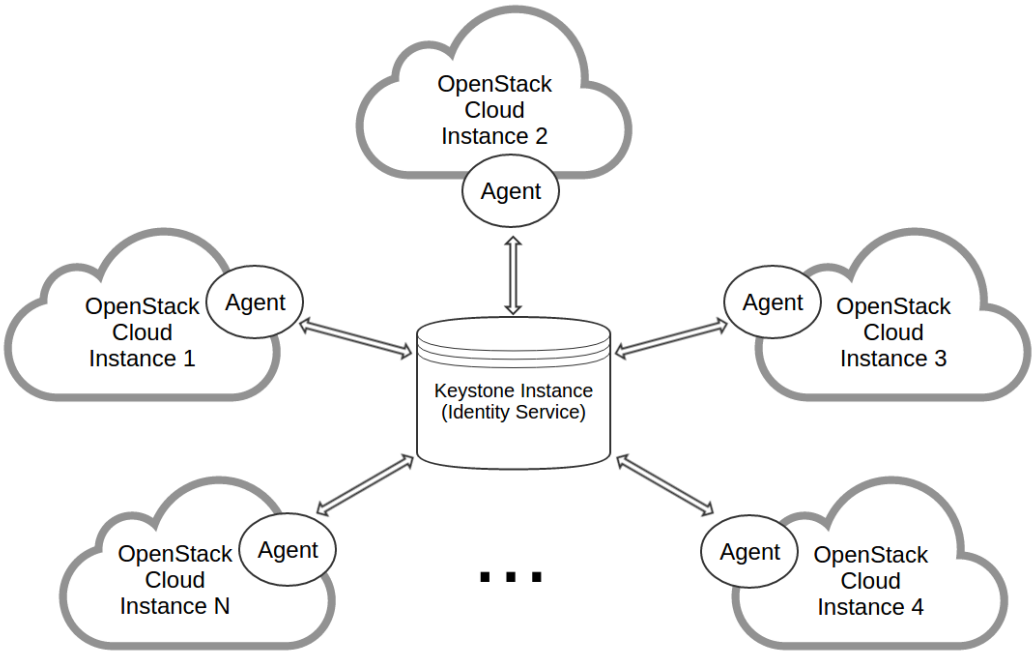


Figure 2.1: System Architecture of P2P OpenStack Clouds System

2.3 Integration with OpenStack

As we mentioned in previous chapter, making changes in OpenStack is complicated and risky of causing cloud to work in improper way. To this point, agent is designed to be integrated with OpenStack cloud instance by running as a stand alone service. Since OpenStack exposes its APIs in RESTful fashion which allows a user to operate actions via cURL, OpenStack command-line client, REST clients or OpenStack SDK based program [12]. Consequently, agent is also capable of operating actions on OpenStack cloud via OpenStack RESTful APIs as desired. In addition, agent is

designed to provide RESTful HTTP service to work as a message broker which sites between its affiliated OpenStack cloud instance and users, and performs mapping of agent service endpoints and cloud service endpoints. More precisely, agent listens on a specific port and waits for requests from users. Figure 2.2 illustrates how agent works as message broker between cloud instances and users. As shown in this figure, user interacts with agent via RESTful request and agent acquires services from cloud instances via RESTful request as well. By given an example, a user sends a request to the service endpoint of agent, once the request is received by the agent, the agent firstly do a mapping between its service endpoint and service endpoint of OpenStack cloud instance, then it forwards the request to the corresponding cloud instance's service endpoint via OpenStack's RESTful API.

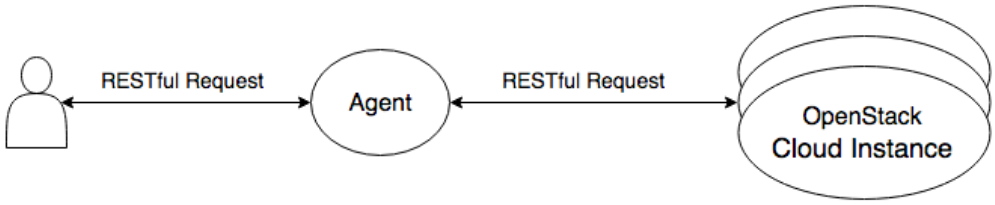


Figure 2.2: Integration with OpenStack

2.4 Resource Federation

Cloud based services are a growing business trend in the IT industry, where service providers establish cloud and offer computing resources (infrastructure, platform and software) to consumers. Consumers often require virtual resources across multiple cloud instances, to settle their application needs. Single cloud provider may not be able to address such requests because of lack of capacity or presence in multiple cloud instances. Enterprises are adopting cloud based service to address their growing computing workload and they need resources across multiple cloud instances. These instances sometimes span across multiple geographical locations. In this thesis work, the desired solution is to provide users an abstraction of OpenStack cloud instances as one based on P2P overlay without affecting user experience. In order to achieve this goal and efficiently utilise virtual resources of various OpenStack cloud instances, such as CPU, RAM, network, image file and storage, resources of these cloud instances are supposed to be federated and provided altogether to users. Resource federation is recognized as a promising mechanism aimed at the interconnection of heterogeneous resources across several independent cloud infrastructures. In order to provide a larger-scale and higher performance infrastructure, federation enables on-demand provisioning of complex service and user can get access to much larger pools of resources. However, OpenStack virtual resources are not feasible to be shared across distinct clouds. For instance, an image file of cloud A can not be used in cloud B when boots a VM in cloud B. In order to solve this problem, the agent is designed to be capable of managing resources across distinct cloud instances. Resource provisioning is required to efficiently allocate limited resource on

resource provider cloud. Also, Provisioning is needed to provide sense of ownership to end user. Agent is responsible for provisioning remote resources. It also maintains provisioning information in a local database. The provisioning info will be used for the purpose of resource info query. Figure 2.3 shows an example of resource federation, a user owns virtual resources from remote clouds on a local project through an agent. One benefit of resource federation is that a user can use a single project in host cloud to scope all the remote virtual resources across cloud instances.

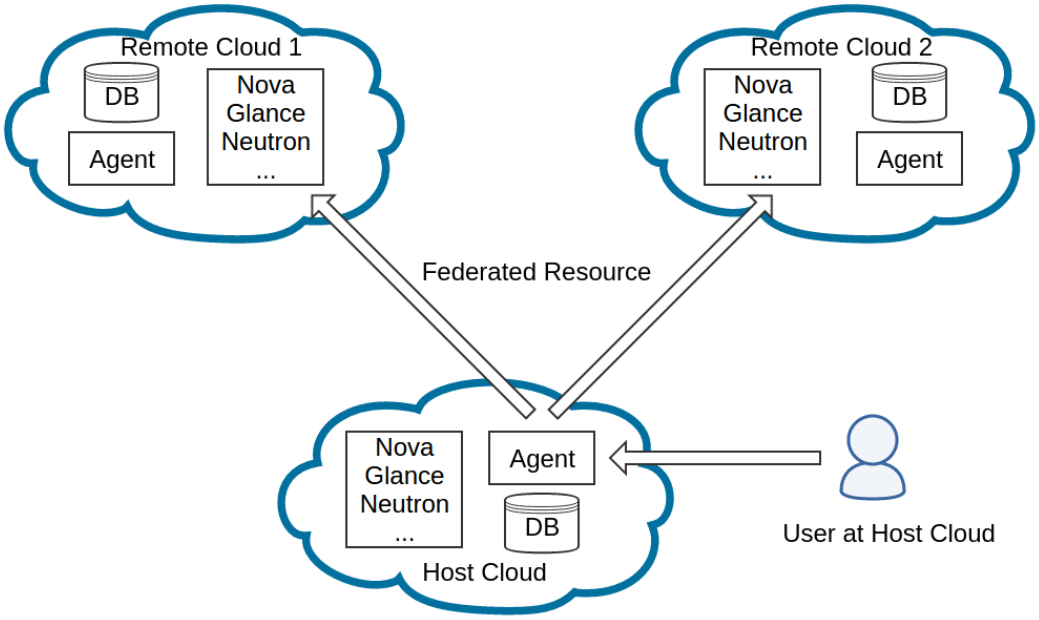


Figure 2.3: Example of Resource Federation

3

Agent Implementation

This chapter discusses implementation details of the agent, including agent architecture, agent database, agent HTTP server, agent proxies and scope of OpenStack APIs that the agent supports. The software agent is purely developed in Python and mainly uses modules come with OpenStack installation.

3.1 Agent Architecture

The agent consists of multiple components, an Agent DB, an HTTP server, four proxies, a CYCLON component in which a process of CYCLON protocol runs periodically and a Scheduler component. The HTTP server provides service for Agent API, Keystone Proxy, Glance Proxy, Neutron Proxy and Nova Proxy. Figure 3.1 shows agent architecture and responsibilities of each component are listed below.

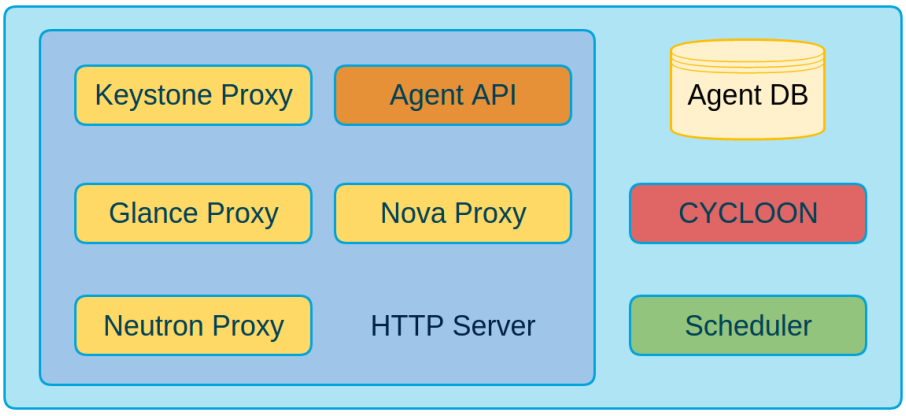


Figure 3.1: Agent Architecture

- Agent DB: Runs MySQL database service and is responsible for storing user's resource allocation information.
- HTTP Server: Listens on a specific port, forwards request from a user to OpenStack cloud and deliver response from OpenStack cloud to the user.
- Agent API: Exposes agent's service endpoint to user by providing RESTful APIs.
- Keystone Proxy: Works as proxy and maps agent's service endpoint to OpenStack Identity service endpoint.
- Glance Proxy: Works as a broker and maps agent's service endpoint to OpenStack Image service endpoint.

- Neutron Proxy: Works as broker and maps agent's service endpoint to OpenStack Networking service endpoint.
- Nova Proxy: Works as broker and maps agent's service endpoint to OpenStack compute service endpoint.
- CYCLON: An implementation of Cyclon protocol. Maintains a list of its affiliated agent's neighbors and exchanges its view of neighbors with other agents periodically.
- Cloud Selection Scheduler: Comprised of filters and weighers. When receives a virtual resource creation request, selects an OpenStack cloud instance to create requested virtual resource based on strategies of filters and weighers.

Every agent component collaborates with each other to serve users. Once user request arrives at agent HTTP server, Agent API component distributes the request to Keystone proxy, Glance proxy, Neutron proxy or Nova proxy relevantly, waiting for the response from the related proxy. These four proxies are brokers between agent and OpenStack clouds, they receive mapped user request from Agent API component and forwards the request to relevant OpenStack cloud according to information stored in Agent DB in the case of querying or deleting a virtual resource, or decision made by cloud selection scheduler in the case of creating virtual resource [14]. Once receives a request of creating or deleting a virtual resource, the Agent DB is updated after the relevant request is successfully operated by OpenStack cloud. The CYCLON component runs Cyclon protocol, periodically swaps neighbor list with another agents in P2P system and provides neighbors' information to its affiliated agent when choosing an OpenStack cloud instance to create a virtual resource. Details of agent components will be discussed in following sections.

3.2 Agent DB

The Agent DB components provides persistent data storing service. Instead of directly using OpenStack database, in this solution, a dedicated database is used by the agent to insert, delete and read data of users' virtual resource information. As a proof-of-concept, this solution only considers entities such as project, user, image, network, subnet and VM. A project has multiple users and a user can belong to multiple projects with various authorities; A project can have several networks and a network can be shared with several projects; A network contains multiple subnets and a subnet can be shared with multiple VMs; A project has multiple VMs and an image boots several VMs. Except for VM, each entity such as image, network, subnet has a unique id originally created by its affiliated OpenStack cloud instance, called uuid_cloud in this solution, and a unique id created by its affiliated agent, called uuid_agent in this solution. The uuid_cloud is only unique within entity's affiliated OpenStack cloud, while the uuid_agent is globally unique within the whole P2P system. Since an entity excepts VM can be located at multiple OpenStack cloud instances, so the relationship between uuid_agent and the uuid_cloud is one-to-many. In other words, one uuid_agent can be mapped to multiple uuid_cloud. Instead of sending a request with entity's original unique id created by OpenStack cloud instance when deleting or querying information of network, subnet or image,

user sends a request with the globally unique id created by the agent. As each VM is globally unique and a VM is infeasible to be at multiple clouds, so it's not necessary to generate uuid_agent for VM. The unique id of a VM stored in agent database is just the unique id originally created by its affiliated OpenStack cloud instance. Figure 3.2 show an Entity-Relationship diagram (ER-diagram) of image, network, subnet and VM.

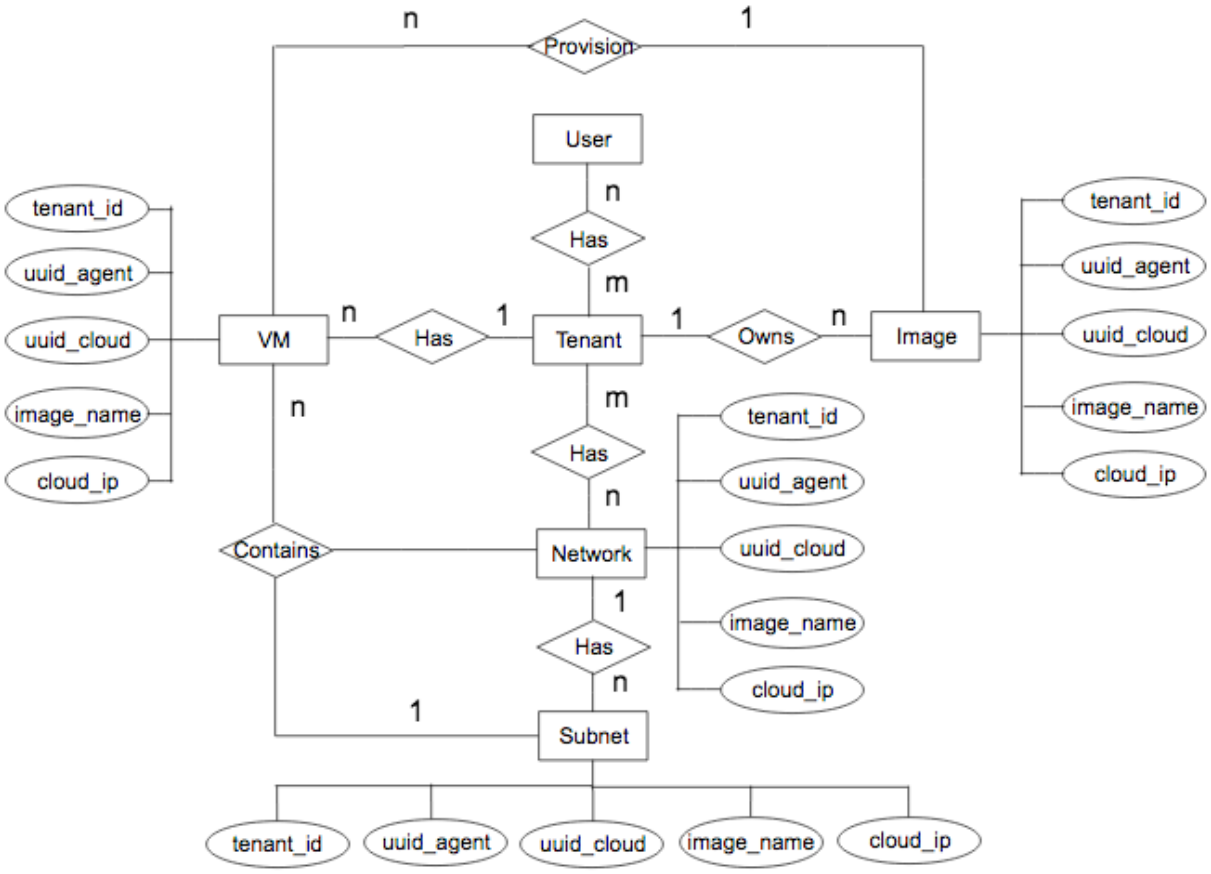


Figure 3.2: Entity Relationship Diagram of Image, Network, Subnet and VM

The Agent DB stores data for entities such as image, network, subnet and VM, and it is updated only in case of creating or deleting a virtual resource successfully. As MySQL comes with OpenStack installation, so in this solution MySQL is used as the dedicated database in Agent DB. The agent database contains four tables which are Image table, Network table, Subnet table and VM table. A Python SQL toolkit and Object Relational Mapper (ORM) – SQLAlchemy is used to interact with MySQL. SQLAlchemy provides efficient database access and it is able to classifies data set into object models in a decoupled way [15]. As a virtual resource is possible in multiple clouds, uuid_cloud holds the property of primary key.

Agent DB works as a supporting component to store user's virtual resource allocation information for this P2P solution. When the agent receives a request of querying virtual resource information, it first looks up through its Agent DB and maps uuid_agent to uuid_cloud, then forwards request to relevant OpenStack cloud

instance's service endpoint by using the mapped uuid_cloud. The case of receiving a request of deleting resource is similar to receiving a request of querying, the only difference is if the virtual resource is successfully deleted by OpenStack cloud, the agent updates its local database by removing the entry for the deleted resource. In the case of receiving a request of creating a virtual resource, agent forwards the request to service endpoint of chosen OpenStack cloud instance, if the request is successfully operated by cloud instance, then agent generates a globally unique id for the created virtual resource and update its local agent database by inserting a new entry to the relevant data table.

3.3 Agent HTTP Server

Agent HTTP server is based on Python Web Server Gateway Interface (WSGI). WSGI is capable of serving as interface between web applications and servers [16]. Conceptually, WSGI interface consists of "server" side and "application" side, where the "application" side is invokable by the "server" side. In our solution, the agent HTTP server is implemented using a Python library – Eventlet. Eventlet is a concurrent networking library for Python that provides built-in WSGI interface [17] and uses libevent, an event notification library, for highly scalable non-blocking I/O [18]. Apart from this two strengths, Evenlet is developed based on green threads which are capable of doing networking tasks and work in cooperative fashion [19]. The agent HTTP server initiates a pool of green threads which pools control concurrency at boot time. This is beneficial to limit amount of memory an application can consume or make application behave consistently in the case of handling unpredictable incoming connection. By default, the size of the green pool is 1000. Every time, the agent HTTP server receives a RESTful request from a user, it firstly parses this request based on requested API URL then invoke "application" side which is the relevant callable proxy component, meanwhile, the user remains waiting until gets any response. Once the relative operation is done by proxy component, the agent HTTP server sends a response back to the user and terminates the connection between user and agent HTTP server. In addition, a waiting time is defined at the server side, instead of making a user wait for unnecessary long time, if the relevant proxy component does not respond after a timeout, the agent HTTP server raises an exception and sends an error message back to the user and terminate the connection.

3.3.1 Agent API and Proxy Components

Agent sits between its affiliated cloud and users, acting as a message broker, receiving RESTful requests from users, mapping Agent APIs to OpenStack APIs and forwarding mapped request to OpenStack service endpoints. Agent exposes its internal services by providing RESTful APIs in the same manner as OpenStack RESTful APIs. The only difference is that various OpenStack service endpoints listen on their dedicated ports, instead, agent service endpoints listen on one specific port. An Agent API scheme is shown below:

http://<agent-ip>:<agent-port>/<suffix>

Agent parses received request based on HTTP method and URL suffix, then forwards the parsed request to the relevant proxy component. The mapping between agent service endpoint and OpenStack service endpoint is operated by the relevant proxy component such as Keystone Proxy, Glance Proxy, Neutron Proxy and Nova Proxy. Each proxy component maps parsed request from Agent API to OpenStack service endpoint and forwards mapped request to OpenStack service endpoint, waiting for the response from OpenStack. Figure 3.3 illustrates the work flow of handling user request by Agent API and relevant proxy components.

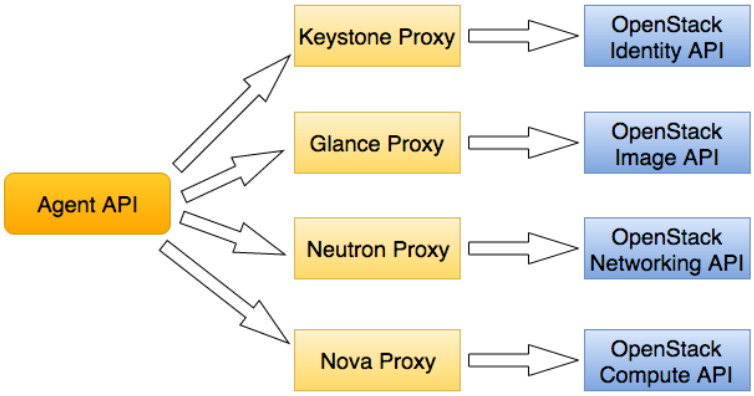


Figure 3.3: Work Flow of Handling User Request by Agent API and Proxy Components

As a proof of concept, agent supports partial major OpenStack APIs which supports Create, Read, Delete (CRD) operations of OpenStack Identity API, Compute API, Image service API and Networking API. Each letter in the acronym CRD can map to an HTTP method. Table 3.1 shows the mapping between CRD operations and HTTP methods.

Table 3.1: Mapping Between CRD And HTTP Methods

Operation	Method
Create	PUT/POST
Read	GET
Delete	DELETE

Agent API is strictly based on RESTful API, and exposes agent's service endpoints to users. From a user's perspective, the user just needs to send a request to Agent API endpoint, instead of sending a request to standard OpenStack API endpoint. As Agent API is in an OpenStack fashion, in which user has the same experience ofV using standard OpenStack.

OpenStack Identity service generates authentication tokens that permit access to OpenStack services' REST APIs. Users obtain this token and the URL endpoints

for other service APIs by supplying their valid credentials to the authentication service. Table 3.2 shows APIs mapping between OpenStack Agent service and Identity service.

Table 3.2: Mapping Agent Service APIs and Identity Service APIs

Method	Agent Service Endpoint	Identity Service Public Endpoint	Usage
POST	:port/v3/auth/tokens	:5000/v3/auth/tokens	Authenticate identity and generates token
GET	:port/v3/auth/tokens	:5000/v3/auth/tokens	Validate and show information for token

OpenStack Image service is responsible for storing virtual machine images and maintaining a catalog of available images. Table 3.3 shows the mapping between OpenStack Image service APIs and Agent APIs.

Table 3.3: Mapping Between Agent Service APIs and Image APIs

Method	Agent Service Endpoint	Image Service Public Endpoint	Usage
GET	:port/v2/images	:9292/v2/images	List public virtual machine (VM) images
POST	:port/v2/images	:9292/v2/images	Creates a virtual machine (VM) image
GET	:port/v2/images/{image_id}	:9292/v2/images/{image_id}	Show details for an image
DELETE	:port/v2/images/{image_id}	:9292/v2/images/{image_id}	Delete an image
PUT	:port/v2/images/{image_id}/file	:9292/v2/images/{image_id}/file	Upload binary image data
GET	:port/v2/images/{image_id}/file	:9292/v2/images/{image_id}/file	Download binary image data

OpenStack Networking API is used for managing virtualized networking resource such as networks, subnets and ports. The APIs mapping between Agent service and OpenStack Networking is shown in Table 3.4.

Table 3.4: Mapping Between Agent Service APIs and Networking APIs

Method	Agent Service Endpoint	Networking Public Endpoint	Usage
GET	:port/v2.0/networks	:9696/v2.0/networks	List networks
POST	:port/v2.0/networks	:9696/v2.0/networks	Create a network
GET	:port/v2.0/networks/{network_id}	:9696/v2.0/networks/{network_id}	Show network details
DELETE	:port/v2.0/networks/{network_id}	:9696/v2.0/networks/{network_id}	Delete network
GET	:port/v2.0/subnets	:9696/v2.0/subnets	List subnets
POST	:port/v2.0/subnets	:9696/v2.0/subnets	Create subnets
GET	:port/v2.0/subnets/{subnet_id}	:9696/v2.0/subnets/{subnet_id}	Show subnet details
DELETE	:port/v2.0/subnets/{subnet_id}	:9696/v2.0/subnets/{subnet_id}	Delete subnet
GET	:port/v2.0/ports	:9696/v2.0/ports	List ports
POST	:port/v2.0/ports	:9696/v2.0/ports	Create ports
GET	:port/v2.0/ports/{port_id}	:9696/v2.0/ports/{port_id}	Show port details
DELETE	:port/v2.0/ports/{port_id}	:9696/v2.0/ports/{port_id}	Delete port

OpenStack Compute API is an interface of Compute service that provides server

capacity in the cloud. Compute servers come in different flavors of memory, cores, disk space, and CPU, and can be provisioned in minutes. Interactions with Compute servers can happen programmatically with the OpenStack Compute API. Table 3.5 shows APIs mapping between Agent service and OpenStack Compute service.

Table 3.5: Mapping Between Agent Service APIs and Compute APIs

Method	Agent Service Endpoint	Networking Public Endpoint	Usage
GET	:port/v2.1/{tenant_id}/servers	:8774/v2.1/{tenant_id}/servers	List servers
POST	:port/v2.1/{tenant_id}/servers	:8774/v2.1/{tenant_id}/servers	Create a server
GET	:port/v2.1/{tenant_id}/servers/detail	:8774/v2.1/{tenant_id}/servers/detail	List all servers with details
GET	:port/v2.1/{tenant_id}/servers/{server_id}	:8774/v2.1/{tenant_id}/servers/{server_id}	Show details for a server
DELETE	:port/v2.1/{tenant_id}/servers/{server_id}	:8774/v2.1/{tenant_id}/servers/{server_id}	Delete a server
GET	:port/v2.1/{tenant_id}/flavors	:8774/v2.1/{tenant_id}/flavors	List flavors
POST	:port/v2.1/{tenant_id}/flavors	:8774/v2.1/{tenant_id}/flavors	Create a flavor
GET	:port/v2.1/{tenant_id}/flavors/detail	:8774/v2.1/{tenant_id}/flavors/detail	List all flavors with details
GET	:port/v2.1/{tenant_id}/flavors/{flavor_id}	:8774/v2.1/{tenant_id}/flavors/{flavor_id}	Show details for a flavor
DELETE	:port/v2.1/{tenant_id}/flavors/{flavor_id}	:8774/v2.1/{tenant_id}/flavors/{flavor_id}	Delete a flavor
GET	:port/v2.1/{tenant_id}/os-hypervisors	:8774/v2.1/{tenant_id}/os-hypervisors	List hypervisors

Proxy components work as message brokers. They are responsible for mapping Agent API endpoint to OpenStack service endpoint, forwarding mapped request to OpenStack cloud and delivering response sent from OpenStack cloud to user via agent's HTTP service. In the case of receiving a request of GET method, involved proxy component map Agent API endpoint to OpenStack service endpoint by looking up cloud IP address stored in agent database and appending appropriate port number and URL suffix. Then the proxy component forwards the user request with user's token to mapped OpenStack cloud endpoint and delivers response sent by the cloud to the user via agent's HTTP service. A process of handling a request of GET method by the proxy component is shown in WorkFlow 1. Handling DELETE request is similar to handling GET request. WorkFlow 2 shows the process of handling a request of DELETE method. In the case of handling a request of POST method, there is one more step than then previous workflow. WorkFlow 3 shows the process of handling a request of POST method.

Workflow 1 Handling Request of GET Method

1: Query cloud IP address in agent's local database
2: Map to cloud service endpoint by appending appropriate service port number and URL suffix to retrieved IP address
3: Forward user request with user's token to mapped cloud service endpoint
4: Wait for response from cloud
5: Modify response received from cloud and deliver to user

Workflow 2 Handling Request of DELETE Method

1: Query cloud IP address in agent's local database
2: Map to cloud service endpoint by appending appropriate service port number and URL suffix to retrieved IP address
3: Forward user request with user's token to mapped cloud service endpoint
4: Wait for response from cloud
5: If request is successfully operated by cloud, then update agent's local database
6: Modify response received from cloud and deliver to user

Workflow 3 Handling Request of POST Method

1: Retrieve cloud's virtual resource consumption information
2: According to cloud selection strategy, choose a cloud to allocate the virtual resource.
3: Map to cloud service endpoint by appending appropriate service port number and URL suffix to the chosen cloud's IP address
4: Forward user request with user's token to mapped cloud service endpoint
5: Wait for response from cloud
6: If request is successfully operated by cloud, then update agent's local database
7: Modify response received from cloud and deliver to user

3.4 Implementation of CYCLON protocol

In CYCLON protocol[13], each peer is aware of a small continuously changing set of other peers, named its neighbors. More formally, each peer maintains a neighbor list in a fixed-size, small cache of c entries (with typical value 10, 50, or 100). Each peer repeatedly initiates a neighbor exchange operation, called shuffle, with a subset of l neighbors (1 <= l <= c) from its neighbor list, where l is a parameter, known as shuffle length. In this solution, agent runs CYCLON protocol as a standalone process. Neighbor list of an agent is stored in Memcached [21]. Memcached comes with OpenStack installation and it's a high-performance, distributed memory object caching system which is capable of key-value store for small chunks of arbitrary data. A neighbor data object has a key-value pair, key is neighbor's IP address and value is age which indicates the age of the neighbor since the moment it is created on an agent's neighbor list. Agent periodically initiates neighbor exchanges with its neighbors at a fixed period T. The shuffling operation of the initiating agent P is shown in Workflow 4.

Algorithm 4 Shuffling

1: Increase the age of all neighbors on P's neighbor list by one
2: Choose a subset containing a neighbor Q with the highest age among all neighbors on P's neighbor list, and l-1 other random neighbors.
3: Replace Q's entry with a new entry of age 0 and with P's IP address
4: Send the updated subset to agent Q
5: Receive a subset from Q and remove Q's entry on P's neighbor list
6: Discard entries pointing at P and entries already contained on P's neighbor list
7: Update P's neighbor list to include all remaining entries, by firstly using empty neighbor slot (if any), and secondly replacing entries among the ones sent to Q

On reception of a shuffling request, agent Q responds by sending back a random subset of at most l of its neighbors, and update its own neighbor list based on received entries. The receiving agent Q does not increase any neighbor's age until it is its turn to initiate a shuffle. In the case of adding nodes and removing nodes in this P2P system, the implement is strictly based on CYCLON protocol.

Since agent is able to actively initiate a shuffling operation and handle a shuffling operation initiated by other agent concurrently. As well as, Memcached is frequently queried during the shuffling period, thence lock mechanism is used to avoid write-read conflict. Lock is acquired right after agent initiate a round of shuffling, more precisely, the lock is acquired before step 1 in Workflow 4, and released after the shuffling operation is done. Lock mechanism brings a possibility of causing dead lock, for instance, agent P shuffles with agent Q, agent Q shuffles with agent L, while agent L shuffles with agent P. In order to avoid dead lock, a fixed waiting time is set between step 4 and step 5 in Workflow 4. If the waiting time is exceeded, then the agent release its lock and interrupt shuffling operation.

3.5 Cloud Selection Scheduler

As a proof-of-concept, agent only supports creating virtual resources of images and VMs. When agent receives a request of creating an image or booting a VM, it has to decide where to create the requested virtual resource among those OpenStack cloud instances on its neighbor list. Therefore, cloud selection scheduler is essential to be implemented at agent to improve resource utilization. A cloud selection scheduler consists of two parts: filter and weigher. To prevent resource wastage either from excessive occupation or through idling, cloud selection scheduler must predict a virtual resource's consumption and choose a suitable OpenStack cloud instance before the virtual resource is created. The cloud selection scheduler is in charge of scheduling decisions and taking into consideration of available OpenStack cloud instances which may be characterized by different resource capacities and features. When a virtual resource creation request is received by agent, filter is applied to determine if an OpenStack cloud instance has the adequate free capacity to meet the requested resource parameters. After filtering, weigher is applied to score all the filtered OpenStack cloud instances to pick the best one to create requested virtual resource.

Filters and weigher are designed and implemented as pluggable modules on agent. As a proof-of-concept, agent currently has two filters and three weighers. More cloud selection strategies can be complemented in the future. These two filters are categorized as follows:

- Resource-based filter: Retrieve information of resource capacity from all OpenStack cloud instances which are on agent's neighbor list and filter cloud instances which have an adequate free capacity to meet required resource parameter regarding memory, disk, CPU cores, and so on.
- Randomness-based filter: Filter OpenStack cloud instances based on a random two choices function[14]. The random function simply picks two OpenStack cloud instances from agent's neighbor list. According to [14], pick two at random has a signification improvement in system performance.

After filtering, the scheduler weighs available OpenStack cloud instances by applying weighers and select the best rated cloud instance to create virtual resource. In this implement, there are three weighers named disk-based weigher, memory-based weigher and image-based weigher separately. They are shown as follows:

- Disk-based weigher: Rates a cloud instance higher according to free disk. The more free disk a cloud instance has, the higher is its rate.
- Memory-based weigher: Scores a cloud instance higher based on available memory. A higher rated cloud instance has more free memory.
- Image-based weigher: Rates an OpenStack cloud instance relatively higher scores which have the required image in the case of booting a VM.

Weighers are used in different cases and multiple weighers can be used together

while weighting. In case of creating an image, the disk-based weigher is prior to be used. While in case of booting a VM, the memory-based weigher and the image-based weigher are supposed to be applied. Workflow5 shows a combination usage of randomness-based filter, memory-based weigher and image-based weigher in case of booting a VM.

Workflow 5 Combination Usage of Randomness-based Filter, Memory-based Weigher and Image-based Weigher

1: If agent's neighbor list is empty, then boot a VM at its affiliated cloud instance.
2: If there is only one neighbor on agent's neighbor list, then apply memory-based weigher and image-based weigher among the neighbor and agent's affiliated cloud instance. The one with higher score is chosen to boot VM.
3: If more than two neighbors are on agent's neighbor list, first filter neighbors and affiliated cloud instances then apply memory-based weigher and image-based weigher among the two filtered cloud instances. The one with a higher score is chosen to boot VM.

4

Evaluation and Result

In this chapter, the evaluation results and comparison of the proposed P2P system and standard OpenStack are presented and discussed. The experimental environment setup will be discussed followed by result discussion and analysis.

4.1 Evaluation Framework

In this evaluation, in order to generate simulated loads and retrieve results from simulations, Rally is chosen as the evaluation framework. Rally is a benchmarking tool for validating, performance testing and benchmarking OpenStack at scale [22]. Rally's benchmarking engine allows to write parameterized benchmark scenarios and automatically perform tests under simulated real user loads. Results of these tests and benchmarks, such as average/maximum response time and failure rate are presented by Rally in a human readable form.

Rally enables user to customize benchmark scenarios. Benchmark scenarios are what Rally actually uses to test the performance of an OpenStack deployment. Each benchmark scenario performs a small set of atomic operations, thus testing some simple use case, usually that of a specific OpenStack project. For example, the "create_image_and_boot_instances" benchmark scenario allows to benchmark the performance of a sequence of only several simple operations: it first creates tenants and users per tenant, then creates an image and boots instances (or VMs) per user (with customizable parameters), finally clean up resource created by this benchmark.

4.2 Experimental Environment Setup

The experimental environment is not setup on bare metal machines, it is setup on VMs of an OpenStack cluster instead which is hosted by 10 Ericsson's GEP5-64-1200 rack servers. Each GEP5-64-1200 consisted of 2 Intel Xeon E5-2658 v2 @ 2.40GHz processors, 64 GB of RAM, 10 x 400 GB Intel SSD DC S3700 disk drives. Experimental VMs are given various types of configurations in terms of the number of virtual CPUs (VCPUs), RAM. Table 4.1 shows four types of configuration. VMs of controller node and compute node are comprised of 2 VCPUs and 4 GB of RAM. VMs of Rally client node consisted of 1 VCPU and 2 GB of RAM. VMs of Keystone node type are given a relatively larger number of VCPUs and RAM to

Table 4.1: Four Types of Node Configuration

Configuration	Keystone Node	Controller Node	Compute Node	Rally Client Node
Number of VCPUs	8	2	2	1
RAM	16 GB	4GB	4GB	2GB
Disk	160 GB	10GB	10GB	10GB
Operating System	Ubuntu LTS 14.04	Ubuntu LTS 14.04	Ubuntu LTS 14.04	Ubuntu LTS 14.04

reduce effects of Keystone to system performance. Each node has different OpenStack services running, Table 4.2 shows services run on Keystone node, controller node and compute node. Number of VMs we run in experiments based on testing scenarios. The OpenStack version chosen in this evaluation is Liberty. The CYCLON protocol parameters are set as follows. The length of the neighbor list is c = 4, shuffle length is l = 2, and fixed shuffling interval is T = 30s. In addition, the cloud selection scheduler applies a randomness-based filter for filtering. In the case of creating an image, disk-based weigher is used. Whereas in the case of booting a VM, memory-based weigher and image-based weigher are used.

Table 4.2: Services Run on Keystone Node, Controller Node and Compute Node of P2P System

Type of Node	Service
Keystone Node	mysql, rabbitmq, keystone
Controller Node	agent, rabbitmq, mysql, keystone, nova-api, nova-certificate, nova-object-store, nova-conductor, nova-scheduler, nova-certificate-authentication, neutron, glance-api, glance-registry
Compute Node	nova-compute, nova-network

Rally benchmark scenario chosen in this evaluation is the one discussed before - "create_image_and_boot_instances" with customized parameters. Due to limited resources in terms of number of VCPUs, RAM and storage in this experimental environment, the image file used in this benchmark is an empty image file and using empty image file does not cause any error to the process of booting instances on OpenStack. In addition, the virtual hardware template for instances, called flavor, is created as follow: 1 VCPU, 1 MB of RAM and 10 GB of disk. In the chosen benchmark scenario, the number of tenants, users per tenant and instances per user all are one. In other words, the sequence of atomic operations of the chosen scenario is creating a tenant, creating a user of that tenant, creating an image and booting an instance.

4.2.1 Scenario One

In scenario one, two testbeds are set up for the purpose of comparison. One is for the proposed P2P system, one is for standard OpenStack. The performance of these two systems is aimed to be evaluated and compared in terms of a varied number of concurrent user loads against system (we use concurrency as alias in following text) and fixed system size.

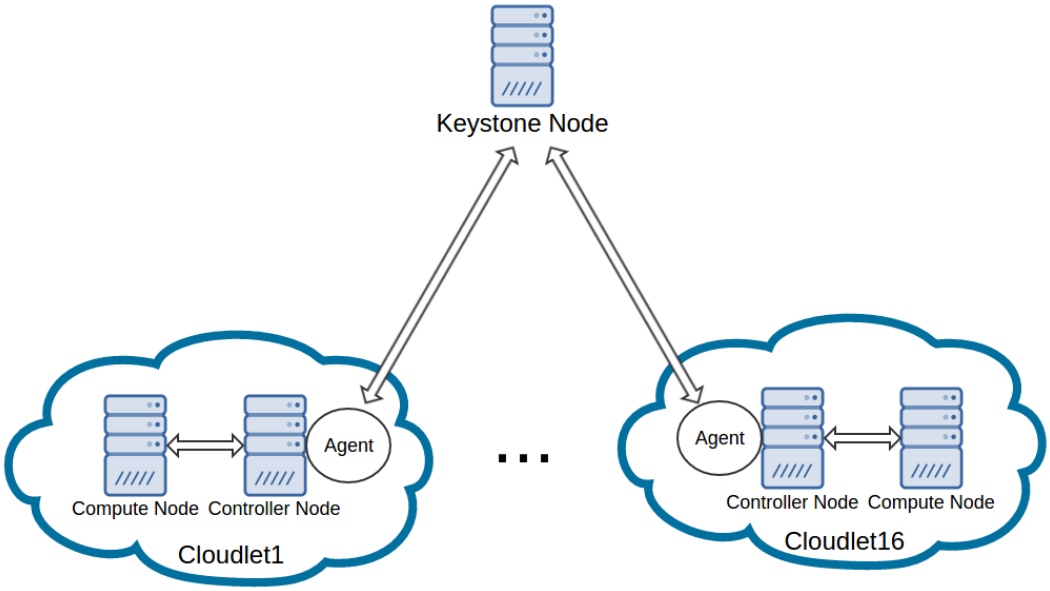


Figure 4.1: System Architecture of P2P testbed

Figure 4.1 shows system architecture of the P2P testbed which is comprised of one centralized Keystone node and sixteen cloudlets, each cloudlet consists of one controller node and one compute node. Besides, each cloudlet runs one agent on the controller node. All the cloudlets share identity service provided by the Keystone node. Besides, Rally client nodes are created in the case of different concurrency and each Rally client is on pair with one agent. Rally clients concurrently send requests to agents under the "create_image_and_boot_instances" benchmark scenario. From an overall system perspective, these concurrent requests are uniformly distributed to sixteen agents. For instance, in the case of concurrency 16, sixteen Rally clients send requests to the P2P system, more precisely, sixteen Rally clients send requests to sixteen agents and each Rally client executes the benchmark scenario and only send requests to the agent that they are on pair with.

Figure 4.2 shows system architecture of the standard OpenStack testbed which consists of one centralized Keystone node, one cloudlet which consists of one controller node and sixteen compute nodes. The Keystone node provides identity service for this testbed and the sixteen compute nodes are connected to the controller node to provide nova-compute service. Rally client nodes are also created in case of different concurrency and all the Rally clients are bonded to the controller node of the standard OpenStack system. In other words, all Rally clients execute the benchmark scenario and send requests to the controller node concurrently. For instance, in the case of concurrency 16, sixteen Rally clients send requests to the standard OpenStack system, more precisely, sixteen Rally clients concurrently execute the benchmark scenario and send requests to the controller node.

In every case of varied concurrency, the benchmark is executed repeatedly 25 times on both the P2P system and the standard OpenStack system.

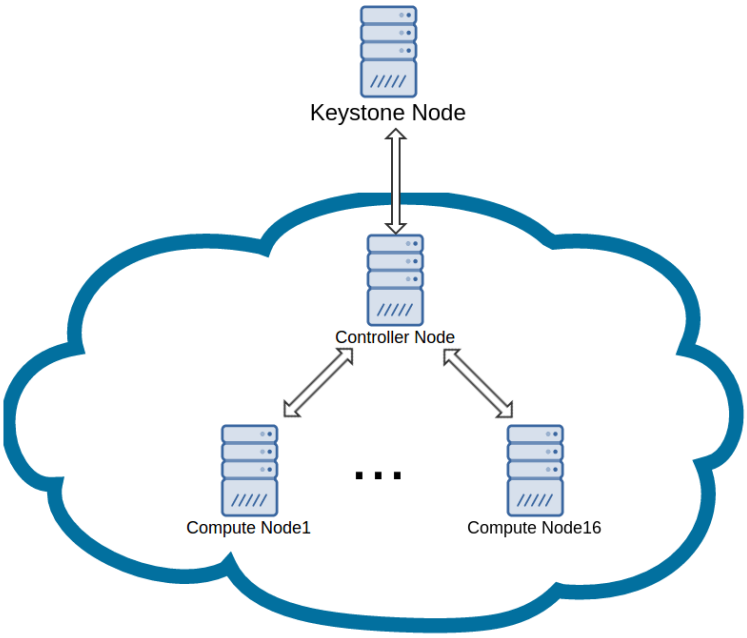


Figure 4.2: System Architecture of Standard testbed

4.2.2 Scenario Two

In scenario two, one P2P testbed is set up at a size of one Keystone node, varied number of cloudlets. Precisely, each cloudlet is comprised of one controller node, one compute node and one agent running on the controller node. The P2P testbed is essentially the same as the P2P testbed in scenario one, while system performance in scenario two is aimed to be evaluated regarding varied system size and fixed concurrency. The fixed concurrency is set as 16 and system size is scaled from 1 up to 16. In addition, Rally client nodes are created in the case of different system size and each Rally client is on pair with one agent. The benchmark in scenario two is the same as the one in scenario one. For every system size, the benchmark is repeatedly executed 25 times on the P2P testbed.

4.3 Experimental Results and Analysis

This section shows experimental results and analysis of scenario one and scenario two separately.

4.3.1 Scenario One

Evaluation results of scenario one are presented and discussed in this subsection regarding response time, failure rate, CPU usage and distribution of VMs.

4.3.1.1 Response Time

Table 4.3 shows the response time of the P2P system with 16 Cloudlets handling Rally benchmark of creating an image and booting a VM in the case of various concurrency. The confidence interval for average response time is 95%. The total time of creating an image and booting a VM increases quickly when concurrency is more than 16. Figure 4.3 shows plotted result of this measurement. As concurrency increases, the response time of creating an image, booting a VM or total grow linearly. Specifically, the response time of creating an image is getting considerably longer than the response time of booting a VM as concurrency increases. When concurrency is larger than 8, the response time of creating an image is always longer than the response time of booting a VM.

Table 4.3: Create An Image and Boot A VM (P2P System with 16 Cloudlets)

Concurrency	Create An Image			Boot A VM			Total		
	Standard Deviation	Margin of Error	Average	Standard Deviation	Margin of Error	Average	Standard Deviation	Margin of Error	Average
1	0.139	±0.055	3.227	1.023	±0.401	4.489	1.031	±0.406	7.716
2	0.130	±0.026	3.576	1.123	±0.220	4.149	1.144	±0.224	7.725
4	0.186	±0.036	4.078	1.148	±0.225	4.206	1.181	±0.232	8.284
8	0.356	±0.049	4.752	1.307	±0.181	4.592	1.384	±0.192	9.345
16	1.251	±0.123	6.020	1.644	±0.161	5.244	2.404	±0.236	11.264
32	5.039	±0.349	15.036	4.715	±0.327	13.244	7.578	±0.525	28.280
64	11.831	±0.583	33.059	8.027	±0.395	22.206	17.073	±0.841	55.264
128	29.739	±1.032	76.199	19.676	±0.683	39.679	45.304	±1.572	115.878

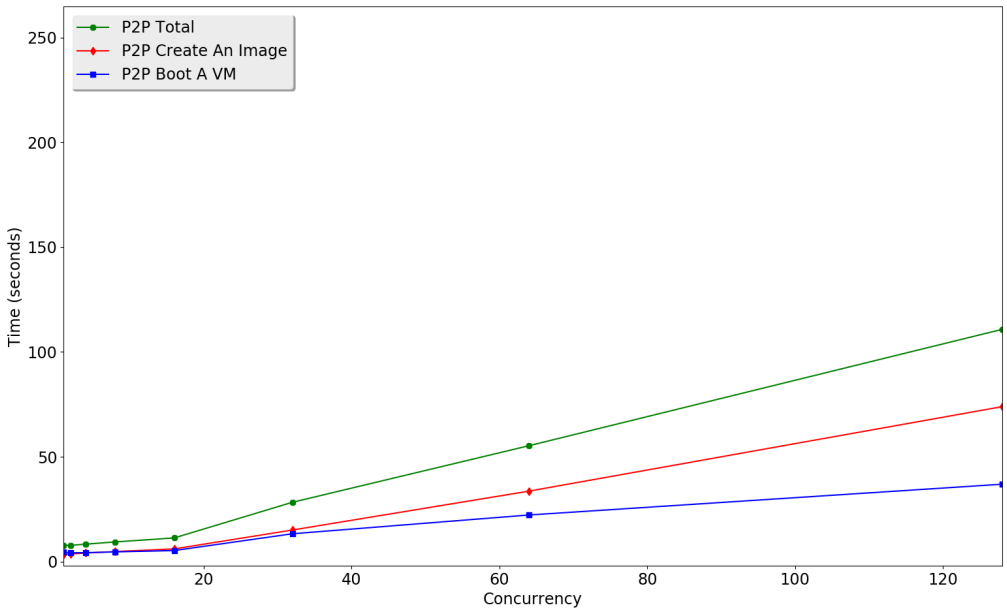


Figure 4.3: Create An Image and Boot A VM (P2P with 16 Cloudlets)

Table 4.4 shows evaluation result of the standard OpenStack with 16 compute nodes handling Rally benchmark of creating an image and booting a VM in the case of various concurrency. The confidence interval for average response time is 95%. The total time of creating an image and booting a VM increases sharply when concurrency is 8. Figure 4.4 shows plotted result of this measurement. The response time of booting a VM is always longer than the response time of creating an image.

Table 4.4: Create An Image and Boot A VM (Standard OpenStack with 16 Compute Nodes)

Concurrency	Create An Image			Boot A VM			Total		
	Standard Deviation	Margin of Error	Average	Standard Deviation	Margin of Error	Average	Standard Deviation	Margin of Error	Average
1	0.085	±0.033	2.707	0.204	±0.080	4.122	0.234	±0.091	6.829
2	0.116	±0.032	2.992	0.405	±0.112	4.553	0.445	±0.123	7.545
4	0.319	±0.062	3.148	0.453	±0.089	6.142	0.587	±0.115	9.289
8	1.008	±0.139	4.825	1.131	±0.157	11.132	1.600	±0.222	15.957
16	1.353	±0.133	10.933	5.414	±0.531	24.103	6.171	±0.605	35.036
32	6.044	±0.419	27.843	17.504	±1.213	49.989	20.722	±1.436	76.832
64	8.565	±0.422	58.708	31.539	±1.554	86.636	38.158	±1.881	145.344
128	28.279	±1.188	113.696	62.687	±2.632	149.202	81.488	±3.422	262.898

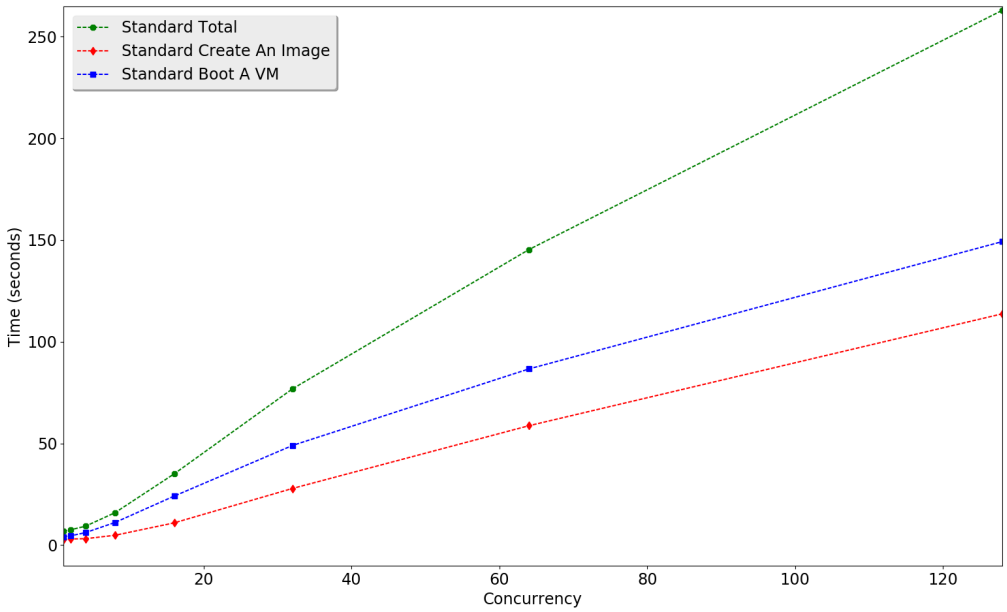


Figure 4.4: Create An Image and Boot A VM (Standard with 16 Compute Nodes)

Figure 4.5 shows a comparison of the response time of the P2P system with 16 cloudlets and standard OpenStack with 16 compute nodes handling Rally benchmark of creating an image and booting a VM in terms of a various number of concurrency. From Figure 4.5, it is obvious that the P2P system performs better

than the standard OpenStack system as concurrency increases. More precisely, for the standard OpenStack system, it takes relatively more time handle the whole benchmark than the P2P system when concurrency grows. Figure 4.5 also shows that for the standard OpenStack system, when concurrency increases, the performance of booting a VM markedly affect the result of total response time. However, for the P2P system, the response time of booting a VM rises steadily compared to the standard OpenStack system, the response time of creating an image mainly affects the result of total response time instead.

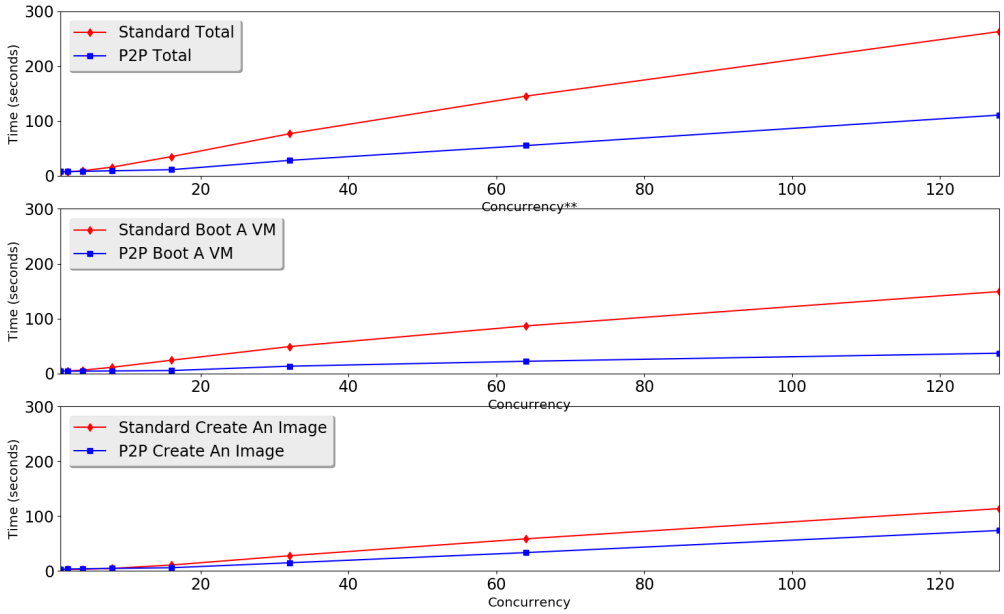


Figure 4.5: Comparison of P2P System and Standard OpenStack Handling Benchmark of Creating An Image and Booting A VM

4.3.1.2 Failure Rate

Figure 4.6 shows the failure rate of the P2P system and standard OpenStack handling Rally benchmark of creating an image and booting a VM in the case of different concurrency. The failure rate is defined as if the process of booting an image and creating a VM can be successfully handled by the system. When concurrency is less than 64, the failure rate of both systems is 0%. However, both systems start to drop requests when concurrency is 64. When concurrency is up to 128, the failure rate of the standard OpenStack system is significantly higher than the failure rate of the P2P system.

4.3.1.3 CPU Usage

In this measurement, CPU usage of a controller node is periodically recorded every 3 seconds. Figure 4.7 demonstrates a comparison of CPU usage of controller node

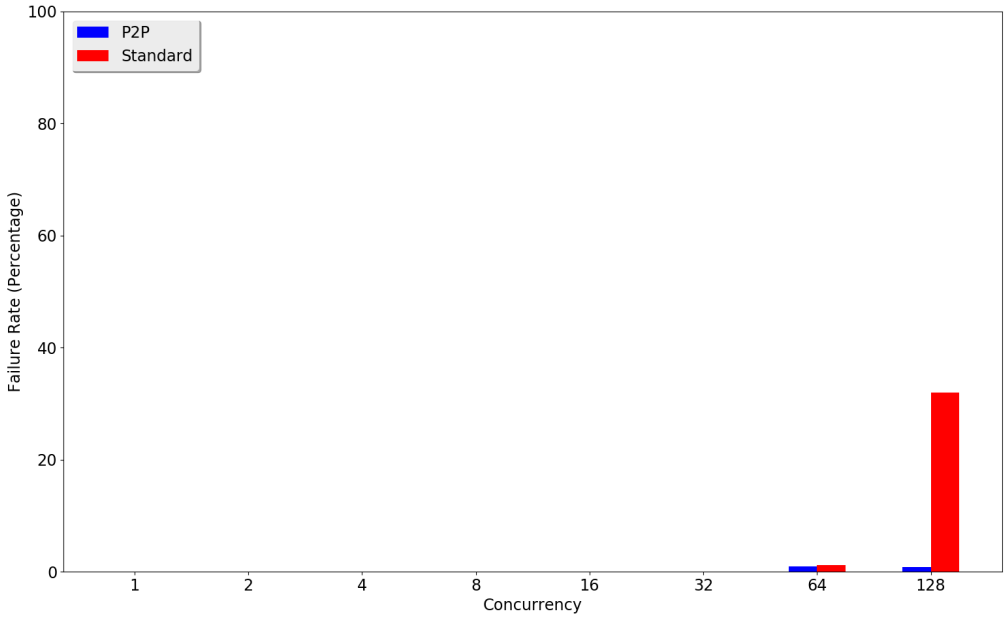


Figure 4.6: Failure Rate of P2P System and Standard OpenStack Handling Benchmark of Creating An Image and Booting A VM

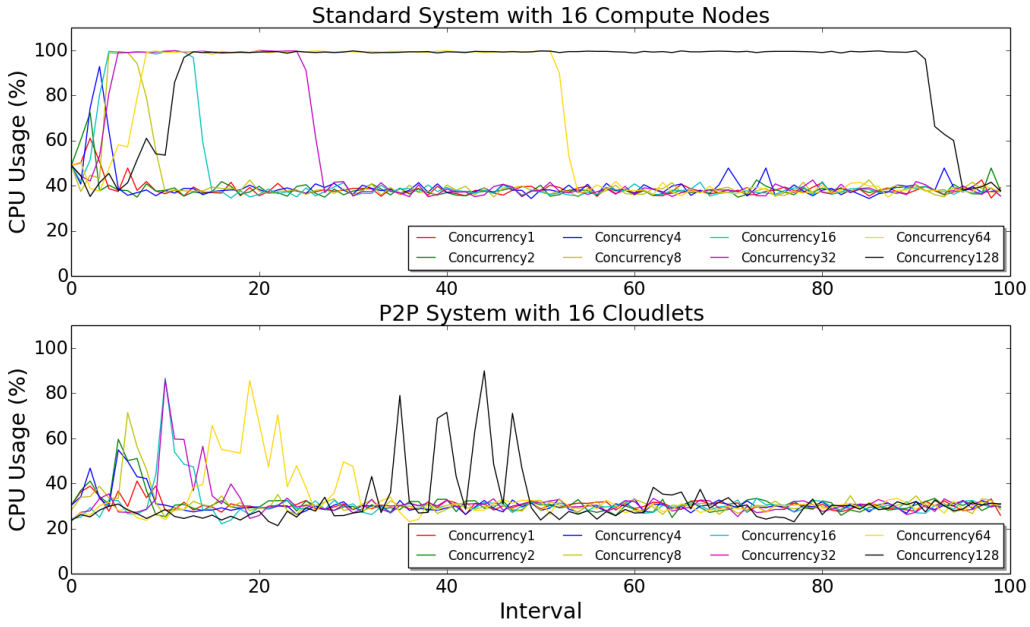


Figure 4.7: Comparison of CPU Usage of P2P Testbed and Standard OpenStack Testbed in Case of Various Concurrency

of the P2P system and the standard OpenStack system handling Rally benchmark of creating an image and booting a VM in the case of various concurrency. As for the P2P system, the controller node whose affiliated cloudlet is most frequently chosen to boot VMs at one round of execution is picked to represent CPU usage. From Figure 4.7, it is clear that CPU usages of the controller node of the standard OpenStack system in most cases reach 100% and stay around over a period. As for the P2P system, CPU usages of a controller node never reaches 100%, instead, rarely goes above 80% and does not stay around at a high percentage.

Figure 4.8 shows a comparison of CPU usage of controller node of the P2P system and the standard OpenStack system handling Rally benchmark of creating an image and booting a VM when concurrency is 128. As for the standard OpenStack system, it is obvious that CPU usage of the controller node quickly goes up to 100% and constantly remains over a long period until the benchmark is finished. The figure at bottom illustrates CPU usages of all sixteen controller nodes of the P2P system handling benchmark of creating an image and booting a VM when concurrency is 128. Although CPU usages of controller nodes in the P2P system also quickly goes up, they do not peak at close to 100% and stay around at a high percentage over a long period.

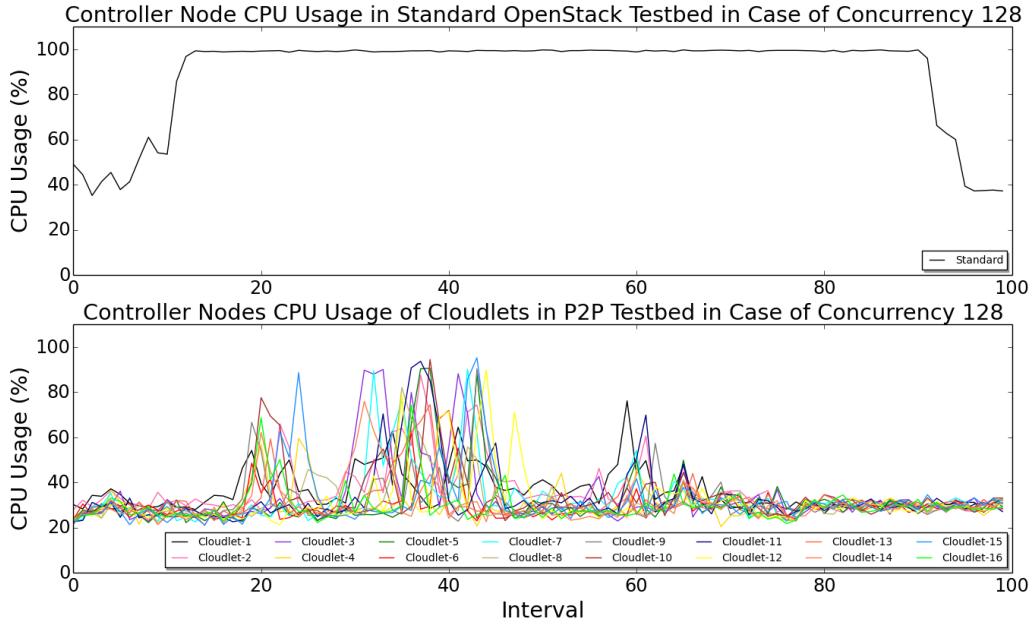


Figure 4.8: Comparison of CPU Usage of Controller Nodes in P2P Testbed and Standard OpenStack Testbed in Case of Concurrency 128

4.3.1.4 Distribution of VMs

Figure 4.9 shows the distribution of VMs in the P2P system over 25 times of execution of handling benchmark of creating an image and booting a VM in cases of

concurrency is 16, 32 and 64. It can be seen that VMs are fairly distributed to various OpenStack cloudlets. There is no striking diversity of VM distribution among all the OpenStack cloudlets.

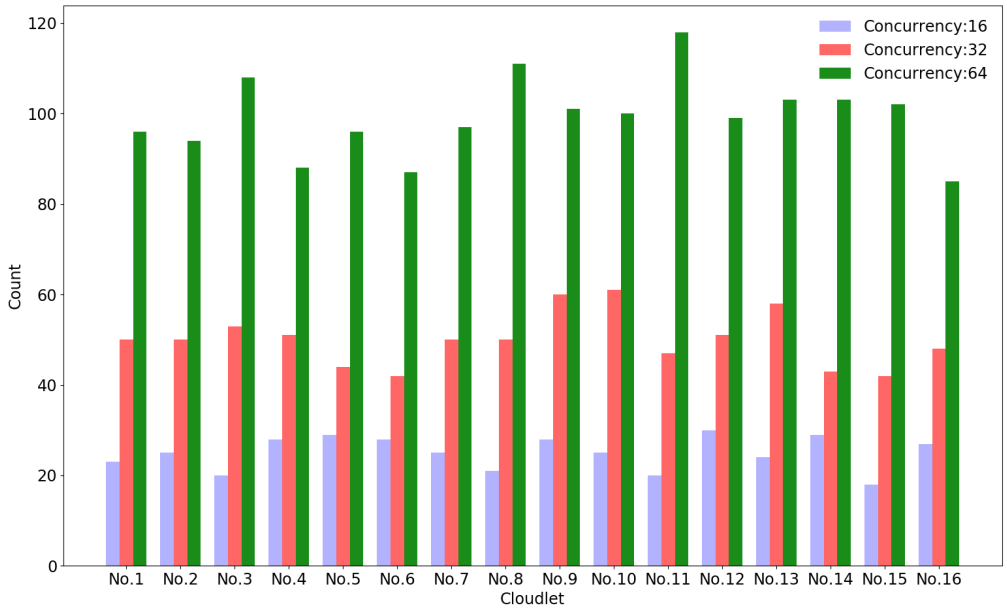


Figure 4.9: Distribution of VMs over 25 Times of Handling Benchmark of Creating An Image and Booting A VM

4.3.2 Scenario Two

Evaluation results of scenario two are presented and discussed in this subsection in terms of response time and CPU usage.

4.3.2.1 Response Time

Table 4.5 shows the response time of the P2P system at various scales handling Rally benchmark of creating an image and booting a VM when concurrency is 16. The confidence interval for average response time is 95%. Figure 4.10 shows plotted results of this evaluation. It is obvious that the response time of booting a VM drops significantly when system size scales from one to two which leads to a shorter response time of finishing the whole benchmark scenario. As system size scales from two to sixteen, the response time of booting a VM decreases steadily and response time of creating an image fluctuates slightly.

4.3.2.2 CPU Usage

In this measurement, CPU usage of a controller node is also periodically recorded every 3 seconds. Figure 4.11 demonstrates the comparison of CPU usage of controller

Table 4.5: Create An Image and Boot A VM (P2P System with Varied System Size in Case of Concurrency 16)

System Size	Create An Image			Boot A VM			Total		
	Standard Deviation	Margin of Error	Average	Standard Deviation	Margin of Error	Average	Standard Deviation	Margin of Error	Average
1 Cloulet	2.274	±0.322	8.02	6.06	±0.857	20.714	8.022	±1.135	28.735
2 Cloudlets	1.509	±0.148	6.096	6.016	±0.598	10.069	6.716	±0.658	16.165
4 Cloudlets	1.719	±0.169	7.572	4.046	±0.397	7.416	4.739	±0.464	14.988
8 Cloudlets	1.077	±0.106	7.349	2.238	±0.233	5.883	2.612	±0.256	13.232
16 Cloudlets	1.251	±0.123	6.020	1.644	±0.161	5.244	2.404	±0.236	11.264

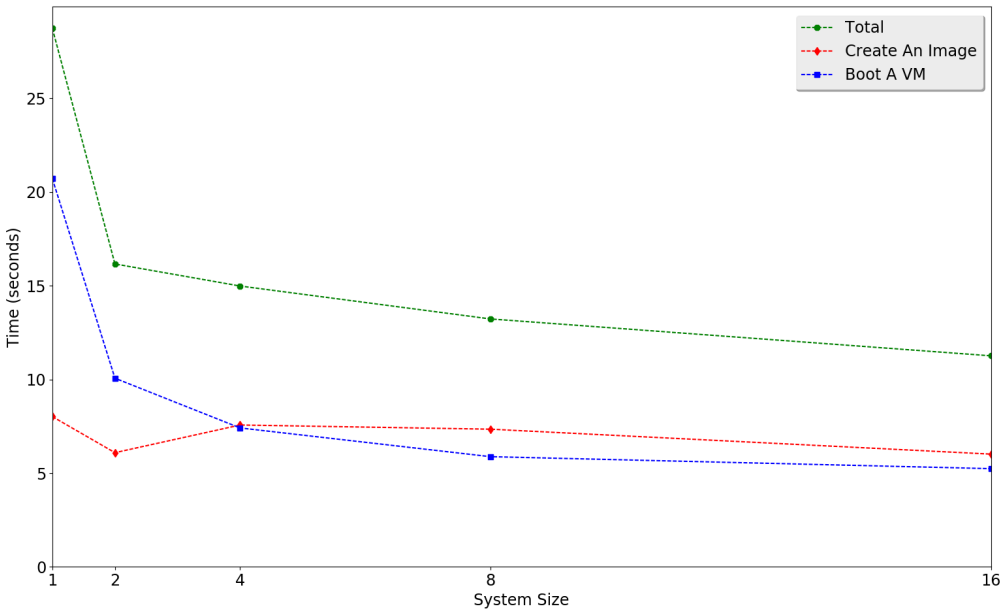


Figure 4.10: Response Time in Case of Fixed Concurrency and Varied System Size

node of the P2P system handling Rally benchmark of creating an image and booting a VM in cases of varied system size and concurrency is 16. At one round of execution, the controller node whose affiliated cloudlet is most frequently chosen to boot VMs is picked to represent CPU usage. From Figure 4.11, it can be seen that only when system size is one, CPU usage of controller node goes up to 100% and stay around for a while then starts fluctuating till benchmark is finished. When system size is 2, 4, 8 or 16, CPU usage of chosen controller node just peaks at one point but never stay around for a period. When system size is one, the system takes a longer time to handle benchmark of creating an image and booting a VM, as the CPU of controller node is overloaded. Overloaded CPU essentially affects system performance by taking longer execution time.

From the above measurements, by scaling cloud instances through agents, the pro-

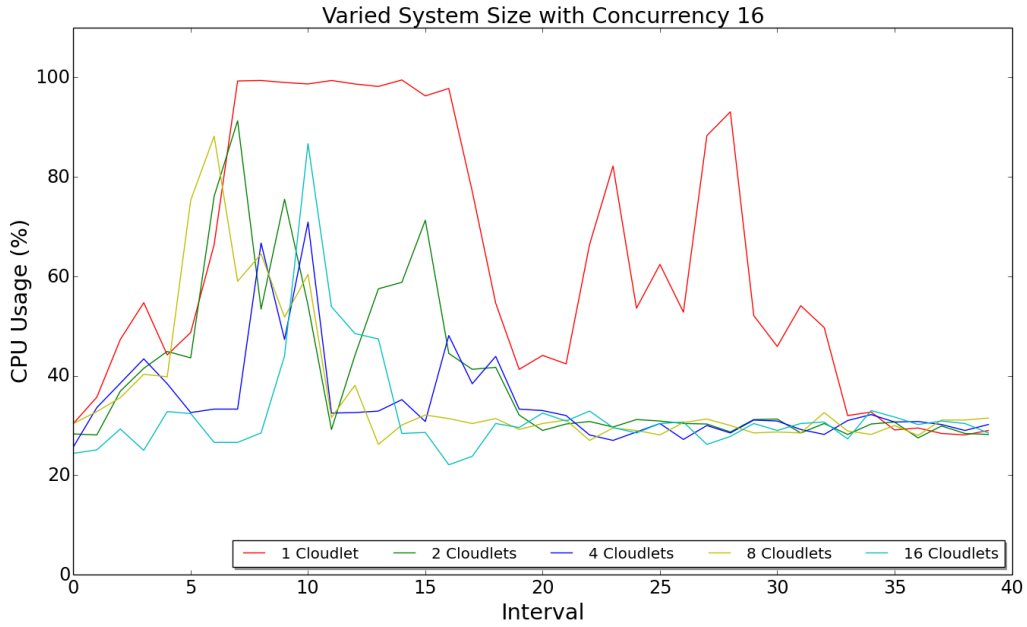


Figure 4.11: CPU Usage in Cases of Varied System Size and Fixed Number of Concurrency

posed P2P solution is feasible to create virtual resource across OpenStack cloud instances and shows significantly lower response time when handles concurrent requests by executing request at each OpenStack cloud instance separately and avoiding CPU on controller node being overloaded. Moreover, compared to standard OpenStack deployment, the P2P solution is more failure-resistant and efficient in terms of creating an image and booting a VM. Without considering affection of identity service - Keystone, the P2P solution has no limitation on the number of cloud instances and its performance improves with the increasing number of cloud instances.

5
Future Work

This thesis report has discussed design and implement of a P2P solution for scaling OpenStack cloud instances. Although VMs owned by one project are feasible to be created on distributed OpenStack cloud instances, these VMs can not functionally communicate with each other. Besides, the agent currently only support proxying OpenStack's services such as compute service, network service, image service and identity service. An important next step will be to enrich agent's functionality by implementing more proxies for other OpenStack services. Also, the current solution is not fully distributed, as cloud instances in this P2P network share a centralized Keystone node which provides identity service. The Keystone node would be a potential bottleneck, if the whole P2P system is going to scale massively. Therefore, one important subject of future work will be the replacement of the centralized identity service with a distributed identity service. We envisage this replacement will lead to a better system reliability and performance. Moreover, in the cloud selection scheduler of our prototype, we kept strategies as simple as possible. Another important subject will be implementing more filters and weighers based on optimized algorithms to improve virtual resource utilization. We believe that optimized cloud selection scheduler will bring on better utilization of virtual resources and improved system performance regarding response duration.

6
Related Work

The problem of scaling cloud instances has been studied for the past decade in both industry and academic area. This chapter discusses work related to our study.

Cells is a built-in functionality of Nova which could scale an OpenStack cloud in a more distributed fashion. When this functionality is enabled, compute nodes in an OpenStack cloud are partitioned into groups called Cells [23]. Cells essentially provides the means to create logical fences around OpenStack resources such as compute nodes. However, Cells is still considered experimental and it's not an option for Ericsson, since Ericsson desires a solution which supports as many OpenStack default features as possible, such as Security Group, Availability Zones and Server Groups in which Cells does not support.

Besides Cells, OpenStack cascading solution is another concept proposed by Huawei [24]. Huawei's solution is to map the underlying OpenStack to a compute node and use a parent OpenStack to orchestrate child OpenStack cloud instances. Huawei's idea is innovative. However, the cascading solution works like fractal [25], it scales in a hierarchical structure. Due to its attribute, it would be complex and expensive to implement and manage when the whole cloud system is growing larger. Another drawback of the cascading solution is that, although the child OpenStack cloud instances are still accessible even if the parent OpenStack cloud instance is down, the consistency will be lost between the parent instance and child instances once the parent instance is up again.

Brasileiro et al.[26] present a middleware, called Fogbow, designed to support federations of independent IaaS cloud vendors. Fogbow consists of three main components: membership manager, allocation manager and messaging service. The membership manager is for member discover. The allocation manager runs at a given cloud and operates actions to its affiliated cloud. The messaging service runs at a particular site and allocation managers and membership managers communicate through this messaging service. However, no redundancy support for the messaging service which could be a single-point failure when the federated system scales up.

In [27], Buyya et al. propose InterCloud framework in which the federated network of clouds is interceded by a Cloud Exchange. Every cloud runs a Cloud Coordinator which is responsible for publishing offers according to the service the cloud provides. The Cloud Brokers requests the required capacity and bids on the offers. End users are associated with a cloud instance, which is responsible for fulfilling user demands.

However, the Cloud Exchange is a potential single-point failure and it leaves each cloud instance responsible for user identification.

In [28], Celesti et al. introduce a horizontal cross-federation solution based on the Cross-Cloud Federation Manager which is a placeable component in cloud infrastructure and consists of three sub-components: discovery agent, match-making agent and authentication agent. The discovery agent is responsible for discovering all available foreign clouds, the match-marking agent manages the process of choosing the best cloud instance to deploy virtual resources and the authentication agent takes duties to build security context among federated clouds. However, system performance evaluation of this solution is not presented and limitation of sharing resources on cloud owner's demand is still a challenge behind this solution.

[29] and [30] propose the Reservoir virtualization architecture which consists of two main parts: service provider and infrastructure provider. The service provider lease virtual resources from infrastructure provider and matches user needs by finding resources that the user's application requires. The infrastructure provider manages physical infrastructure and offers pool of resources such as computing capacity, network and storage to service providers. Resources on an infrastructure site are virtualized and partitioned into virtual execution environments (VEEs). A service application is capable of using several VEEs across infrastructure providers. However, transfer tasks from one provider to another is still a challenge for Reservoir.

The Contrail project [31] is build on the results of Reservoir project by adding vertical integration of PaaS and Iaas models. It provides users with a single access point to resources belonging to various cloud providers. Contrail acts as a broker between users and cloud provider and it is composed of three layers: interface layer, core layer and adapters layer. The interface layer exposes a way to interact with the federated systems via CLI and HTTP interfaces. The core layer contains modules that are responsible for identity management, application deployment and Service Level Agreement (SLA) coordination. The adapters layer enables access to infrastructure providers. Whereas, the single access point of Contrail is a potential single-point failure.

A layered service model of SaaS, PaaS and IaaS is proposed in [32]. The inter-Cloud federation is implemented at every service layer and delegated by broker specific to the entities at that layer. The top SaaS layer handles requirements of executing applications and maps performance metrics of applications to resources at the PaaS layer. The PaaS layer represents as a bridge between applications requirements and infrastructure resources. The IaaS layer provides resources such as computing capacity, storage and network to layers above it. Information flow between these three layers are delegated and translated by brokers. However, the layered model brings complexity to the case of scaling horizontally at each layer.

Compared to Huawei's Cascading solution[24] and the layered service model[32], our proposed solution scales horizontally, no hierarchical relation between cloud in-

stances and system architecture is flat, by which the solution enables cloud instances to easily join or leave without affecting proper functioning. Different from [26] and [31], the centralized Keystone component in our solution is feasible to be redundant. In contrast to [27], user identification is handled by the dedicated component Keystone instead of by each cloud instance in our solution. In [28], limiting sharing resources on cloud owner's demand is a challenge for the cross-federation solution, conversely, our solution enables cloud owner to elastically share resources. In [29] and [30], the Reservoir has a challenge of transferring tasks from one cloud instance to another, however, in our solution, tasks of uploading an image or creating a VM is able to be transferred across cloud instances.

7

Discussion and Conclusion

7.1 Societal and ecological contribution

Cloud computing is bringing benefits to our lives in many ways. It is a fundamental infrastructure to cutting-edge technologies in areas of Internet of Things, autonomous driving, artificial intelligence and others. Besides, cloud providers, such as Amazon's AWS, Microsoft's Azure and Google cloud platform, are becoming more and more popular with startup companies or small firms as the first choice to deploy their online services. From a societal aspect, by our solution, geographically distributed cloud instances are possible to be federated as one and the federated P2P system is able to be more economic and serve more users with lower cost on bare metals. From an ecological perspective, compared to standard OpenStack deployment, our P2P solution needs less machines but behaves more efficient which leads to produce less carbon dioxide, consumes less electricity.

7.2 Conclusion

We have presented a solution and its prototype for scaling OpenStack cloud instances by using P2P technologies. The P2P solution is motivated by a centralized message queue of a standard OpenStack cloud instance which causes scalability issue when a number of compute nodes at the cloud instance reaches its limitation. Instead of solving this problem by fixing the centralized message queue, we proposed a P2P solution on cloud cluster level which abstracts multiple OpenStack cloud instances as one, providing the same user experience as using a single standard OpenStack cloud instance. The abstraction of cloud instances is achieved by deploying a software agent on each cloud instance which works as a message broker, sitting between its affiliated cloud instance and end users. The agent exposes its dedicated API to end users and maps its API to standard OpenStack service endpoint. Besides, these agents form a P2P network and association of agents is based on an inexpensive group membership protocol - Cyclon and distributed search. Our experiment results show that it is feasible to integrate virtual resources across multiple OpenStack cloud instances while abstracting them as a single cloud instance. Compared to a standard OpenStack deployment with same system size, the proposed P2P solution has higher failure resistance to certain operations. We conclude that without considering affection of the centralized identity service - Keystone, the P2P approach has no limitation on the number of cloud instances and its performance improves with an increasing number of cloud instances. Also, system performance and resource

utilization of the P2P solution can be enhanced by implementing optimized cloud selection strategies. Replacing the centralized identity service with a distributed solution remains open for further exploration.

Bibliography

[1] Rimal Bhaskar Prasad and Eunmi Choi. (2011). A service-oriented taxonomical spectrum, cloudy challenges and opportunities of cloud computing. International Journal Of Communication Systems, 25(6), pp.796-819. doi: 10.1002/dac.1279.

[2] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. (2003). Looking up data in P2P systems. Commun. ACM 46(2), pp.43-48. doi: 10.1145/606272.606299.

[3] Korzun, D. and Gurtov, A. (2013). Structured peer-to-peer systems. 1st ed. New York, NY: Springer. ISBN: 978-1-4614-5483-0.

[4] Al-Sakib Khan Pathan, Muhammad Mostafa Monowar and Zubair Md. Fadlullah. 2013. Building next-generation converged networks (1st ed.). Boca Raton, FL: CRC Press. ISBN: 1466507616 9781466507616.

[5] Beverly Yang, Hector Garcia-Molina. (2001). Comparing hybrid peer-to-peer systems, Proceedings of the 27th International Conference on Very Large Data Bases, Rome, Italy, September 11-14, 2001. San Francisco, CA, USA: Morgan Kaufmann Publishers.

[6] Xiang-Wen Huang, Chin-Yun Hsieh, Cheng Hao Wu and Yu Chin Cheng. (2015). A Token-Based User Authentication Mechanism for Data Exchange in RESTful API, 18th International Conference on Network-Based Information Systems, Taipei, Taiwan, September 2-4, 2015. Washington, DC, USA: IEEE Computer Society.

[7] Open source software for creating private and public clouds. (2016). OpenStack. Retrieved from https://www.openstack.org/

[8] Remove DB between scheduler and compute nodes : Blueprints : OpenStack Compute (nova). (2016). Blueprints.launchpad.net. Retrieved from https://blueprints.launchpad.net/nova/+spec/no-db-scheduler

[9] OpenStack Havana Scalability Testing. (2016). Cisco. Retrieved from http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/OpenStack/Scalability/OHS/OHS2.html

[10] Openstack Docs: Architecture. (2016). OpenStack. Retrieved from http://docs.openstack.org/

[11] Quang Hieu Vu, Mihai Lupu and Beng Chin Ooi. (2010). Peer-to-Peer Computing (1st ed.). Berlin, Heidelberg: Springer-Verlag. ISBN: 978-3-642-03513-5.

[12] OpenStack Docs: OpenStack API Documentation. (2016). OpenStack. Retrieved from http://developer.openstack.org/api-guide/quick-start/index.html

[13] Spyros Voulgaris, Daniela Gavidia and Maarten van Steen. (2005). CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Netw Syst Manage, 13(2), pp.197-217. doi: 10.1007/s10922-005-4441-x.

[14] Michael Mitzenmacher. (2001). The power of two choices in randomized load balancing. IEEE Transactions on Parallel and Distributed Systems, 12(10), pp.1094-1104. Piscataway, NJ, USA: IEEE Press.

[15] SQLAlchemy - The Database Toolkit for Python. (2016). Sqlalchemy.org. Retrieved from http://www.sqlalchemy.org/

[16] PEP 333 – Python Web Server Gateway Interface v1.0. (2016). Python.org. Retrieved from https://www.python.org/dev/peps/pep-0333/

[17] Eventlet Networking Library. (2016). Eventlet.net. Retrieved from http://eventlet.net/

[18] Libevent – an event notification library. (2016). Libevent.org. Python.org. Retrieved from http://libevent.org/

[19] Eventlet 0.19.0 Documentation. (2016). Eventlet.net. Retrieved from http://eventlet.net/doc/basic_usage.html

[20] Holger Giese, Stephan Hildebrandt and Leen Lambers. (2012). Bridging the gap between formal semantics and implementation of triple graph grammars. Software and Systems Modeling (SoSyM), 13(1), pp.273-299. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

[21] Memcached - a distributed memory object caching system. (2016). Memcached.org. Retrieved from https://memcached.org/

[22] Rally - OpenStack. (2016). Wiki.openstack.org. Retrieved from https://wiki.openstack.org/wiki/Rally

[23] OpenStack Docs: OpenStack Operations Guide. (2016). Docs.openstack.org. Retrieved from http://docs.openstack.org/openstack-ops/content/scaling.html

[24] OpenStack cascading solution - OpenStack. (2016). Wiki.openstack.org. Retrieved from https://wiki.openstack.org/wiki/OpenStack_cascading_solution

[25] Nathan Lazarus, Christopher D. Meyer and Sarah S. Bedair. (2014). Fractal Inductors. IEEE Transactions on Magnetics, 50(4), pp.1-8. doi: 10.1109/TMAG.2013.2290510.

[26] Francisco Brasileiro, Giovanni Silva, Francisco Araújo, Marcos Nóbrega, Igor Silva and Gustavo Rocha. (2016). Fogbow: A Middleware for the Federation of IaaS Clouds. 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, Colombia, May 16-19, 2016. Piscataway, NJ, USA: IEEE Press.

[27] Rajkumar Buyya, Rajiv Ranjan and Rodrigo N. Calheiros. (2010). InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. Proceedings of the 10th international conference on Algorithms and Architectures for Parallel Processing, 1, pp.13-31. Berlin, Heidelberg: Springer-Verlag. doi: 10.1007/978-3-642-13119-6_2.

[28] Antonio Celesti, Francesco Tusa, Massimo Villari and Antonio Puliafito. (2010). How to Enhance Cloud Architectures to Enable Cross-Federation. 2010 IEEE 3rd International Conference on Cloud Computing, Miami, Florida, USA, July 5-10, 2010. Piscataway, NJ, USA: IEEE Press.

[29] Benny Rochwerger, David Breitgand, Eliezer Levy, Alex Galis, Kenneth Nagin, Ignacio M. Llorente, Rubén Santiago Montero, Yaron Wolfsthal, Erik Elmroth, Juan A. Cáceres, M. Ben-Yehuda, Wolfgang Emmerich and Fermín Galán. (2009). The Reservoir model and architecture for open federated cloud computing. IBM Journal of Research and Development, 53(4), pp.4:1-4:11. Riverton, NJ, USA: IBM Corp.

[30] Benny Rochwerger, David Breitgand, Amir Epstein, David Hadas, Irit Loy, Kenneth Nagin, Johan Tordsson, Carmelo Ragusa, Massimo Villari, Stuart Clayman, Eliezer Levy, Alessandro Maraschini, Philippe Massonet, Henar Munoz and Giovanni Tofetti. (2011). Reservoir - When One Cloud Is Not Enough. Computer, 44(3), pp.44-51. Los Alamitos, CA, USA: IEEE Computer Society Press.

[31] Emanuele Carlini, Massimo Coppola, Patrizio Dazzi, Laura Ricci and Giacomo Righetti. (2011). Cloud Federations in Contrail. Proceedings of the 2011 international conference on Parallel Processing, pp.159–168. Berlin, Heidelberg: Springer-Verlag.

[32] David Villegas, Norman Bobroff, Ivan Rodero, Javier Delgado, Yanbin Liu, Aditya Devarakonda, Liana Fong, S. Masoud Sadjadi, Manish Parashar. Cloud federation in a layered service model. Journal of Computer and System Sciences, 78(5), pp.1330–1344. Orlando, FL, USA: Academic Press.